Modeling, Simulation and Synthesis in an Embedded Software Design Flow for an ARM Processor

Gunar Schirner, Gautam Sachdeva, Andreas Gerstlauer, Rainer Dömer

Technical Report CECS-06-06
May 25, 2006

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-2625, USA
(949) 824-8059

{hschirne, gsachdev, gerstl, doemer}@uci.edu
`http://www.cecs.uci.edu/`

Modeling, Simulation and Synthesis in an Embedded Software Design Flow for an ARM Processor

Gunar Schirner, Gautam Sachdeva, Andreas Gerstlauer, Rainer Dömer

Technical Report CECS-06-06
April 10, 2006

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

{hschirne, gsachdev, gerstl, doemer}@uci.edu
`http://www.cecs.uci.edu`

Abstract

System level design is one approach to tackle the complexity of designing a modern System-on-Chip. One major aspect is the capability of developing the system model irrespectable of the later occurring hardware software split, with the goal to develop both hardware and software seamlessly at the same time and to merge the traditionally separated development flows.

Hardware/software co-simulation is needed for an efficiently integrated design flow. Depending on the design phase, this co-simulation can be performed at different levels of abstraction. Early in the design phase, a a very abstract simulation at the unpartitioned specification level yields fast functional results. On the other end, the cycle accurate simulation of RTL hardware and instruction set simulated software allows an accurate insight to the final system performance.

This report focuses on the software perspective of a co-design/co-simulation environment. In form of a case study, we address three major tasks necessary to build an integrated embedded software design flow: modeling of a processor core (including an instruction set simulator), porting of a RTOS to the selected processor core, and the embedded software generation that includes RTOS targeting of the generated code.

In particular, we have modeled a popular ARM core, the ARM7TDMI, at an abstract level, as well as on a cycle-accurate level using SWARM, an Instruction Set Simulator (ISS) for the ARM core. Furthermore, we have ported MicroC/OS-II, a Real-Time Operating System (RTOS), to run on top of the SWARM ISS. Finally, we implemented a software generation tool. It automatically synthesizes C code, targeted to the selected Real-Time Operating System (RTOS), from the refined design captured in the a system level design language.

We demonstrate our embedded software development flow by use of an automotive application. An example of anti-lock breaks uses a distributed architecture of sensors and actuators connected via a Controller Area Network (CAN). We undergo all steps of the design flow starting with the capturing of the specification model, down to validation of the implementation with an ISS based co-simulation. Our results show that the co-design/co-simulation environment is feasible. All refined models, including the ISS based cycle-accurate model, show a functional correct behavior.

Contents

List of Figures

List of Acronyms

AHB Advanced High-performance Bus. System bus definition within the AMBA 2.0 specification. Defines a high-performance bus including pipelined access, bursts, split and retry operations.

AMBA Advanced Microprocessor Bus Architecture. Bus system defined by ARM Technologies for system-on-chip architectures.

APB Advanced Peripheral Bus. Peripheral bus definition within the AMBA 2.0 specification. The bus is used for low power peripheral devices, with a simple interface logic.

ASB Advanced System Bus. System bus definition within the AMBA 2.0 specification. Defines a high-performance bus including pipelined access and bursts.

ATLM Arbitrated Transaction Level Model. A model of a system in which communication is described as transactions, abstract of pins and wires. In addition to what is provided by the TLM, it models arbitration on a bus transaction level.

BFM Bus Functional Model. A wire accurate and cycle accurate model of a bus.

FCFS First Come First Serve. A scheduling policy in which the job are executed in their arrival order.

HAL Hardware Abstraction Layer. A layer in the BF model, which provides insertion point for implementing PE computation and communication services.

CAN Controller Area Network. A serial communications protocol with a focus for automotive application.

ISA Instruction Set Architecture. The part of the processor architecture visible to the programmer. It sits between the hardware and software.

ISS Instruction Set Simulator. A simulation model that models the datapath of a processor at instruction set level.

IRQ Interrupt Request.

MAC Media Access Control. Layer within the OSI layering scheme.

OSI Open Systems Interconnection. An communication architecture model, described in seven layers, developed by the ISO for the interconnection of data communication systems.

PC Program Counter. A register in a processor that points to the memory location of the instruction in execution.

PE Processing Element. A system component with computation capability, like programmable processor, custom hardware, controller, and IPs.

PIC Programmable Interrupt Controller. An programmable multiplexer that maps from many external interrupts to few internal interrupts. It is available as a slave on the processor bus.

SoC System-On-Chip. A highly integrated device implementing a complete computer system on a single chip.

TLM Transaction Level Model. A model of a system in which communication is described as transactions, abstract of pins and wires.

RTOS Real-Time Operating System. An operating system designed for embedded application and having deterministic nature.

SLDL System Level Design Language.

Modeling, Simulation and Synthesis in an Embedded Software Design Flow for an ARM Processor

Gunar Schirner, Gautam Sachdeva, Andreas Gerstlauer, Rainer Dömer

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA

{hschirne, gsachdev, gerstl, doemer}@uci.edu
`http://www.cecs.uci.edu`

Abstract

System level design is one approach to tackle the complexity of designing a modern System-on-Chip. One major aspect is the capability of developing the system model irrespectable of the later occurring hardware software split, with the goal to develop both hardware and software seamlessly at the same time and to merge the traditionally separated development flows.

Hardware/software co-simulation is needed for an efficiently integrated design flow. Depending on the design phase, this co-simulation can be performed at different levels of abstraction. Early in the design phase, a a very abstract simulation at the unpartitioned specification level yields fast functional results. On the other end, the cycle accurate simulation of RTL hardware and instruction set simulated software allows an accurate insight to the final system performance.

This report focuses on the software perspective of a co-design/co-simulation environment. In form of a case study, we address three major tasks necessary to build an integrated embedded software design flow: modeling of a processor core (including an instruction set simulator), porting of a RTOS to the selected processor core, and the embedded software generation that includes RTOS targeting of the generated code.

In particular, we have modeled a popular ARM core, the ARM7TDMI, at an abstract level, as well as on a cycle-accurate level using SWARM, an Instruction Set Simulator (ISS) for the ARM core. Furthermore, we have ported MicroC/OS-II, a Real-Time Operating System (RTOS), to run on top of the SWARM ISS. Finally, we implemented a software generation tool. It automatically synthesizes C code, targeted to the selected RTOS, from the refined design captured in the a system level design language.

We demonstrate our embedded software development flow by use of an automotive application. An example of anti-lock breaks uses a distributed architecture of sensors and actuators connected via a Controller Area Network (CAN). We undergo all steps of the design flow starting with the capturing of the specification model, down to validation of the implementation with an ISS based co-simulation. Our results show that the co-design/co-simulation environment is feasible. All refined models, including the ISS based cycle-accurate model, show a functional correct behavior.

1 Introduction

System-On-Chip (SoC) design faces a gap between the production capabilities and time-to-market pressures. The design space, to be explored during the SoC design, grows with the improvements in the production capabilities, while at the same time shorter product life cycles force an aggressive reduction of the time-to-market. Addressing this gap has been the aim of recent research work. System-level design is one approach that aims to reduce the time-to-market, accelerate the design process and increase the productivity. It allows for a seamless co-design of hardware and software without special attention to the final hardware software split.

Throughout the system level design flow, the software concerns have to be taken into account. For validation, hardware software co-simulation is needed at different levels of abstraction, starting from the specification level down to the level of cycle accurate simulation of a system using cycle accurate hardware and an Instruction Set Simulator (ISS).

1.1 Problem Statement

In order to reduce time-to-market, designers utilize system level design that reduces the complexity by moving to higher level of abstraction. The system level design process starts with a specification in an System Level Design Language (SLDL) and performs step-wise refinement using a system synthesis tool. Time and cost of software development can be dramatically reduced when it is integrated into the system level design flow.

Besides an SLDL, which is able to capture both hard and software components, three major elements are needed in order to support the software aspect of the design flow:

- Processor models that capture the processor at different levels of abstraction.
- RTOS support for the processor.
- A software generation tool that synthesizes user code targeted for the selected RTOS.

This document reports on a case study for each of these elements as outlined in Figure 1.

1.2 Related Work

System level modeling has become an important issue, as a means to improve the SoC design process. SLDLs for capturing such models have been developed (e.g. SystemC [27], SpecC [20]). Different frameworks for system level design and software synthesis have been developed.

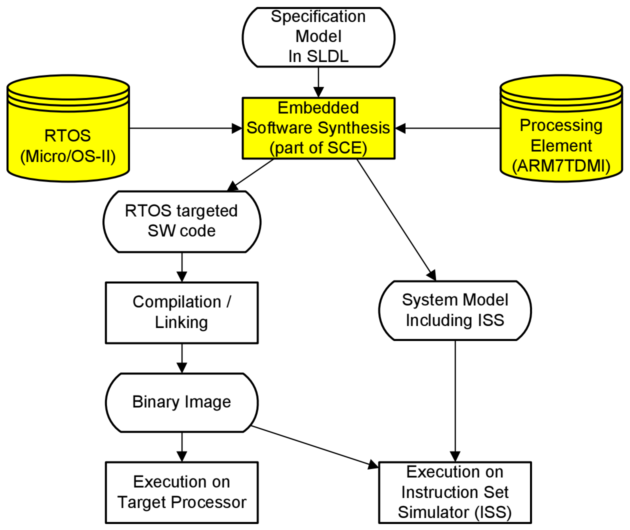

Specification Model In SLDL
RTOS (Micro/OS-II)
Embedded Software Synthesis (part of SCE)
Processing Element (ARM7TDMI)
RTOS targeted SW code
Compilation / Linking
System Model Including ISS
Binary Image
Execution on Target Processor
Execution on Instruction Set Simulator (ISS)

Figure 1: Software synthesis.

Benini et al. [9] introduce MPARM, a platform for multi-processor SoCs design. It includes processor models (ARM), SoC bus models (AMBA), memory models and code development tools (GNU toolchain). It provides a multi-processor cycle accurate architectural simulator. They integrate a number of instruction set simulators by encapsulating each ISS in a SystemC wrapper. The main purpose is system level analysis, including profiling of system performance, execution traces, signal waveform, and power estimation.

Herrara et al. [28] describe an approach for embedded software generation from SystemC. The proposed methodology is based on the redefinition and overloading of SystemC class library elements. Their goal is to use the same SystemC code that allows system-level specification and verification, and SW/HW co-simulation and embedded software generation. However, they impose strict requirements for the specification in the input SystemC model.

The POLIS [8] approach is based on a finite state machine-like representation, called Co-Design Finite State Machine (CFSM). Each element in the network of CFSMs represents a component being modeled in the system. The software synthesis process is performed in two steps, transformation of CFSM specification into an S-Graph, and generation of C code from the S-Graph. This work is mainly for reactive real-time systems and is not designed for general applications.

[11] presents a software refinement flow based on the SystemC SLDL. It too makes use of integrating an instruction set simulator for hardware/software co-simulation.

SoC Designer with MaxSim Technology [6], is a commercial tool set for fast modeling, simulation and debugging for complex System-on-Chips designs. It provides a graphical user interface for interactive system design, modeling and simulation. It provides cycle-accurate and cycle-approximate models with support for SystemC. VaST Systems [36] offers a similar set of tools with a focus on embedded systems. Furthermore, CoWare [13] presents with the Virtual Platform Designer an integrated solution for the custom platform software design.

1.3 Outline

This document is organized as follows, Section 2 describes modeling of the processor (we selected the ARM7TDMI [7]). This section will describe the abstract models, as well as the integration of a suiting ISS (SWARM [14]). Second, Section 3 reports on the porting of an RTOS (μC/OS-II [29]) toward the selected processor core. Thirdly in Section 4, we will give an overview how the generated SW code is targeted to the selected processor core.

Combining these three elements enables an integration of the software development and simulation into the system level design. The advantages of this integration are shown with an example of car anti lock breaks in Section 5.

2 Processor

The first key point for supporting a software development flow is to integrate a processor model into the system flow. By capturing the fundamental characteristics of the processor it makes the sytem design flow aware of a processor as a processing element, allows to map software to it and to estimate the performance. Therefore we will first describe the selected processor and its modeleding. We capture processor at two levels of abstraction. First as an abstract behavioral model that yields early exploration results and second as an integration of an Instruction Set Simulator (ISS) for a cycle accurate execution of the target binaries.

We chose an ARM core as a modeling example, since ARM is the industry leading provider of 32-bit embedded RISC microprocessor with almost 75% of the market [4]. In particular, we focused on the ARM7TDMI.

2.1 ARM7TDMI

According to [2] the ARM7TDMI core is at the present the industry's most used 32-bit embedded RISC microprocessor. We therefore selected this core as a basis for the embedded software synthesis. The core provides high performance with very low power consumption. It is a RISC architecture, provides high instruction throughput and real-time interrupt response. The ARM7TDMI core is 100% binary compatible with other ARM7 family cores and forward-compatible with the ARM9, ARM9E and ARM10E families. The processor core is supported by a wide range of operating systems.

The ARM7TDMI has a Von Neumann architecture, with a single 32-bit data bus carrying both instructions and data. Only load, store and swap instructions can access data from memory and data can be 8-bit (bytes), 16-bit (halfwords) and 32-bit (words) [7].

2.1.1 Instruction Pipeline

As shown in Figure 2, the ARM7TDMI contains a three-stage pipeline.

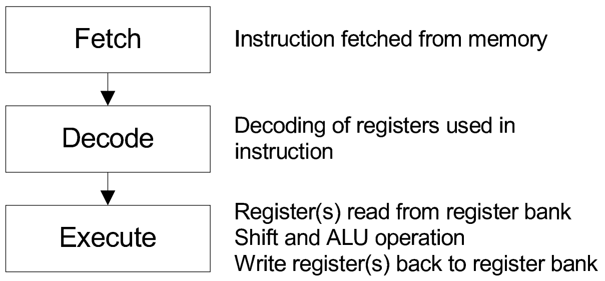

Fetch
Instruction fetched from memory
Decode
Decoding of registers used in instruction
Execute
Register(s) read from register bank
Shift and ALU operation
Write register(s) back to register bank

Figure 2: ARM7TDMI Instruction Pipeline (Source [7]).

The three-stage pipeline allows concurrent operation of the processing and memory system. While one instruction is executed, its successor is being decoded at the same time and yet another instruction is being fetched from the memory. The Program Counter (PC) refers to the instruction being fetched rather than the instruction being executed. Therefore, the value of the PC is usually ahead by two address locations with respect to the instruction in the execution stage.

2.1.2 Architecture

The ARM7TDMI [2], an implementation of the ARMv4T architecture, is a 32-bit RISC processor with a unified 32-bit data and address bus. Figure 3 shows the architecture of the ARM7TDMI processor core.

For increased memory efficiency, it supports both the ARM and the Thumb instruction sets. While the ARM instruction set is 32-bit wide and offers the full instruction flexibility, the Thumb instruction set is limited to 16-bit. It only implements the most frequently used instructions, which according to [7] cover 65% of typical ARM code.

Each 16-bit Thumb instruction is internally translated to the corresponding 32-bit counterpart. Thumb instructions operate on the same register set, the same 32-bit ALU and 32-bit memory address as the 32-bit ARM instructions. With that, the Thumb instruction set allows for very compact code, while delivering at the same time the performance of the 32-bit architecture.

The ARM7TDMI processor can be extended by custom co-processors. Up to 16 co-processors can be tightly coupled with the ARM7TDMI core for implementation of highly specialized additional instructions. Detailed information about cycle counts and the two instruction sets can be found in [2, 7].

To react to external events, the ARM7TDMI has two low active, level sensitive interrupts signals (nIRQ and nFIQ). nIRQ triggers the general purpose interrupt and nFIQ the Fast Interrupt Request (FIQ) with a higher priority that the nIRQ. In addition to the higher priority, the FIQ is optimized for faster execution. It uses a reduced number of registers that are exclusively available during interrupt execution. A lower number of registers, minimizes the overhead of context switching and reduces the interrupt overhead.

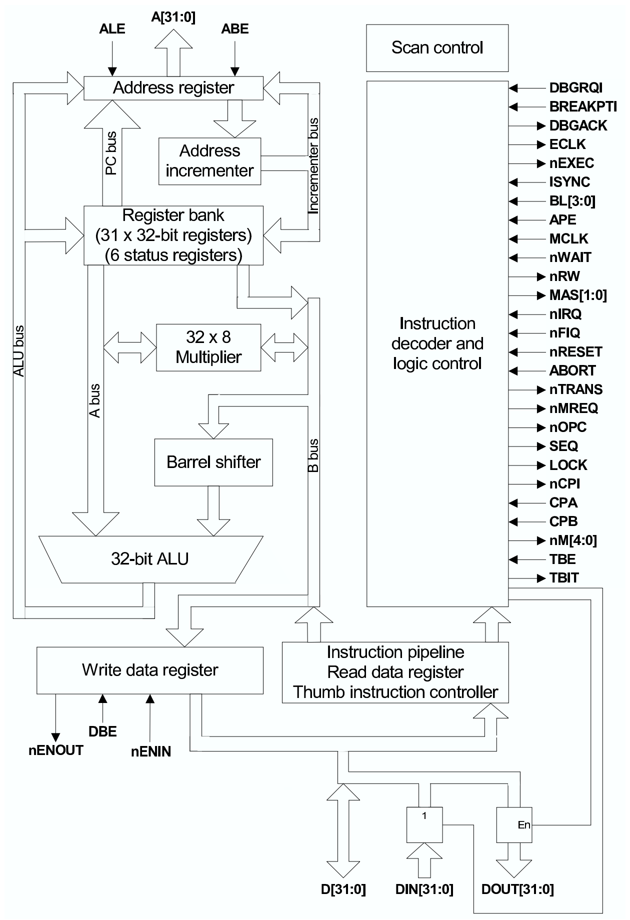


Figure 3: ARM7TDMI processor core (source [7]).

2.1.3 Bus Architecture

The ARM7 Thumb family processor cores are designed for use with AMBA on-chip bus architecture. ARM defined with the Advanced Microprocessor Bus Architecture (AMBA) [3] a widely used on-chip bus system standard. It contains a group of busses, which are used hierarchically as shown in Figure 4. The Advanced High-performance Bus (AHB) is a system bus designed for connecting high-speed components including ARM processors.

Although the initial ARM7TDMI design did not include a direct connection to the Advanced High-performance Bus (AHB), it can be connected through a wrapper, provided with the AMBA Design Kit [2], to the AMBA AHB.

The AHB is a multi-master bus that operates on a single clock edge. High performance is achieved

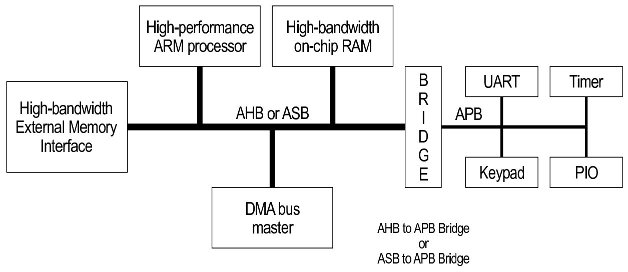

High-performance ARM processor
High-bandwidth on-chip RAM
High-bandwidth External Memory Interface
AHB or ASB
BRIDGE
UART
Timer
APB
Keypad
PIO
DMA bus master
AHB to APB Bridge or ASB to APB Bridge

Figure 4: AMBA bus architecture (Source [3]).

by a pipelined operation that overlaps arbitration, address, and data phases, and by the usage of burst transfers. Split and retry transfers allow the slave to free the bus if the requested data is temporary unavailable. The AHB also employs a multiplexed interconnection scheme to avoid tri-state drivers.

2.2 Abstract Processor Modeling

In order to perform software synthesis in our design and refinement flow, based on the System-on-Chip Environment (SCE) [1], the target processor has to be captured as a Processing Element (PE) [22]. A PE is a system component that processes data (performs computation) by executing application specific algorithms. Examples for a PE are custom designed hardware or a programmable element (e.g. a processor). Each PE is captured at multiple levels of abstraction. Throughout the design process a model with an increasing amount of detail will be used.

A *Behavioral Model* of a PE is the most abstract model, which describes only the basic characteristics. It is used for the PE allocation and mapping of computation behaviors of the specification model during architecture exploration.

Later in the design flow the *OS Model* is used as a template for inserting an RTOS model for abstract scheduling.

The *Bus Functional Model* is a pin-accurate model, that adds communication layers describing communication behavior of the component. For a programmable PE, a cycle-accurate *Instruction Set Model* is also used for clock cycle-accurate simulation of the PE.

Next sections describe in more detail each model for the selected processor, the ARM7TDMI.

2.2.1 Behavioral Model

A PE behavioral model is used for PE allocation during architecture exploration and defines the basic characteristics of a PE. It enables the refinement flow to map user behaviors to it and to analyze the performance in various aspects.

The ARM7TDMI behavioral model contains three aspects. It contains annotations for general database management, attributes and weight tables.

The annotations for the database management contain the basic information needed for managing the PE in the database. For example it includes a categorization of the ARM7TDMI PE as a general purpose core that is a 32-bit RISC. It also contains references to the more detailed models (OS Model, Bus Functional Model and Cycle Accurate Model). Furthermore it contains the bus connectivity of the processor and its address range.

Figure 5 shows the behavioral model of ARM7TDMI. The ARM7TDMI is connected through an AHB wrapper to the AMBA AHB. For simplicity however, we integrated this wrapper into our ARM7TDMI model as if the processor had a direct interface to the AHB bus.

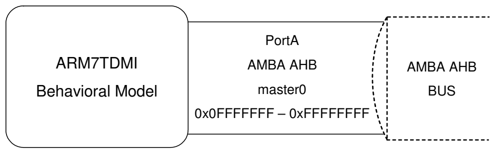


Figure 5: ARM7TDMI behavioral model.

Attribute annotations describe the basic characteristics of a PE. The attributes include, clock frequency, MIPS (million instructions per second), power consumption, instruction width, data width, data memory and program memory. These attributes of the ARM7TDMI have been defined according to the documentation [7, 2, 4]. Some of the attributes are defined as ranges, to adapt the processor to the current design needs. As one example, the clock frequency can be set between 12.5 MHz and 75 MHz to meet the application demands.

The main information of an PE behavioral model is captured in form of weight tables. The weight tables are used for interpretation of the generic profiling results. The profiler collects generic profiling information during simulation for each PE. It counts the execution times for each basic block and determines the amount of instructions for each block. The generic results are then interpreted for a particular PE using the PE's weight tables. This interpretation yields the first application and target specific performance data. The interpreted profiling results are then used for comparative analysis of different design alternatives.

The ARM7TDMI behavioral model contains two weight tables. The computation weight table contains number of cycles to execute each possible operation per data type. The second weight table contains parameters for calculating the code size. Each item in the latter weight table indicates the number of instructions required to perform an operation with a certain data type.

Using the weight tables for an interpretation of the profiling results dramatically simplifies the process. Despite this simplification, the results are useful for exploration. When comparing the performance estimation of two alternative designs the relation between these results is of interest for design decision. As long as the estimation fulfills the fidelity property [19], which requires the relation between the estimation results of the two designs to be identical to the relation in a real implementation, an absolute accuracy is not required. For simplicity, we made the following assumptions for populating the weight tables.

1. ARM7TDMI has a three-stage pipeline in which an instruction is first fetched, then decoded and finally executed. We assumed for the clock-cycle count that the instruction is in the execution stage with fetching and decoding already performed in the previous cycles. Hence, fetch and decode cycles are not included in the cycle count.

2. We assume all operands to be are available in registers and furthermore the absence of data hazards. Hence, our cycle count metric captures the best case results.

3. The weight tables reflect the cycle count without any co-processor[1].

4. The number of instructions required for floating point operation highly depends on the floating point emulation library or the hardware support. To have some estimation, we assumed a four times longer execution for float data types over an integer data type. Operations on double data types are assumed to be eight times longer. For example, we captured a float addition to require four instructions in four cycles.

In summary, we assume best case conditions in the weight tables.

2.2.2 OS Model

If more than one behavior is mapped to a PE, the mapped behaviors have to be scheduled. A programmable PE allows only sequential execution at a time. One scheduling approach is dynamic scheduling as executed by an RTOS on the target system. However, it is not desirable to execute a complete RTOS at an early design stage due to the simulation overhead. Therefore, an abstract RTOS model as described in [23] is used for exploration of different scheduling policies.

The ARM7TDMI OS model provides a template. Later, the refinement tools fill this template based on the selected scheduling policies and parameters. The RTOS model is not a complete RTOS, but implements the necessary concepts including task management, real-time scheduling, task synchronization and interrupt handling. The PE's OS model is inserted as another layer around the behavioral model.

2.2.3 Bus Functional Model

The Bus Functional Model is a pin-accurate model, which contains information about the PE's communication interfaces and describes the communication behavior. Additionally, it contains a template for computation functionality that will be filled by adding communication layers on top of the behavioral model. The general requirements for the PE Bus Functional Model (BFM) can be found in [22].

The ARM7TDMI BFM consists of a behavior hierarchy as shown in Figure 6. The outer shell is the

[1] For the ARM7TDMI, a Vector Floating Point (VFP) coprocessor is needed for a hardware support of floating point operations.

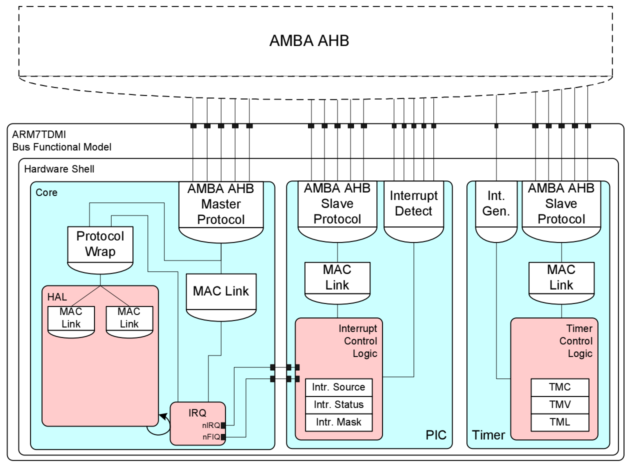


Figure 6: ARM7TDMI bus functional model overview.

bus functional shell that defines all pins of the processor. It contains three parallel executing behaviors: the processor core, a Programmable Interrupt Controller (PIC) and a timer. The processor core models the bare core as described in the ARM documentation. The PIC is an interrupt controller that maps 32 external interrupts to the two core interrupt lines (nIRQ, nFIQ). The timer, disabled by default, can generate periodic interrupts. The BFM communicates to the outside via the AMBA AHB and interrupts can be triggered by one of the 32 external interrupt lines.

The bus implementation of the AMBA AHB itself is taken from the bus database. The layer based implementation is realized in channels as described in [35]. Each layer is implemented in a separate channel. In order to communicate, the according channel is instantiated inside the behavior.

The processor core contains two behaviors that are executing in parallel. The Hardware Abstraction Layer (HAL) shell acts as a template. Throughout the synthesis the template will be filled with the user computation behaviors. The Interrupt Request (IRQ) behavior contains an interrupt handling logic that communicates with the PIC and triggers execution of the interrupt handlers.

Both behaviors the Hardware Abstraction Layer (HAL) and the IRQ communicate through the AHB master interface. The protocol layer interface is wrapped by the the protocol wrap channel before it connects to the HAL shell. The wrapper disables the interrupt handling while a bus transaction is in progress. Doing so avoids preemption of a partially finished transaction and is needed to maintain accurate protocol timing.

The IRQ behavior has two connections to the outside, it terminates the nIRQ and nFIQ lines that signal an interrupt from the PIC and a media access layer link channel for the data connection to the PIC. This connection is used for communicating with the PIC, in order to determine the actual source of an interrupt and to clear the interrupt after handling it.

The PIC behavior is connected as a AHB slave to the AHB bus, which is implemented through the AMBA AHB Slave Protocol channel. For a Media Access Control (MAC) layer access it uses the slave version of the link layer. Additionally a channel for the interrupt detection is instantiated inside the PIC that performs the recognition of an interrupt from an external source. The Interrupt Control Logic behavior listens to the interrupt detection logic, sorts the incoming interrupts by priority and signals an aggregated interrupt to the core.

For the bus functional model, we captured a Programmable Interrupt Controller (PIC) according to NEC's System-on-Chip Lite+ definition in [25], with a few simplifications. The PIC provides 32 maskable interrupts, with a 2-level programmable priority. It is build out of 4 basic blocks: low level interrupt detection modules, the interrupt control logic, register behaviors and the slave channels for the bus interface. The interrupt detect modules are responsible for recognizing the interrupt condition on PE's interrupt lines and to change the interrupt status register (*Intr. Status*) according to the detected state.

The control logic behavior combines the interrupt information of all detect modules, sorts them by priority and signals the interrupt condition to the processor core through nIRQ and nFIQ if a non-masked interrupt is active. It then provides in *Intr. Source* the highest priority active interrupt, which the processor evaluates to select the interrupt service routine to execute. The PIC also contains a mask behavior, so that the core as the possibility to mask individual interrupts through the *Intr. Mask* register. For a clean startup, all interrupts are disabled by default.

Additionally, we modeled a programmable timer unit also following NEC's System-on-Chip Lite+ definition in [25]. The timer can be programmed to generate periodic interrupts. It is used in the later described cycle accurate model for the actual RTOS to keep track of time. The timer's activity (reset, count-up, count-down) is controlled by the Timer Control Register (TMC) and the period through the Timer Load Register (TML). These registers are available via the AHB interface. The timer is disabled by default. It will be enabled by the RTOS running in the cycle accurate model as described in Section 3.2.4.

2.3 Cycle-Accurate Instruction Set Simulator

After the final stage of the embedded software synthesis the generated binaries have to be tested and the final system performance has to be evaluated. For that purpose an Instruction Set Simulator (ISS) is required, that provides an accurate model of a processor (besides execution on the actual target processor) and accepts the binary code of the target processor.

The embedded C code is generated in the software synthesis as the final step of software refinement (see Section 4). The C code is then compiled and linked against all required libraries (e.g. RTOS library) to yield the final target binary. For the validation of the final target binary, co-simulation of hard and software can be used, which requires an ISS in the system level design flow. The ISS model, as part of the database, then replaces parts of the bus functional processor model. A bus functional model of the target design with an integrated ISS offers true hardware/-software co-simulation capabilities and yields cycle accurate timing information for all aspects: software execution, hardware execution[2] and communication.

Next we will describe the selection of an ISS from multiple candidates and cover its integration into the database.

2.3.1 Selection of an ISS

An Instruction Set Simulator (ISS) is a simulation model that models the micro architecture of a processor at the instruction set level. An ISS reproduces the execution behavior of the actual processor by reading binary instructions, decoding and executing them while incrementally maintaining all relevant internal state information of the processor. It emulates all components accessible to the target assembly code, such as registers, program counter and processor status flags. Additionally, it can provide debug access to internal state information of the processor that are usually hidden in a real hardware (e.g. detailed status of the pipeline). Using an ISS eases software development and debugging by providing a deterministic execution; especially useful for analyzing race conditions.

For integration into the system level design flow, we considered several instruction set simulators for the ARM7 core:

1. SimpleScalar [10]
2. GDB ARMulator [15]
3. SWARM [14]
4. ARM ARMulator [5]

The selection process of the ISS was guided by the requirements of the development and simulation environment. Especially the integration into the co-simulation environment poses some restrictions. For example, an ISS is typically implemented to run as a stand alone process. However, for a co-simulation it has to run within the context of the co-simulation engine. We have considered the following requirements for the ISS selection:

1. Cycle-accurate simulation of ARM7 micro-architecture.
2. Cycle callable interface (API) for cycle-by-cycle simulation.
3. Access to the bus interface of the ISS.
4. Ability to handle interrupts and optionally to provide an interrupt controller.
5. Availability of source code for the ISS.
6. Availability without license cost.

[2] Assuming the custom hardware has been refined to RTL.

A key requirement for the ISS is a cycle callable API, to call it cycle-by-cycle while simulating along with other components within the design. The ARM7 core will be connected to different system-on-chip components through the AMBA bus. Therefore, the ISS has to provide an access to the bus interface to integrate with our AHB bus model. For synchronization with other components, the ISS has to implement interrupt handling from different sources. For an easier integration and adaptation the source code of the ISS should be available.

Figure 7 compares each considered simulator according to the defined criteria. The following paragraphs discuss each simulator individually.

SimpleScalar [10] is a tool set consisting of compiler, assembler, linker, simulator and visualization tools. The SimpleScalar simulator supports the ARM7 instruction set, but models the SA-1100 micro-architecture. Both StrongARM SA-1 core and ARM7 core are based on ARMv4 ISA but have different micro-architectures. While StrongARM SA-1 core has a five-stage pipeline, the ARM7 implements a three-stage pipeline. Therefore SimpleScalar's StrongARM model would yield inaccurate timing information for the ARM7.

The **GNU Project debugger (GDB)** [15] contains an ARM emulator called GDB ARMulator. The GDB ARMulator emulates ARMv4T and ARMv5ET, and supports both Thumb and DSP extension. However, it lacks support for the interrupts. Furthermore it is an instruction-by-instruction simulator that only simulates the functionality and not the micro architecture. Therefore it does not offer accurate cycle count information.

SWARM (Software ARM) [14] is a C++ based model of the ARM7 processor's datapath at the bus level. It supports ARMv4 Instruction Set Architecture (ISA), and has the acceptable limitation of requiring binaries generated from gcc (in the COFF format). The simulator does not support the 16-bit Thumb extension. SWARM implements a cycle callable interface and provides cycle accurate simulation. Furthermore, is supports interrupt handling.

ARM itself offers an ISS, called **ARMulator** [5], for its processor cores as a part of its RealView Developer Suite. The ARMulator supports many ARM core processor families, including ARM7, ARM9E and ARM10E. It provides a cycle accurate simulation with support for interrupts and exceptions. However, it requires a license and is not freely available.

Weighting the features of each ISS candidate, we selected SWARM for our work. SWARM provides a cycle accurate simulation for the ARM7 processor core and models the ARM processor core data path at the bus level. It provides a cycle-by-cycle callable interface and supports handling interrupts and exceptions (it even includes a interrupt controller). Being a university research project, SWARM is freely available in source code. Additionally, SWARM is the only simulator that individually simulates the ARM processor core, while the remaining three simulators are a part of a larger tool set and support ISAs of other processor cores not needed. We expect an easier integration due to the reduced complexity.

2.3.2 SWARM (Software ARM)

SWARM [14] is a modular constructed simulator for the ARM processor's datapath at the bus level. It has been implemented as a hierarchy of C++ classes, which allows selective use of either the simple core only or of a core with cache. SWARM models the internal datapath based on the ARM7 core. External elements, such as the interrupt controller, are based on definitions of StrongARM. Since it models access to external elements, the SWARM includes a bus interface, which can be used for connecting additional components.

The instructions from the binary stream are decoded into control signals that contain information for managing the internal datapath and the bus interface. SWARM supports most of the instructions of the ARMv4, including data processing, data transfer, load/store and co-processor register transfer. However, it does not completely implement the ARM instruction set and does not support the Thumb instruction set. SWARM requires a COFF binary image produced by gcc. It includes a timer, LCD and UART controller, which are modeled after the specification of the Intel SA-1110 processor [12]. It also includes rudimentary support for a basic system co-processor to aid extending the ISA.

SWARM implements two interrupt signal lines

ISS	Processor Core / Microarcitecture supported	ISA Modeled	ISA complete Thumb suppport	Cycle-by-cycle Callable	Interrupt Handling	Cost	License
SimpleScalar	SA-1 core SA-11xx	ARMv4T	Yes Thumb support	Yes	No but can be extended to include	Free	SimpleScalar LLC license Acad. Non-comm. & comm. Academic.non-comm:free
GDB ARMulator	ARM 6,7	ARMv4T,5TE	Yes Thumb Support	No Instr-by-Instr	No/Yes Stops on interrupt	Free	GNU General Public License
SWARM	ARM 6,7	ARMv4	Binaries compiled with gcc No Thumb Extension	Yes	Yes Interrupt Controller	Free	GNU General Public License
ARM ARMulator	ARM 7,9,9E,10E	ARMv4T,5T,6	Yes	Yes	Yes	"$"	ARM Limited Academic License

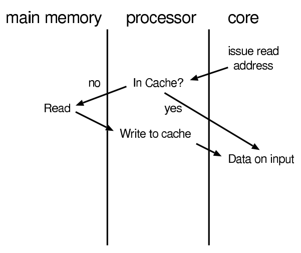
Figure 7: Instruction-set simulators for ARM core.

(IRQ and FIQ) to interrupt the core and basic interrupt handling. It also includes an 32-bit interrupt controller based on the Intel SA-1110 [12] for synchronization with external peripheral devices.

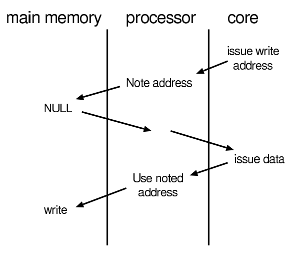
SWARM attempts to realistically simulate the memory hierarchy for a ARM machine and even includes a cache model. However, the external memory bus interface is not complete. It provides an abstract untimed bus interface with 32-bit address and data values. SWARM utilizes an internal memory of 12 MB, starting at address zero, where the COFF binary file is loaded into during initialization and where the execution starts from.

Figure 8 depicts the decisions during a write and a read memory access in SWARM. Accessing the memory is divided into three levels: the ARM core interface, the ARM processor interface that contains cache, and the main memory. In case of a read cache miss, the ARM processor halts the core, fetches the data from the main memory into the cache and then releases the core. In case of a read cache hit, the data is available in the next cycle and the core execution is not halted. Memory writes are implemented as write through that immediately appear in the external memory.

SWARM has been implemented as a hierarchy of C++ classes as outlined in the Figure 9. SWARM, being a research project at the University of Glasgow, has not been completely implemented. However, the current version is sufficient to execute binaries compiled with gcc. Even so, the implementation contains some restrictions.



(a) Read



(b) Write

Figure 8: SWARM memory accesses (Source [14]).

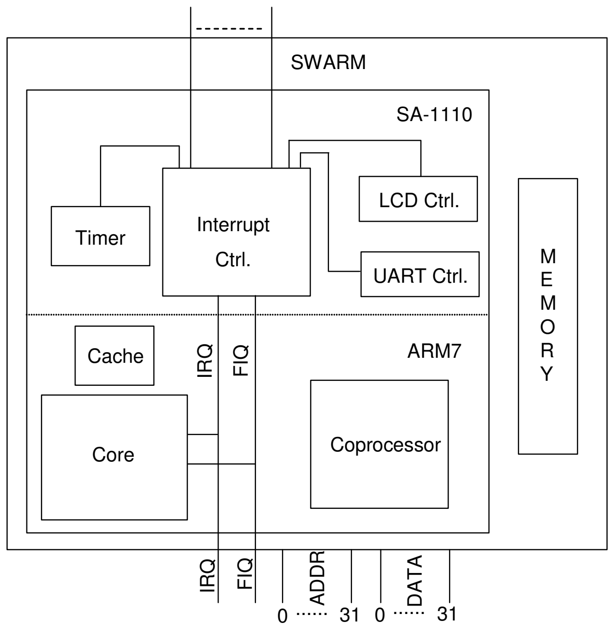


Figure 9: SWARM implementation (source [14]).

Our work includes some changes to SWARM in order better to match the real ARM core. We have performed three main alterations.

First, the interrupt lines nIRQ and nFIQ in the SWARM core have been incorrectly modeled as edge sensitive. Thus, an interrupt will not be detected if the interrupt is disabled during the actual interrupt event. Even after enabling the interrupt, the according interrupt service routine is not executed. The ARM7TDMI core, on the other hand, is level sensitive to the two interrupt signal lines. Hence, enabling the interrupts after the occurance of an interrupt, with the interrupt line still being active, will trigger execution of the interrupt handler. We modified the SWARM core to implement level sensitive interrupts.

Second, we changed SWARM to avoid preemption of an nIRQ interrupt. An interrupt triggered through nFIQ has higher priority than an nIRQ triggered interrupt. Thus, an nFIQ interrupt can preempt an nIRQ interrupt. However, this preemption of an nIRQ interrupt is not handled correctly in SWARM, which loses the order of execution. As a workaround we have disabled the nFIQ during execution of an nIRQ interrupt. Although this does not yield timing accurate results, it allows correct execution. As future work we plan to investigate fixing the FIQ interrupt handling.

Third, the interrupt controller included in SWARM is incomplete. For reasons of future optimization we have decided not to use the included PIC, but modeled a PIC as a part of the bus functional model external to SWARM and disabled the SWARM internal PIC.

2.3.3 Cycle-Accurate PE Model

A cycle-accurate model of a programmable PE provides cycle accurate simulation of the PE's instruction set and is called the instruction set model. The instruction set model replaces parts of the bus functional model. Therefore, the interface of the cycle-accurate model must exactly match the interface of the corresponding bus-functional model [22]. In other words, the cycle-accurate model is a SpecC behavior with identical external ports as the bus-functional model, it adds a refined cycle-accurate timing.

In order to reach the cycle-accurate execution we integrate the SWARM ISS into the database of our refinement flow. The refinement tool flow uses the SLDL SpecC [20] for capturing the design and the database elements. While SpecC is a superset of C language, SWARM, on the other hand, is based on C++ classes. To integrate SWARM, we first created a C wrapper around the ISS (SWARM) that provides a C-level API, which can be called from SpecC behavior.

The SWARM with its C wrapper is embedded into a SpecC behavior *ARM7TDMI_ISS*. The wrapping behavior *ARM7TDMI_ISS* calls the ISS cycle by cycle and interfaces with the remaining design (e.g. external slaves on the same bus). It uses the C-API to translate between SWARM events and external events. As such it detects a SWARM bus access on the SWARM abstract bus interface and calls the channel of the bus functional model to execute the requested bus transfer. On the other hand it monitors the interrupt inputs and triggers an SWARM internal interrupt, should an interrupt occur. The wrapping behavior *ARM7TDMI_ISS* advances the time in the SpecC simulation according to the clock definition of the utilized ARM7 processor core.

Figure 10 shows the instruction set model of the ARM7TDMI. It looks very similar to the bus functional model and reuses the same programmable interrupt controller. However, instead of the previously used core shell for abstract execution, it instantiates wrapping behavior *ARM7TDMI_ISS* that contains the SWARM. The *ARM7TDMI_ISS* connects to the AHB via the AMBA AHB master protocol channel. Note that due to the incomplete state of the SWARM internal PIC and for future optimization, we do not use the SWARM internal PIC. We neither use the LCD, UART controller or the internal timer.

As shown in Figure 10, the instruction set model has the identical pin level interface as the ARM7TDMI bus functional model including wires for the AMBA AHB master and slave interface as well as the interrupt wires. As in the BFM, the AHB wires are connected to the inlined master and slave protocol channels, which in turn are used by the media access layer channels (AMBA Master MacLink and AMBA Slave MacLink).

The two low active interrupt wires nIRQ and nFIQ directly connect the PIC to the *ARM7TDMI_ISS*. The PIC signals a non-masked interrupt from any of the 32

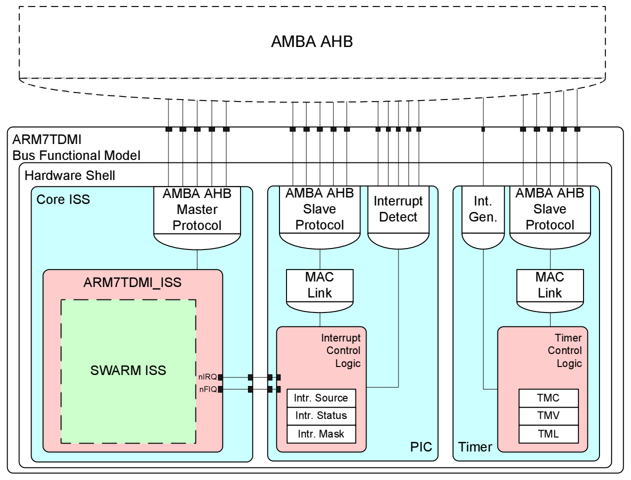


Figure 10: SWARM instruction set model.

incoming interrupt wires to the ISS behavior through the either the nIRQ or the nFIQ line. The wrapping behavior *ARM7TDMI_ISS* checks both lines for each cycle of the ISS. It forwards the interrupt signal to the SWARM ISS by calling the C-level API for setting and clearing the IRQ/FIQ request. This triggers execution of the interrupt service routine within the SWARM simulated code, which in turn then communicates through the AMBA AHB with the PIC to determine the interrupt source.

Upon startup the wrapping behavior *ARM7TDMI_ISS* initializes the SWARM, which loads the binary file of the user program into the SWARM internal memory. Execution starts then at address zero (SWARM internal memory). The wrapper *ARM7TDMI_ISS* calls the SWARM cycle-by-cycle in an endless loop. For each iteration, the *ARM7TDMI_ISS* behavior checks external interrupts, drives the ISS's asynchronous inputs and drives the external bus interface if an according I/O instruction is executed within the SWARM ISS.

In a non-I/O processor cycle, the *ARM7TDMI_ISS* advances SpecC time for one processor clock period and advances the ISS by a single cycle. In case of external bus read or write, the *ARM7TDMI_ISS* simulates the bus by calling the bus protocol channel *AMBA AHB Master Protocol* imported from the bus database. Calling the protocol channel advances the simulation time depending on the bus state and the selected slave, the *ARM7TDMI_ISS* then advances the ISS internal cycle count accordingly. It updates the PE ports for every external I/O operation according to the processor state and bus state. The instruction set model is clock-cycle accurate and integrates with external components in the design through the bus functional interface and interrupts.

3 Real-Time Operating System

During the refinement process the designer may assign multiple behaviors to one software processing element (CPU). Due to the inherent sequential execution nature of a processor, behaviors have to be scheduled either statically or dynamically. An RTOS is needed to run on the target processor for dynamic scheduling.

In the software synthesis stage, C code is generated from the behaviors representing the software application running on the PE. The generated C code is cross compiled to the target processor's instruction set using a cross compiler. The final executable is generated by linking it against a customized RTOS. In the RTOS targeting stage, an actual RTOS is selected from the database and is inserted in the code for providing necessary scheduling services.

In order to target an RTOS to a particular processor, the RTOS needs to be first adapted for the selected target processor. This chapter starts with the selection of an RTOS for the target ARM core. It then focuses on the integration of the selected RTOS.

3.1 RTOS Selection

An RTOS is an operating system that has been designed for real time applications. The RTOS, through its scheduling algorithms and deterministic nature, guarantees that the system deadlines can be met. Or to be more accurate, an RTOS provides services timing deterministic so that it does not hinder the system from meeting the deadlines. Deterministic execution times are a strict requirement, an additional goal is to minimum response time for interrupts. An RTOS provides the system with the basic services of scheduling, multitasking and synchronization.

Many RTOSs are available, ranging from cost free to commercial ones. Therefore, we describe first our selection process. We have considered the following RTOSs for our work:

1. RTEMS [30]
2. TinyOS [31]
3. eCos [16]
4. μC/OS-II [29]

Almost any RTOS can be adapted for targeting on the ARM core. Therefore, based on the target processor we have no strict requirement for the RTOS. However, while considering the concurrency and mapping performed in the refinement flow, there were few requirements that we considered while selecting the RTOS. The requirements that we took in to account are listed below:

1. The RTOS should be able to support multitasking or concurrency.
2. It should have some mechanism for intertask communication and synchronization.
3. Priority and first-come-first-serve scheduling has to be supported.
4. The source code should be easy to adapt for an ARM core.
5. The size for the RTOS should be small.
6. It should be freely available with detailed documentation.

The first three requirements were the most important, although very basic, requirements for the RTOS. The remaining requirements were considered for an easy adaptation to the ARM core.

Figure 11 shows a comparison of the four considered RTOSs. It contains the information based on our metrics mentioned above.

Note that even though, μC/OS-II is no longer freely available, we considered this RTOS in an older version available with source code in the book [29].

TinyOS [31], is an event-driven architecture and offers only a limited concurrency. It supports only a single process, therefore does not include process management. Tasks are scheduled with a simple FIFO scheduler and cannot preempt other tasks. Only interrupts can preempt a task.

The three RTOSs - **RTEMS**, **eCOS** and **μC/OS-II** are suitable for our work as they supported multitasking and have some mechanism for intertask communication. Even though RTEMS offers more features as compared to the other two, we selected μC/OS-II for our work due to the following reasons. The μC/OS-II has a very small footprint, yet at the same time it does provide all necessary features. Its source code is well organized, understandable and can be adapted easily to the ARM core. In addition to that we had previous experience with this RTOS.

3.2 MicroC/OS-II

μC/OS-II [29] is a multitasking real-time kernel that provides an execution environment for many tasks, where each task can utilize system resources. It provides transfer of execution from one task to the other, so that resources can be used efficiently and timing deadlines can be achieved. μC/OS-II provides low latencies for the kernel services. In order to achieve timeliness, priority scheduling is supported. Each task is assigned a priority and is scheduled according to it. Furthermore, preemption is supported, a higher priority task may preempt execution of a lower priority task in order to perform a time-critical function.

μC/OS-II is ROMable - it can execute as firmware from the ROM of an embedded systems. It is portable since it has been implemented mostly in ANSI C and contains only a small amount of assembly code for adaptation to a particular processor core. In fact, it has been ported to more than 40 different processor architectures ranging from 8- to 64- bit microprocessors, microcontrollers and digital signal processors [29].

μC/OS-II provides a fully **preemptive** real-time kernel and **priority** scheduling that always executes the highest priority ready task. It supports multitasking, where the application software can define up to 56 tasks (8 tasks are reserved for μC/OS-II). Being a real-time kernel, the execution time of most of the μC/OS-II functions and services is **deterministic**. In other words, the time it takes to execute a function can be estimated, which is necessary to make any real-time guarantees.

RTOS	Cost	Footprint	Portable to ARM	Scheduling	Concurrency	IPC	Debug	API
RTEMS	Free	64K~128K	Yes	SCHED_RR, SCHED_FIFO	pThreads	Semaphores, Mutexes, Condition-variable, Pqueues	GDB, DDD, Debug over-ethernet, serial, BDM	RTEID/ORKID, uITRON , POSIX
TinyOS	Free	400bytes	Yes	FIFO, Premptive	Limited - Two Threads of Execution: Tasks & Hardware event handler	Exclusive shared memory	TOSSIM (simulator + Debugger)	Custom
eCos	Free	20K~200K	Yes	SCHED_RR, SCHED_FIFO	Yes	Semaphores, Mutexes, Condition-variable	GDB	uITRON, POSIX
µC/OS-II	$	2K~20K	Yes	Fixed Priority, Preemptive	Yes	Semaphares, Mutexes	GDB	POSIX

Figure 11: RTOS survey (source Jinhwan Lee).

*µ*C/OS-II is a small real-time kernel with a memory footprint of about 20KB. It is a good candidate for application specific RTOS configuration, since it can be scaled down in footprint if the application does require fewer features (down to 2K bytes of code space according to [29]).

3.2.1 MicroC/OS-II Structure

*µ*C/OS-II is small with about 5,500 lines of code, mostly in ANSI C. The source code is well organized. Figure 12 shows *µ*C/OS-II the file structure and includes the hardware/software architecture as well. The kernel code is organized into three segments:

Application Specific Code contains the user specific application software as well some code related to the *µ*C/OS-II. This includes initializing and starting the kernel as well as using the kernel specific API for task management, synchronization and communication.

Processor-Independent Code is the main code of the *µ*C/OS-II kernel and is independent of the actual target processor. It provides the kernel services for task management, time management, semaphores, scheduling policy and memory management.

Processor-Specific Code contains an adaptation layer: the port to the selected target processor, which varies with processors. This code typically manipulates directly individual processor registers, for example in order to switch contexts.

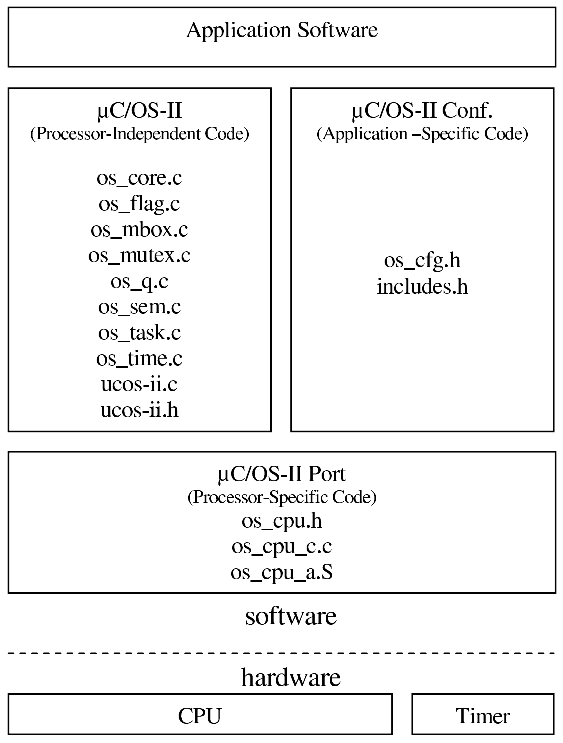


Figure 12: MicroC/OS-II hardware/software architecture (Source [29]).

3.2.2 Kernel and Kernel Services

The kernel, the heart of the operating system, provides multitasking services so that the application can be divided into smaller manageable tasks that share the same processor. Based on the scheduling policy, the kernel decides which task to run and switches between tasks (context switch). It saves the context (i.e. the CPU registers) of the current task onto the it's stack, loads the context of the new task and continues executing the new task.

*μ*C/OS-II kernel provides a number of system services, for a detailed description including their implementation please refer to 12.

Task Management. *μ*C/OS-II supports a multitasking environment with up to 56 application specific tasks. In order to manage these tasks, *μ*C/OS-II kernel provides services for creation, deletion, to change a task's priority, to suspend and resume a task and to get more runtime information about a task.

Time Management. By use of a system timer interrupt, application specific between every 10ms to 100ms, *μ*C/OS-II keeps track of the real-time by incrementing a 32-bit tick counter. It allows the user to set and query the time, as well as to suspend a task for a user specified time. Internally *μ*C/OS-II runs the scheduler for each timer tick.

Semaphore Management. *μ*C/OS-II promotes inter task communication through shared data structures. It simplifies the exchange of large amounts of data, but requires synchronization. Exclusive access to the data is needed avoid corruption of data. *μ*C/OS-II contains a semaphore implementation and provides an API for the essential operations: creation, deletion, obtaining, querying and returning of a semaphore.

Mutual Exclusion Semaphore Management. In addition to generic semaphores *μ*C/OS-II provides a specialized mutual exclusion semaphore (mutex). This is a binary semaphore that allows to gain exclusive access to the resources and provides additional features that reduce the priority inversion problem.

Memory Management *μ*C/OS-II provides support for dynamic memory allocation. It uses a fixed block size allocation scheme to avoid fragmentation, as available memory is typically small in embedded applications. An application can allocate and deallocate these memory blocks.

The kernel also manages interrupts, it disables the interrupts while entering the critical sections of the code, e.g. the manipulation of kernel internal data structures, and re-enables the interrupt when leaving the critical section. This prevents that multiple tasks enter the critical section simultaneously and corrupt data. An interrupt can suspend or resume execution of a task. In case the resumed task is the highest priority ready task, then it will execute as soon as the interrupt handler has finished.

As mentioned for the **Time Management** services, *μ*C/OS-II requires a periodic timer interrupt to keep track of time delays and timeouts. This periodic interrupt is referred to as the clock tick. It should be provided with a frequency of 10 to 100 times a second. A higher frequency allows more fine grained timing decisions, however it results in a higher overhead.

3.2.3 Adapting MicroC/OS-II for an ARM core

*μ*C/OS-II has already been ported to large number of processors. Several ports are available at the *μ*C/OS-II web site [32]. We based our processor adaptation on an ARM port [33] and adjusted it according to the SWARM ISS. Figure 13 depicts the RTOS as it runs within t he ISS in our co-simulation environment.

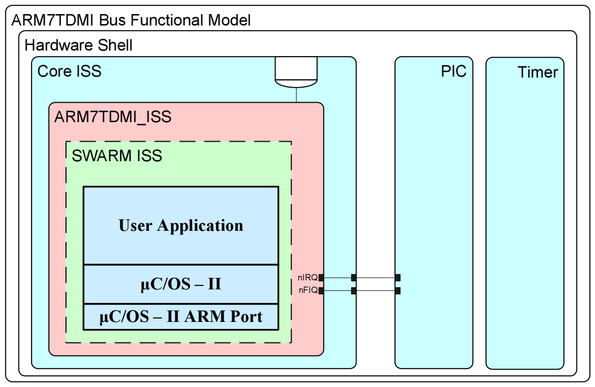


Figure 13: Setup for *μ*C/OS-II.

The existing port for the ARM core mainly needs adjustments for the gcc cross compiler and the working environment. As part of adaptation to the utilized compiler, we adjusted the sizes of the data types and the resulting stack layout. The corrections for the working environment include the address map, the communication with the PIC and the timer.

The μC/OS-II port contains the processor-specific code of the operating system. The processor-independent code calls the specific code as C functions. The processor-specific code is implemented in C and/or in assembly code to perform register operations.

This section discusses some key functions of the processor-specific code. A more detailed description can be found in the μC/OS-II book [29].

To protect the kernel internal data structures, μC/OS-II uses the concept of a critical section. During execution of a critical section the interrupts have to be disabled. This avoids preemption by an interrupt service routine, thus avoids unwanted scheduling events and makes execution of the critical section atomic. For this purpose, the port defines two functions that mark the start *OS_ENTER_CRITICAL()*) and the end *OS_EXIT_CRITICAL()* of a critical section. These functions disable and re-enable the interrupts respectively. They make use of two assembly level functions *OS_CPU_SR_Save()* and *OS_CPU_SR_Restore()* that modify the CPSR register. *OS_CPU_SR_Save()* disables all interrupts and returns the list of the previously enabled interrupts. *OS_CPU_SR_Restore()* restores the previous interrupt state by enabling all interrupts in the list.

In order to perform a task level context switch, the port includes the *OSCtxSw()* function. The scheduler performs a context context switch to change the task running on the CPU whenever a higher priority task becomes ready or, when the current task transitions to the waiting state. Figure 14 shows the operations performed during context switch. For that the OS saves the current program counter as a return address on the currents task stack. It stores the values of all registers onto the stack and updates the stack pointer in the TCB of the current task as referenced by *OSTCBCur*. Then it loads the context of the new task, the high ready task with the TCB pointer *OSTCBHighRdy*, by performing the same steps in reverse order. It ends with a return instruction that reads the program counter of the new task from its stack and execution of the new task resumes.

The OS port contains more functions, e.g. *OSStartHighRdy()* for starting the highest priority task, *OSTaskStkInit()* to initialize the task stack and there fore the registers at startup, and *OSTaskCreateHook()* that is called when a task is created. Please refer to ARM port documentation [33] for more detailed information on the functions in the ARM port.

Next, we describe the integration to the simulation environment, namely the interaction with the PIC and the timer.

3.2.4 Interrupt Handling and Timer Integration

The port provides interrupts handling. In order to reduce the amount of code that needs to be adapted for integrating μC/OS-II to a particular setup, the interrupt handling is split into two portions. The initial step prepares for executing an interrupt handler and is independent of the PIC, while the second step communicates with the PIC and executes the user interrupt handler.

In the first step, *OS_CPU_IRQ_ISR()* is called when an IRQ arrives. Its implementation is independent of the address map and can be used irrespective of whether an interrupt controller is present or not.

OS_CPU_IRQ_ISR(), implemented in assembly, only prepares for the actual interrupt handling. It performs register operations to save the current processor state, switches to the interrupt execution mode and calls *OS_CPU_IRQ_ISR_Handler()* for further processing. After this function finished, it restores the saved processor state, returns to the user execution mode and returns to the preempted user code. Note that a different task may be switched in as a result of a scheduling decision during the interrupt execution. Therefore, the implementation has to be consistent with the context switch implementation.

OS_CPU_IRQ_ISR_Handler() is written in C is dependent on the address map and the PIC. We adapted the *OS_CPU_IRQ_ISR_Handler()* according to the PIC present in the instruction set model. The PIC multiplexes from many external interrupts to a few

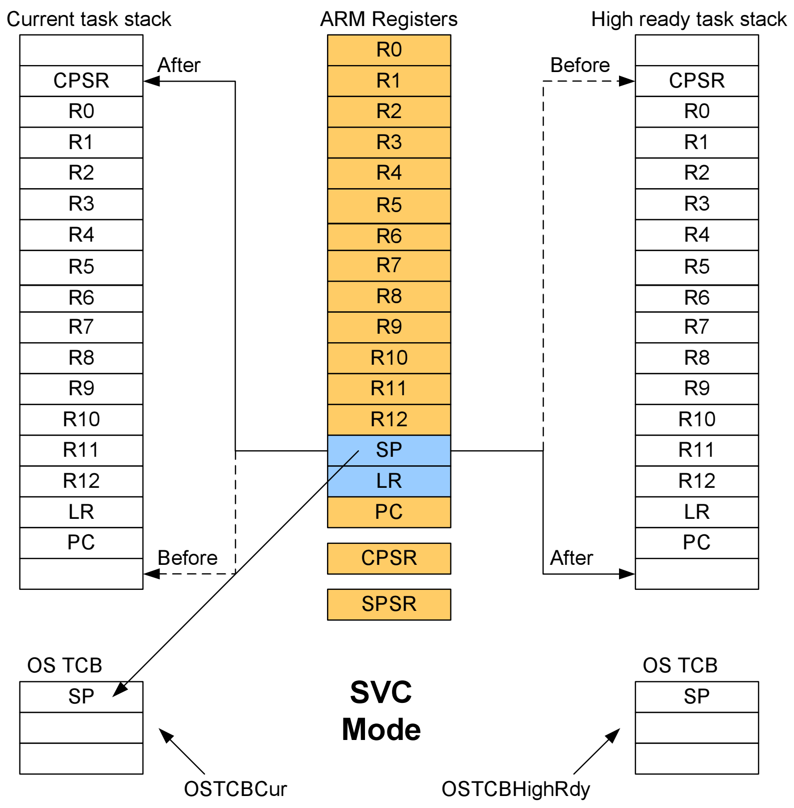


Figure 14: Task level context switch (source [33]).

internal interrupts. In order to determine which external interrupt has caused execution of the IRQ, *OS_CPU_IRQ_ISR_Handler()* reads the *Interrupt Status* register of the PIC and calls a the registered interrupt handler containing the actual user interrupt handler code. We defined and implemented a generic function *UserIrqRegister()* for registering a user function as an interrupt handler for particular interrupt.

In order to keep track of time and for scheduling purposes, μC/OS-II requires a periodic timer interrupt. This timer interrupt is generated by an external timer implemented in the bus functional model. The timer was implemented, like the PIC, according to NEC's System-on-Chip Lite+ specification [25].

The registers of the timer are available via the AHB through an AHB slave interface. The timer interrupt is connected to an interrupt line of the PIC. During the software startup, the function *OSTimeTick* is registered as an interrupt handler associated with the timer interrupt, the timer is enabled and programmed to generate a periodic interrupt every 10ms. Therefore, the OS scheduling and time management functions are executed on a periodic basis.

4 Embedded Software Generation

Figure 15 shows an overview of the design flow. The system level design process is composed of a gradual refinement, starting with the most abstract specification model. Then the designer adds implementation specific information and explores different alternatives. With each refinement step more implementation detail gets added to the system model. One refinement step is the Architecture Refinement, which introduces processing elements and maps behaviors to them. Another refinement step is the scheduling refinement, which schedules execution on each processing element. The final refinement step, with respect to software, is the the embedded software generation.

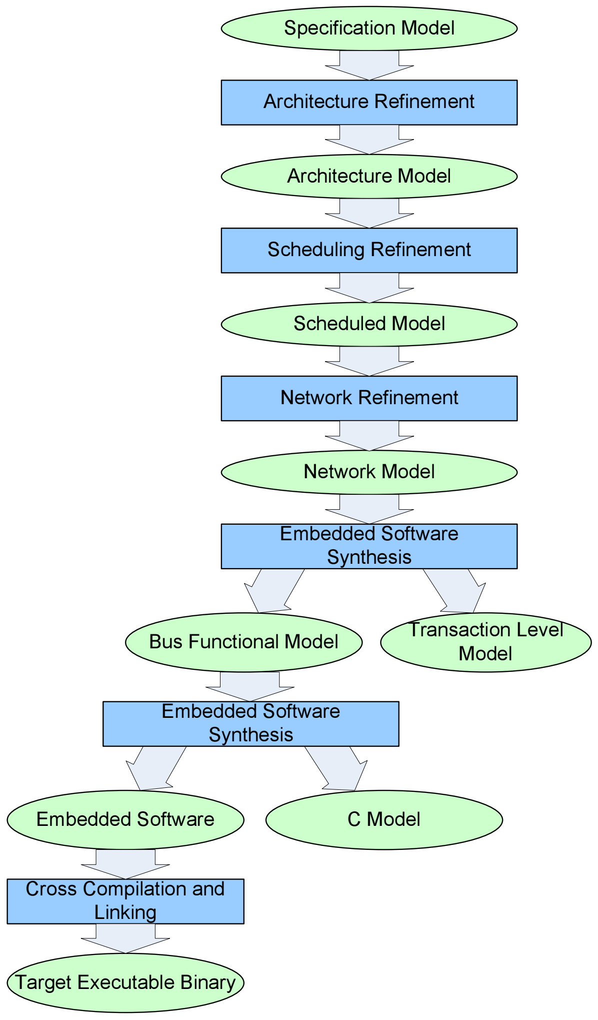


Figure 15: System design flow [21].

4.1 Scheduling Refinement

The software synthesis is divided into two refinement steps. The first step is the Scheduling Refinement. Here the designer specifies the scheduling parameters for processing elements to which multiple behaviors have been mapped. Options are static and dynamic scheduling. Within the dynamic scheduling either First Come First Serve (FCFS) or priority based scheduling are selectable. In case of priority scheduling, the designer has to define the priority of each task.

As a result of the scheduling refinement, behaviors are mapped to a processing element with a notion of tasks. For dynamic priority based scheduling a priority is defined for each task. Tasks are scheduled based on an abstract RTOS that is implemented as a separate channel instantiated in the processing element (see [24, 23]). The output model, the scheduled model, is still completely implemented in the SLDL.

4.2 Embedded Software Generation and RTOS Targeting

The second refinement step is the Embedded Software Synthesis, which generates the target C code and performs the RTOS targeting. The software generation approach is based on the work outlined in [38, 37]. The software synthesis tool generates a set of software tasks from a partitioned design specification, the bus functional model, captured in an SLDL, which in turn is the result of the scheduling refinement. The generated software tasks implemented in target C code are then scheduled by an real time kernel.

During the generation, the SLDL code of a behavior, which is mapped to a software task, is converted to C code. All communication primitives are replaced with plain C code as well. Task and synchronization primitives are replaced with calls to a generic RTOS wrapper, a thin RTOS abstraction layer. The C synthesis tool also generates interrupt handlers in case interrupts are used for synchronization with external components. It generates start-up code that registers these interrupt handlers to the operating system.

The output of the embedded software generation is twofold. For one, the generated C code is reintegrated into the input model (the bus functional model). Here it replaces the original SLDL behaviors that have been mapped to software tasks. It integrates into the remaining system design with an abstract RTOS that has been implemented on top of the SLDL [38]. The C Model can be used for a fast validation of the generated C code. It co-simulates with the remaining system design, usually containing custom hardware. Although the functional correctness can be validated, it does not yield accurate timing results.

Secondly, a flat C code, the Embedded Software, is generated for each programmable processing element. This code is designed for execution on the target processor. The refinement step of Cross Compilation and Linking uses a target specific cross compiler to compile the generated C code and link it against target and RTOS specific libraries. It produces a binary for execution on the target processor.

We extended the refinement database to include software components. For the integration of the ARM7TDMI we populated the software database with the RTOS μC/OS-II, a μC/OS-II wrapper that provides RTOS abstraction API, the RTOS porting code that contains the processor-specific code of the μC/OS-II, a software HAL that implements the MAC layer communication routines, which communicate with the PIC and the timer. Additionally we included an ISS specific libc that provides standard output routines for debugging purposes.

We devised a generic file structure for the software database to maximize reuse between SW components. As an example, we ensured that the communication with the PIC, implemented in C, is independent from the selected RTOS, the processor and the cross compiler. Therefore, the same PIC communication code can be used on all processors that are able to connect to the particular PIC, i.e. that can connect to the AHB in this case.

5 Experiments

In order to show the feasibility of the design flow, with respect to the embedded software generation, we implemented a real life example and executed each step of the refinement flow.

5.1 Example Application

Our example stems from the automotive industry. We implemented an anti lock break system as shown in Figure 16. It contains a single processor, an ARMv7, that runs the control application. The processor communicates through an Advanced Microprocessor Bus Architecture (AMBA) AHB with a Controller Area Network (CAN) controller. The CAN controller, a transducer, is accessible from the CPU through memory mapped I/O. Five devices are connected on the simulated CAN bus. One sensor that measures the break paddle position, a sensor for each wheel that senses the wheel's rotation and a actuator for each wheel that controls the asserted break pressure.

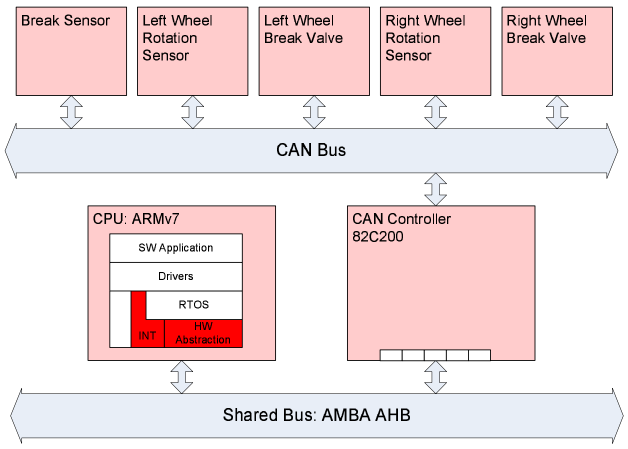


Figure 16: Anti-lock break example architecture.

5.2 Refinement

We first captured the specification model of the example system. Then, we used the existing refinement flow to perform the refinement steps until the generation of the bus functional model. Throughout the process, we used refinement decisions that match the desired target architecture (Figure 16).

The refinement tools use the database entries for the ARM7TDMI processor for synthesizing the system models. The Architecture Model makes use of the abstract processor model (see Section 2.2.1). The Scheduling Refinement includes the OS model as described in Section 2.2.2. The bus functional communication model contains the BFM of the ARM7TDMI (see Section 2.2.3) as well as the model of the bus system AHB [35].

We then used the extended software synthesis tool sc2c (see Section 4) to generate the C code targeted towards the RTOS. We cross compiled the generated C code with a gcc cross compiler [26] that produces binaries in the COFF format and linked against the necessary database components (RTOS, RTOS wrapper, RTOS port, HAL and libc). The result is a binary in the COFF format that can be executed using the SWARM ISS.

We manually exchanged the bus functional model of the ARM7TDMI in the systems BFM with the instruction set model as described in Section 2.3.3. As a result we achieved a bus-functional system model that co-simulates hard and software and provides cycle accurate execution of the software, communication and hardware[3].

5.3 Results

We validated correct functional execution of all created models. All models exhibit correct functionality. An example of an emergency stop maneuver with an initial speed of 20 $\frac{meters}{second}$ (45mph, 72$\frac{km}{h}$) is shown in Figure 17.

The left graph, Figure 17(a), shows the correlation between the break request (*Break In*) as it is read from the break paddle and the adjusted break pressure *Break Out* asserted by the front left break actuator. The output *Break Out* is set depending on the rotation speeds of the wheels. The right graph, Figure 17(b), shows the speeds of both front wheels. It is noticeable that the left wheel locks, starting at normalized break pressure of 50 units. Therefore the anti lock break algorithm reduces the break pressure until the wheel rotates freely again. After which in increases the break pressure again until the next locking of the wheel occurs. The cycle repeats until the car comes to a full stop.

Next, we measured the execution time and the lines of generated code as an indicator of complexity. The results are summarized in Table 1.

Model	Lines of Code	Simulation Time
Spec	238	0.018sec
Architecture	10670	0.009sec
Schedule	11760	0.020sec
Network	14474	0.014sec
TLM	22035	0.153sec
BFM	22048	125min
BFM + C	23330	123min
BFM(ISS) + C	22390 + 1416	208min

Table 1: Model complexity in lines of code and simulation time.

The results in Table 1 show that up to the transaction level model the execution time is negligible, significantly less than a single second of execution time. Starting with the BFM, the execution time dramatically increases to over two hours. This dramatic increase is due to the application. Each 20ms of simulated time, the status of all sensors is queried and the output is calculated. Therefore only a minimum computation is performed. The simulation is mostly busy simulating the idle bus systems. Both, the AHB, running at 25 MHz, and the 1 MHz CAN use explicit clocks. The CAN especially contributes to the slowdown since each of the six simulated CAN nodes operates on an own local clock and the CAN standard requires oversampling the bus[4] for synchronization with clock of the sending CAN node.

The bus-functional model with the re-integrated C code executes as fast as the bus-functional model due to the low amount of computation. Finally the ISS based bus-functional model is 66% slower than the bus-functional model. The SWARM ISS is executed as well with a simulated frequency of 25 MHz and therefore adds an significant overhead.

In summary, all models are functionally correct. The execution time analysis shows, that the simulation effort dramatically increases with an increased accuracy. Although it has to be noted that the particular example was limited by simulating explicit clocks during the idle times of the system.

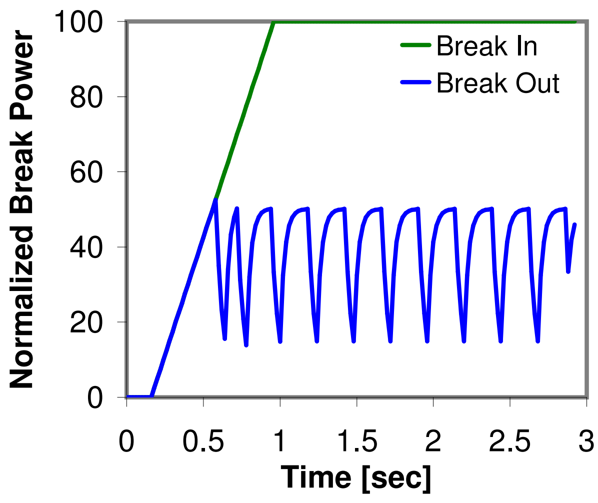
6 Conclusions

In this document we presented an extension of a system level refinement flow to include hardware software co-development and co-simulation. In form of a case study, based on the widely used processor ARM7TDMI, we described three major tasks necessary for a software support throughout the design flow.

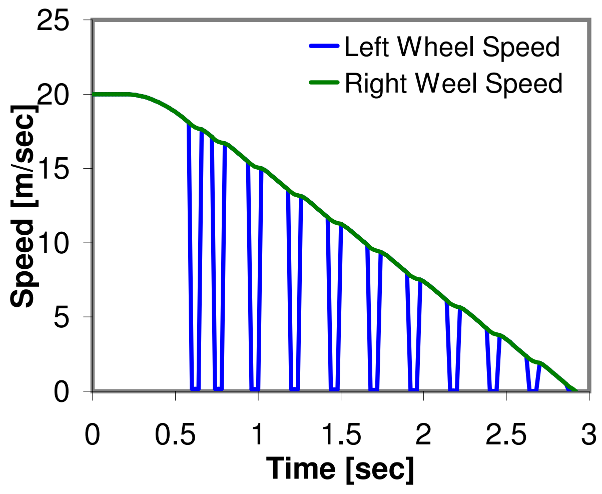
First, we described modeling of the processor at different levels of abstraction. Capturing a processor for the refinement flow allows to map behaviors to programmable processing elements (processors), to co-simulate the model and to estimate the system performance. We described different levels of abstraction, starting from an abstract, weight table based approach ranging to a cycle accurate execution of the

[3]Assuming RTL synthesis has been performed as well.

[4]Each bit on the bus is oversampled (e.g. 12 times, [17])



(a) Break Power



(b) Wheel Speed

Figure 17: Anti-lock break simulation.

target binaries on an Instruction Set Simulator (ISS). We successfully integrated SWARM, an ARM7 ISS, into our refinement and co-simulation flow.

Second, we analyzed the needs for an Real-Time Operating System (RTOS). We described our selection criteria for the chosen RTOS μC/OS-II and covered insight about the performed adaptation to the selected processor.

Thirdly, we reported on the embedded software synthesis and described an extension that now includes the synthesis of target C code. For the effective use of the generated target code, we covered the extension of the refinement flow's database to contain software components, such as the RTOS and the Hardware Abstraction Layer (HAL) defining the bus driver.

Using an automotive example system of anti-lock breaks, we have validated the extension of the refinement flow. We performed an step-by-step refinement through all stages of the design and utilized the three introduced elements: the processor models, the RTOS and the embedded software synthesis.

The experimental results show the functional correctness of all models, including the cycle-accurate simulation based on the SWARM ISS. With that we have for the first time reached with automatic refinement the final stage of embedded software synthesis and cycle-accurate co-simulation.

However, our experimental results also show that the execution speed dramatically reduces with an increased detail level. Whereas the abstract models execute in less than a second, the ISS based co-simulation requires more than three hours. Analyzing the results, yielded that the particular example is limited by simulating the busses during the idle time of the system and that the actual computation contributes only minimally.

In summary, we have shown the feasibility for a hard/software co-design and for an automatic refinement flow. We have demonstrated hardware software co-simulation capabilities a each stage of the design.

In future work, we will show the cycle accurate execution timing. We will implement more software components. Furthermore, we will investigate into improving the simulation speed of the co-simulation environment.

References

[1] Samar Abdi, Junyu Peng, Haobo Yu, Dongwan Shin, Andreas Gerstlauer, Rainer Dömer, and Daniel Gajski. System-on-chip environment (SCE version 2.2.0 beta): Tutorial. Technical Report CECS-TR-03-41, Center for Embedded Computer Systems, University of California, Irvine, July 2003.

[2] Advanced RISC Machines Ltd. (ARM). ARM7TDMI (Rev 3) Product Overview. www.arm.com/pdfs/DVI0027B_7_R3.pdf.

[3] Advanced RISC Machines Ltd (ARM). AMBA Specification (Rev. 2.0), ARM IHI 0011A. www.arm.com/products/solutions/AMBA_Spec.html.

[4] Advanced RISC Machines Ltd. (ARM). PROCESSOR CORE OVERVIEW. www.arm.com/products/CPUs/index.html.

[5] Advanced RISC Machines Ltd. (ARM). RealView Developer Suite Instruction Set Simulator. www.arm.com/products/DevTools/RealViewISS.html.

[6] Advanced RISC Machines Ltd. (ARM). SoC Developer with MaxSim Technology. http://www.arm.com/products/DevTools/MaxSim.html.

[7] Advanced RISC Machines Ltd. (ARM). ARM7TDMI (Rev 4) Technical Reference Manual, 2001. www.arm.com/pdfs/DDI0210B7TDMIR4.pdf.

[8] Felice Balarin, Massimiliano Chiodo, Paolo Giusto, Harry Hsieh, Attila Jurecska, Luciano Lavagno, Claudio Passerone, Alberto Sangiovanni-Vincentelli, Ellen Sentovich, Kei Suzuki, and Bassam Tabbara. *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach*. Kluwer Academic Publishers, 1997.

[9] Luca Benini, Davide Bertozzi, Alessandro Bogliolo, Francesco Menichelli, and Mauro Olivier. "mparm: Exploring the multi-processor soc design space with systemc". *VLSI Signal Processing*, 41:169–182, 2005.

[10] Doug Burger and Todd M. Austin. *The SimpleScalar Tool Set, Version 2.0*. Computer Sciences Department, University of Wisconsin-Madison, June 1997.

[11] Jrôme Chevalier, Maxime de Nanclas, Luc Filion, Olivier Benny, Mathieu Rondonneau, Guy Bois, and El Mostapha Aboulhamid. A SystemC Refinement Methodology for Embedded Software. *IEEE Design and Test of Computers*, 99(99):148–158, 2006.

[12] Intel Corporation. Intel StrongARM SA-1110 Microporcessor Developer's Manual. developer.intel.com/design/strong/manuals/278240.htm, October 2001.

[13] CoWare. Virtual Platform Designer. www.coware.com.

[14] Michael Dales. *SWARM 0.44 Documentation*. Department of Computer Science, University of Glasgow, November 2000. www.cl.cam.ac.uk/~mwd24/phd/swarm.html.

[15] GDB Developers. GDB User Manual. www.gnu.org/software/gdb/documentation/.

[16] eCOS Community. eCOS home page. ecos.sourceware.org.

[17] Armin Bassemir Florian Hartwich. The configuration of the can bit timing. www.can.bosch.com/, 1999.

[18] A. Gerstlauer; D. Shin; R. Doemer; D. Gajski. System-Level Communication Modeling for Network-on-Chip Synthesis. In *Asia and South Pacific Design Automation Conference*, Shanghai, China, January 2005.

[19] Daniel D. Gajski, Frank Vahid, Sanjiv Narayan, and Ji Gong. *Specification and Design of Embedded Systems*. Prentice Hall, 1994.

[20] Daniel D. Gajski, Jianwen Zhu, Rainer Dömer, Andreas Gerstlauer, and Shuqing Zhao. *SpecC: Specification Language and Design Methodology*. Kluwer Academic Publishers, 2000.

[21] Andreas Gerstlauer, Rainer Dömer, Junyu Peng, and Daniel D. Gajski. *System Design: A Practical Guide with SpecC*. Kluwer Academic Publishers, 2001.

[22] Andreas Gerstlauer, Gunar Schirner, Dongwan Shin, Junyu Peng, Rainer Dömer, and Daniel D. Gajski. System-on-chip component models. Technical Report CECS-TR-06-10, Center for Embedded Computer Systems, University of California, Irvine, May 2006.

[23] Andreas Gerstlauer, Haobo Yu, and Daniel D. Gajski. RTOS modeling for system level design. In Ahmed A. Jerraya, Sungjoo Yoo, Norbert Wehn, and Diedrik Verkest, editors, *Embedded Software for SoC*. Kluwer Academic Publishers, 2003.

[24] Andreas Gerstlauer, Haobo Yu, and Daniel D. Gajski. RTOS Modeling for System Level Design. In *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, Munich, Germany, March 2003.

[25] NEC Electronics (Europe) GmbH. System-on-Chip Lite +. User's Manual. `www.eu.necel.com/_pdf/A17158EE2V0UM00.PDF`, April 2005.

[26] GNU. gcc (gcc-arm-coff version 2.95.3). `ftp://ftp.gnu.org/gnu/gcc`.

[27] Thorsten Grötker, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[28] F. Herrera, H. Posadas, P. Snchez, and E. Villar. Systematic Embedded Software Generation from SystemC. In *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, 2003. `http://csdl.computer.org/comp/proceedings/date/2003/1870/01/18701% 0142abs.htm`.

[29] Jean J. Labrosse. *MicroC/OS-II: The Real-Time Kernel*. CMP Books, 2002.

[30] RTEMS Steering Committee Members. Real-Time Operating System for Multiprocessor Systems home page. `www.rtems.com`.

[31] TinyOS Committee Members. TinyOS home page. `www.tinyos.net`.

[32] Micriμm. μC/OS-II Home Page. `www.ucos-ii.com/arm/index.html#rtosports`.

[33] Micriμm. *μC/OS-II and The ARM Processor, Application Note*, 2004.

[34] Gautam Sachdeva. Integration of an arm core in a system design flow. Master's thesis, Electrical Engineering and Computer Science, University of California, Irvine, March 2006.

[35] Gunar Schirner and Rainer Dömer. Quantitative Analysis of Transaction Level Models for the AMBA Bus. In *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, Munich, Germany, March 2006.

[36] VaST Systems. VaST tools and models for embedded system design. `www.vastsystems.com`.

[37] Haobo Yu. *Software Synthesis for System-on-Chip*. PhD thesis, Electrical Engineering and Computer Science, University of California, Irvine, June 2005.

[38] Haobo Yu, Rainer Dömer, and Daniel Gajski. Embedded software generation from system level design languages. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, Yokohama, Japan, January 2004.