Brigham Young University
BYU ScholarsArchive

Theses and Dissertations

2007-07-10

Compilation and Generation of Multi-Processor on a Chip Real-Time Embedded Systems

Randall S. Klingler
Brigham Young University - Provo

Follow this and additional works at: https://scholarsarchive.byu.edu/etd

Part of the Electrical and Computer Engineering Commons

BYU ScholarsArchive Citation

Klingler, Randall S., "Compilation and Generation of Multi-Processor on a Chip Real-Time Embedded Systems" (2007). *Theses and Dissertations*. 959.
https://scholarsarchive.byu.edu/etd/959

This Thesis is brought to you for free and open access by BYU ScholarsArchive. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of BYU ScholarsArchive. For more information, please contact scholarsarchive@byu.edu, ellen_amatangelo@byu.edu.

COMPILATION AND GENERATION OF MULTI-PROCESSOR ON A CHIP REAL-TIME EMBEDDED SYSTEMS

by

Randall S. Klingler

A thesis submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Electrical and Computer Engineering

Brigham Young University

August 2007

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

Randall S. Klingler

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date	Doran K. Wilde, Chair
Date	James K. Archibald
Date	Brent E. Nelson

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of Randall S. Klingler in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

______________________________ ______________________________

Date

Doran K. Wilde
Chair, Graduate Committee

Accepted for the Department

____

Michael J. Wirthlin
Graduate Coordinator

Accepted for the College

____

Alan R. Parkinson
Dean, Ira A. Fulton College of Engineering and Technology

ABSTRACT

COMPILATION AND GENERATION OF MULTI-PROCESSOR ON A CHIP REAL-TIME EMBEDDED SYSTEMS

Randall S. Klingler

Department of Electrical and Computer Engineering

Master of Science

Current FPGA technology has advanced to the point that useful embedded System-on-Programmable-Chips (SoPC)s can now be designed. The Real Time Processor (RTP) project leverages the advances in FPGA technology with a system architecture that is customizable to specific real-time applications. The design and implementation of the framework for architecting such a system from ANSI-C code is presented. The Small Device C Compiler (SDCC) was retargeted to the RTP architecture and extended to produce a generator directive file. The RTPGen hardware generator was created to consume the directive file and produce a highly customized top-level structural VHDL file that can be synthesized and programmed onto an FPGA such as the Xilinx Spartan-3. Thus, an application specific multiprocessor real-time embedded system is realized from ANSI-C code.

ACKNOWLEDGMENTS

First and foremost, I want to thank my wife Lori for all of her loving support through graduate school. I thank Doran Wilde for many hours of help in research and the writing of this thesis. I also thank Spencer Isaacson and Matt Young for many café meetings to discuss the RTP project.

TABLE OF CONTENTS

LIST OF TABLES xi

LIST OF FIGURES xiii

Chapter 1: Introduction 1

1.1 – RTP Motivation 2

1.2 – RTP Goals 2

1.3 – Related Work 3

1.4 – Goals for the SDCC-RTP Compiler and the RTPGen Hardware Generator 4

1.5 – Contributions 5

1.6 – Outline of Thesis 6

Chapter 2: Real-Time System on a Programmable Chip 7

2.1 – RTP Architecture Overview 7

2.1.1 – RTP System Components 8

2.1.2 – The Task-Resource Matrix 10

2.1.3 – The Scheduler 11

2.1.4 – Context Switching 11

2.1.5 – The RTP Hardware Assisted RTOS 12

2.2 – The RTP Processor and Instruction Set Architecture 13

2.3 – Related Work 16

2.4 – Additional Considerations 17

Chapter 3: Porting the SDCC Compiler **19**

3.1 – Decision to use SDCC 20

3.2 – Base SDCC Complier Functionality 21

3.3 – Required Changes to Base SDCC Compiler 23

3.4 – The SDCC-RTP Compiler: Code Generation 27

3.4.1 – Port Options (main.*) 27

3.4.2 – Register Allocation (ralloc.*) 28

3.4.3 – Code Generation (gen.*) 31

3.4.4 – Peephole Rules (peeph.def) 38

Chapter 4: The SDCC-RTP Compiler **41**

4.1 – Built-in Functions 41

4.1.1 – Compiler Support for Built-in Functions 46

4.2 – Intermediate Symbol Table 47

4.3 – Generator Directives 48

Chapter 5: The Assembler and Linker **51**

5.1 – The Assembler 51

5.2 – The Linker 53

Chapter 6: The RTPGen Hardware Generator **57**

6.1 – Instantiation of Processors 59

6.2 – Creation of Tasks, Resources, and Devices 60

6.3 – RTP Simptris Example 60

Chapter 7: Conclusions and Future Work **73**

7.1 – Contributions Revisited 73

7.2 – Future Work 74

7.3 – Final Words 75

BIBLIOGRAPHY 77

Appendix A: Peephole Rules for the SDCC-RTP Compiler 79

Appendix B: Assembly for RTP Code Generation Example 83

Appendix C: RTP RTOS Assembly Source Code 87

Appendix D: RTP Instruction Set Architecture 97

LIST OF TABLES

Table 3.1: Register Assignments 28

Table 3.2: RTP Recognized iCode Operation Codes.. 32

Table 5.1: Assembler Relocatable File Format.. 52

Table 5.2: Assembler Relocatable Patch Instructions... 53

Table 5.3: RTP Segments 54

LIST OF FIGURES

Figure 2.1: RTP System Architecture General Overview 8

Figure 3.1: C source code for RTP code generation example 34

Figure 3.2: iCode chain for RTP code generation example 35

Figure 3.3: Assembly code for RTP code generation example 37

Figure 4.1: The global.h Header File Template 42

Figure 4.2: Built-in Function Structure Type Definition 46

Figure 4.3: Intermediate Symbol Table for Generator Directives 48

Figure 6.1: Dataflow for C-to-FPGA Compilation 58

Figure 6.2: RTP System Diagram for Simptris Example 61

Figure 6.3: RTP Simptris global.h 63

Figure 6.4: RTP Simptris proc0.c 65

Figure 6.5: RTP Simptris proc1.c 67

Figure 6.6: RTP Simptris RTPsys.gen 68

Figure 6.7: RTP Simptris VHDL Declarations for Processor 0 69

Figure 6.8: RTP Simptris VHDL Instantiation of Processor 0 70

Figure 6.9: RTP Simptris VHDL Instantiation of P0 Task 1 and Scheduler 71

Chapter 1: Introduction

Current trends in system design show migration toward tighter integration. It is now common for processors, memory, and custom hardware to all be contained on a single System-on-Chip (SoC) device. Several recent innovations in hardware/software co-design target these SoCs in an effort to improve embedded system performance and design [1, 2, 3]. Even more recently, advances in Field Programmable Gate Arrays (FPGAs), such as the Xilinx Spartan-3 and Virtex II [4] have made it possible to design powerful and easily customizable System-on-Programmable-Chip (SoPC) devices. These SoPCs, with their low non-recurring engineering costs and extensive customizability represent an increasing portion of embedded system designs. An FPGA has abundant hardware elements but only limited on chip memory space, causing code software to be relatively more expensive. For an FPGA, specialized hardware implementations have the triple benefit of being faster, cheaper, and more predictable than equivalent software implementations.

With abundant hardware resources, an FPGA is a perfect target for a hardware assisted real-time embedded system. The Real Time Processor (RTP) project combines the use of customized hardware and a small software real-time operating system (RTOS) to take advantage of the strengths inherent in an FPGA. The RTP project has been given application support through the Small Device C Compiler (SDCC) targeted to the RTP

architecture, hereafter referred to as the SDCC-RTP compiler, and the RTPGen hardware generator. These application supports, coupled with a VHDL component library comprise the entire C-to-FPGA system.

1.1 – RTP Motivation

As FPGAs increase in gate count and memory capacity, SoPCs will continue to increase in popularity. The RTP infrastructure was designed to take advantage of this continual improvement in technology, as discussed in the following sections. The RTP project was also created for the purpose of focusing on the strengths of FPGA technology in embedded designs. Chapter 2 describes the RTP system architecture in detail.

1.2 – RTP Goals

The RTP project was designed with the objective of accomplishing the following goals:

- To create the infrastructure to implement customized real-time systems
- To design a flexible and scalable system framework that targets state-of-the-art FPGA technology and that will grow with FPGA advances
- To provide multiprocessor support
- To utilize a standard C interface for system development
- To support resource sharing in a uniform and reliable manner
- To find the right balance between doing functions in hardware and software

1.3 – Related Work

The Streams-C C-to-FPGA compiler [5] synthesizes stream-oriented circuits for FPGA based computers from a modified version of the C language. The Streams-C compiler is also comprised of a small number of libraries and functions added to the C language. It is primarily targeted to stream-oriented computation on FPGA-based parallel computers. It synthesizes hardware circuits for a target FPGA board (currently the Annapolis Microsystems Wildforce board) containing multiple FPGAs, external memories, and interconnect. The Streams-C compiler allows the programmer to specify directives to the hardware generator, in the form of pragmas, and uses a pre-processor to convert predefined functions into pragmas. It is shown in [6] that the Streams-C VHDL can give a development speedup of 5 to 10 over hand-coded VHDL, at a penalty of area utilization (up to 4 times), and circuit speed (up to 50%).

Handel-C [7] is designed for C-to-FPGA compilation, and supports soft core processors, such as the Xilinx Microblaze, and the Altera NIOS. Handel-C includes a basic component library, and adds simple constructs to ANSI-C that support direct generation of hardware. Its primary function is the implementation of algorithms in hardware, and it is targeted primarily to software engineers. It enables concurrent hardware and software development within a modified C language environment. Some of the extensions to ANSI-C include: flexible data widths, parallel processing, and communication between parallel elements.

Neither Streams-C nor Handel-C are targeted for real-time embedded applications, whereas the RTP design is intended primarily for this purpose. It provides hardware to support an RTOS, and is designed to work in a multiprocessor architecture.

There has also been a fair amount of work done on minimizing code size. Prior work in this area, in the order in which they are performed, includes:

- Compiler optimizations (many of which are discussed in Chapter 3)
- Peephole rules [12] are used to statically analyze the code generated by the compiler and replace sub-optimal code patterns with optimal ones
- Code compression which requires additional hardware for decompression at run time [13]

The RTP design leverages the SDCC compiler optimizations coupled with a set of peephole rules for code footprint minimization.

1.4 – Goals for the SDCC-RTP Compiler and the RTPGen Hardware Generator

The first goal for the SDCC-RTP compiler is to modify the base compiler to allow for proper function call handling of the RTP specific functions that create tasks, resources, and devices. This modification allows the compiler to interface conveniently with the generator by means of a generator directive file. The second goal for the SDCC-RTP Compiler is to minimize the code size required while maintaining code correctness. Since on chip memory is a scarce resource on an FPGA, a minimal code size will help to lessen the effects of memory limitations. A third goal, which is tightly integrated with the first goals is to have the code well documented for ease of maintenance and future updates as the RTP architecture continues to evolve.

The main goal for the RTPGen Hardware Generator is to produce a top level structural VHDL file that will synthesize a real-time, application specific system to a Xilinx Spartan-3. This is accomplished by analyzing the compiler-produced generator

directive file to determine the equivalency groups of tasks, resources, and devices specified by the application programmer in ANSI-C code. A minimal set of hardware components from the RTP VHDL component library are instantiated to create a system specifically customized to run the application code.

1.5 – Contributions

The new architecture requires new tools to exploit its technological advances. In this thesis I will present the following contributions:

1. I helped define the system-level architecture for the RTP project.
2. I wrote the code generator for the SDCC-RTP compiler, which allows the user to develop applications for the RTP system using ANSI-C code and a set of predefined RTP specific function calls.
3. I created a set of peephole optimizations for the RTP architecture that reduce code size while preserving correctness. I also created a separate peephole optimizing engine to further analyze and compact the code size.
4. I designed a methodology to extend the SDCC-RTP compiler to interface with the RTPGen hardware generator.
5. I wrote the code for the RTPGen hardware generator, which speeds development time and eliminates human error by automating system generation using a correct-by-construction methodology. It produces structural hardware VHDL files that are highly customized to the application code.
6. I produced a much needed document (Chapter 3) that outlines a step-by-step method for retargeting the SDCC compiler to new architectures which can be:

a. Published on the SDCC web site.
b. Used to help others wanting to retarget SDCC.
c. Used as a reference for courses on code generation.

1.6 – Outline of Thesis

This thesis will discuss the implementation of a real-time application specific C-to-FPGA system, written in C. This approach differs from the previously discussed C-to-FPGA implementations [5, 6, 7] in that the application C code is compiled to assembly, with the minimal set of hardware required to execute the code being instantiated. Chapter 2 describes the RTP system-level architecture. Chapter 3 steps through the process of retargeting (porting) the SDCC compiler to the RTP architecture. Chapter 4 discusses the generator support provided by the SDCC-RTP compiler. Chapter 5 discusses the assembler and linker. The implementation of the RTPGen hardware generator, along with an example embedded system is found in Chapter 6. Chapter 7 contains the conclusion and future research ideas.

Chapter 2: Real-Time System on a Programmable Chip

The Real Time Processor (RTP) system architecture has features that make it especially suited for hosting real-time applications. It is based on a flexible and scalable framework of multithreaded multiprocessors tightly coupled with on-chip resources and a hardware assisted Real-Time Operating System (RTOS). It uses a light-weight 16-bit RISC-like processor with instructions to support 32-bit arithmetic. It also uses an innovative task and resource management component called the Task-Resource Matrix (TRM). The RTP System is targeted to FPGAs, in particular the Xilinx Spartan 3-1500™.

2.1 – RTP Architecture Overview

A multiprocessor embedded system allows multiple high priority tasks to run concurrently, maximizing the efficiency of real-time applications, and allowing processor resources to be dedicated to servicing hard real-time deadlines. Tasks are able to communicate through shared resources for synchronization and message passing. These shared resources must be statically declared in application code for proper system generation. Tasks must also be statically declared, and thus cannot be dynamically created nor destroyed, nor can they dynamically migrate between processors.

2.1.1 – RTP System Components

A block diagram of the system architecture is shown in Figure 2.1. Figure 2.1 is only representative of one of many configurations possible for the system.

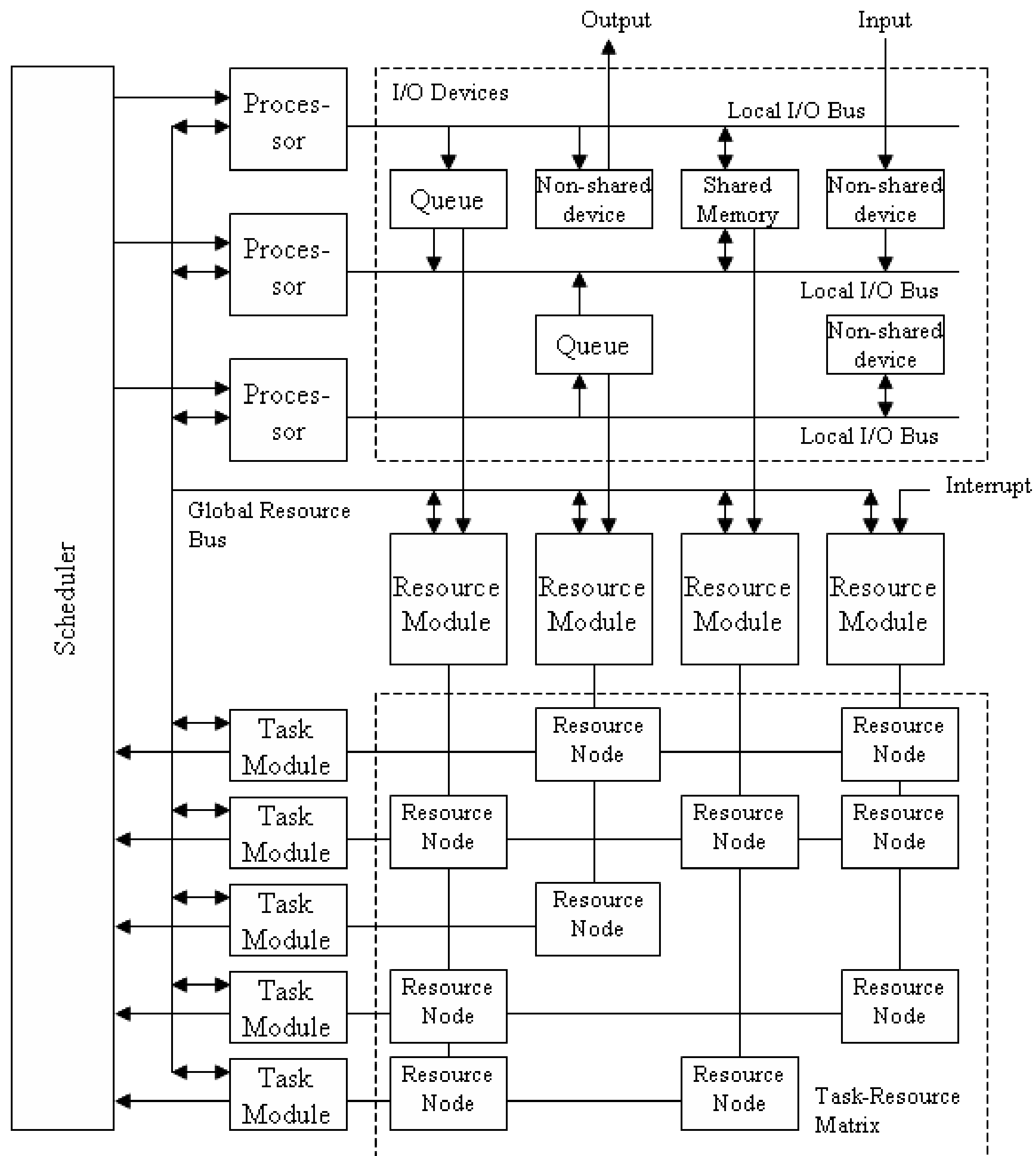

Figure 2.1: RTP System Architecture General Overview

Not shown in Figure 2.1 are the block RAMs for code and data memory for each processor.

The architecture is composed of the following elements, which are primitives available in the VHDL component library of the RTPGen hardware generator:

- An array of 1 to *n* processors, initially limited to 16 maximum. Each processor has its own local memory for code and data. All tasks on the processor share that memory.
- Each processor has its own local I/O space of up to 256 ports. A 16-bit I/O bus is connected to all local peripherals used by that processor, such as serial ports, queues, network interfaces, etc. All tasks that use the same I/O device must reside on the same processor.
- Peripherals that can be shared by two or more tasks, whether they reside on a single processor or on multiple processors, need to be synchronized among the competing tasks. This is done using system "resources" in this architecture. These include hardware implementations of semaphores, events, mutexes, timers, scratchpad memories, interrupt sources, and similar types of circuits that help manage shared resources of any type. Resources can be locked by a single task if mutual exclusion is required.
- The "task-resource matrix" is an innovative feature of the architecture that controls the sharing of resources in a unified way. It contains a "resource node" for each task that needs access to a resource. The resource node keeps track of pending requests and grants for a resource. The task-resource matrix provides information to the scheduler about what tasks are "blocked" waiting for a

resource. This new circuit provides important hardware assistance to the RTOS by keeping track of the tasks that are waiting for specific resources.

- A task module for each task contains information about the task, such as the task id, the processor id of where it is executing, its priority, and the value of a timeout counter for the task. This counter allows the task to set a time-limit that it will wait for a resource. Task priority can be dynamically changed, supporting operations such as priority inheritance to solve the priority inversion problem.
- A shared global resource bus is used to access system resources and task modules.
- The scheduler finds the highest priority ready-task for each processor each cycle and passes their task IDs to their respective processors.

Along with the aforementioned features of the RTP architecture, there are some restrictions. Processors do not share data or program memory, and the architecture is restricted to reside on a single FPGA.

2.1.2 – The Task-Resource Matrix

The Task Resource Matrix (TRM), as seen in Figure 2.1, defines the interaction between all tasks and resources in the system. The TRM does this by tracking the allocation of system resources to tasks, synchronization of tasks, and maintaining mutual exclusion for shared resources. It is physically organized by rows of tasks and columns of resources. Once all of the tasks and resources have been defined, a resource node is placed at the intersection of tasks and resources. The RTP RTOS uses the TRM to allow each resource to be locked for exclusive use of a task when needed.

Each resource node has interface pins that connect to a task, a resource, a processor, and the TRM. It is through these interfaces that the RTP system facilitates inter-process communication. The Task Resource Matrix and the pin interfaces of processors, tasks, resources, and resource nodes is discussed in much more detail in [11].

2.1.3 – The Scheduler

The real-time scheduler function has been moved entirely from software to hardware. In conjunction with the TRM, the scheduler determines which tasks are ready to run and then signals to each processor its highest priority ready-task. To implement priority scheduling, “ready” signals are produced by the TRM to signify which tasks are ready to run. The scheduler compares the priority of all ready tasks, giving higher priority tasks precedence. There is always at least one ready task per processor, the “idle task”, which is the lowest priority task and never blocks on any resource. In order for the scheduler to perform optimally, it is necessary to support task priority inheritance by temporarily raising the priority of a low-priority task holding a resource that is needed by a high-priority task. Priority inheritance is managed under control of the RTOS.

Additional details regarding the hardware implementation of the scheduler can be found in [11].

2.1.4 – Context Switching

Traditionally, context switching requires disabling interrupts, saving all registers and other state information for the active task, restoring another task’s state information and then re-enabling interrupts. This can require hundreds of instructions if done in

software. When interrupts are re-enabled, a higher priority interrupt may be pending, causing yet another context switch.

The RTP System was designed to accelerate an RTOS with hardware resources. One specific area that was targeted is the overhead associated with context switching. A traditional embedded system handles an incoming interrupt via two context switches: the interrupt service routine (ISR) must save the previous context before calling the interrupt handler, following which it must restore the context. The RTP method of handling context switches eliminates the need to save and restore context in an ISR, and nearly eliminates the time cost associated with a context switch such that ISRs can be implemented as high priority tasks, blocked on the interrupt signal. This is accomplished in hardware by giving each task its own register banks (including temporary registers), set of flags, and its own program counter. The processor sends the task ID of the instruction through the pipeline along with the instruction. To switch context, the hardware scheduler has only to provide the processor with the task ID of the highest priority task that is ready to execute.

2.1.5 – The RTP Hardware Assisted RTOS

In a typical embedded system, an RTOS receives little or no support from the hardware. It is required to perform context switches, manage memory, protect shared resources, pass messages, handle interrupts, and schedule tasks. The vast majority of these functions are typically done in software by the kernel. The Real Time Processor Operating System (RTPOS) is a basic micro-kernel that implements all the necessary functions for managing a real-time system. It provides task structures, blocking and non-

blocking communication between tasks via the resource primitives previously described, events, pre-emptive scheduling, priority inheritance, and system timers.

RTPOS is based in part on the Atalanta multiprocessor RTOS kernel as found in [15]. In the RTP system architecture, almost all of the functionality traditionally done in software is moved to hardware using customized instructions, a hardware scheduler, and the TRM. The cost associated with this additional hardware is not prohibitive on an FPGA, where logic elements are abundant. By moving many of the normal kernel functions to hardware, the size of the RTOS code in the scarce memory space on an FPGA is drastically reduced.

With the hardware assistance previously described, RTPOS is able to perform all of the traditional functions described in [15] while only needing a small amount of memory, about 315 machine instructions. This small kernel size allows the RTPOS code to be stored locally for each processor in the system. By comparison, MicroC-OS [16] compiled for a Xilinx MicroBlaze requires over 2000 instructions, roughly seven times greater. It would take a similar increase in the number of instructions to implement the RTPOS purely in software.

Much greater detail regarding the RTPOS kernel including detailed descriptions of system calls and comparisons with the Atalanta and MicroC-OS kernels can be found in [17].

2.2 – The RTP Processor and Instruction Set Architecture

The RTP processor was initially designed to use a 16-bit instruction width, but that did not provide adequate encoding space, so an 18-bit instruction width is used. This

works nicely with the block RAMs on the Spartan 3-1500 and Virtex II, which are also 18-bits wide. A single processor running a single task requires approximately 420 slices on a Xilinx Spartan 3-1500. The RTP processor also includes several custom instructions that allow the RTOS to use small software kernel functions to control the hardware and manage application software.

The RTP processor instruction set includes a variety of custom instructions that directly manipulate the hardware for a specific task or resource to assist the RTPOS kernel. These instructions require approximately 15% of the 18-bit instruction decode space of the RTP processor.

Resource specific control instructions in this new architecture include the following:

- LOCK r: Attempt to reserve system resource r. If unsuccessful, task is blocked until the resource becomes available.
- NB-LOCK r: Attempt to reserve system resource r without blocking. Return information about the task that has locked the requested resource. Using this information, a task can determine if it successfully locked the resource, or if not, what other task has locked it.
- REL r: Release system resource r so other tasks may reserve it.
- RST r: Reset system resource r to initial state.
- ENABLE r: Enable system resource r. The meaning of this instruction varies for each resource.
- DISABLE r: Disable system resource r. The meaning of this instruction varies for each resource.

- SIG1 r: Send signal 1 to system resource r. The meaning of this instruction varies for each resource.
- SIG2 r: Send signal 2 to system resource r. The meaning of this instruction varies for each resource.
- READ r: Read the status of system resource r. The meaning of this instruction varies for each resource.
- WRITE r: Write to the status of system resource r. The meaning of this instruction varies for each resource.

Task specific control instructions include the following:

- R_PRIO: Read task information about the current task and store it in a register. This information includes task ID, processor ID, task status flags, and task priority.
- W_PRIO: Write the task information stored in a register to a specified task. This is used primarily for priority inheritance.
- R_TIME: Read the current task's timeout counter.
- W_TIME: Write a value to the timeout counter of the current task.

There are also 3 predefined functions calls that are handled by the compiler which facilitate the real-time processing framework (described in detail in Chapter 4). Memory is accessed via register direct addressing only. The instructions have the format "*OPERATION destination*, *source*" where several instructions do not require the source and/or destination fields. Most instructions execute in 1 clock cycle. Branching and real-time support instructions may introduce stalls in the pipeline. The implementation details

of the RTP processor can be found in [10]. The RTP instruction set is documented in Appendix D.

2.3 – Related Work

Using separate register banks to reduce the penalty for context switching has been done before. The SPARCLE processor, introduced in [18], used four banks of registers to store separate contexts. When switching threads, a trap handler would save the old PC and status register, then change the current window pointer to the register bank for the new context, and finally restore the new PC and status register. This greatly reduced the amount of data saved and restored by the trap handler. The RTP architecture provides completely separate contexts for each task, including the PC and status register. Context switching is done by the hardware scheduler, not in software as is done by the SPARCLE processor.

In the Silicon TRON project [19], hardware was used to shorten system calls and speed up scheduling. Modules were created for event flags, semaphores, timers, tasks, the scheduler, and a control circuit that interfaced with the CPU using an interrupt and status register. This reduced the RTOS kernel code size by half. In [20], the time required for context-switching in a real-time system was reduced by 50% by adding a register cache to a MIPS R3000 core in an ASIC. The RTP architecture extends these ideas by adding additional hardware to nearly eliminate the time cost of context-switching and reduce further the RTOS kernel code size.

The RTP architecture uses a sparse task-resource matrix to manage system resources. The TRM is somewhat similar to the matrix described in [21]. That research

described a dense task-resource matrix used for detecting deadlock via reduction of unused entries. The RTP architecture uses the TRM to drive a hardware scheduler.

2.4 – Additional Considerations

Four years ago when we first designed the RTP system architecture, we decided upon a custom processor to allow us the ability to define and implement in hardware a number of instructions that allowed us to manipulate the hardware directly for a specific resource or task. However, the RTP system architecture would be equally viable with an off-the-shelf processor such as the Xilinx MicroBlaze. The MicroBlaze has support for up to 8 Fast Simplex Link (FSL) connections, each of which can be connected to a coprocessor. The custom instructions implemented in the RTP processor could be realized via custom coprocessor units attached to the FSL bus.

Without the use of the RTP processor, the RTP system architecture still provides a compelling methodology of handling interrupts by eliminating the need for interrupt handlers. The hardware scheduler, in conjunction with the TRM, can point the processor directly to the task that is blocked on the interrupt. Instead of the two full context switches usually required to process an interrupt, only a single context switch is needed to switch to the new task.

Using a processor such as the MicroBlaze would provide additional functionality not available in the RTP processor, such as hardware dividers, floating point support, and more.

Chapter 3: Porting the SDCC Compiler

The Real Time Processor (RTP) hardware architecture is a vanilla 16-bit pipelined RISC-like architecture with some additional support for real-time processing. The architecture includes sixteen 16-bit registers, supporting native 16 and 32-bit arithmetic and logical operations. It does not have a hardware integer divider, but will compile division/modulo operations to an optimized sequence of assembly instructions. There are 3 predefined functions calls that are handled by the compiler which facilitate the real-time processing framework. Memory is accessed via register direct addressing only. The instructions have the format "*OPERATION destination*, *source*" where several instructions do not require the source and/or destination fields. Most instructions execute in 1 clock cycle. Branching and real-time support instructions may introduce stalls in the pipeline. The RTP instruction set is documented in Appendix D.

Note: The remainder of this chapter is intended as a standalone document suitable for publication and/or classroom use.

One major cost relating to the development of a new processor is the need to develop an accompanying C-compiler for that processor. Rather than writing the Real Time Processor (RTP) compiler from scratch, three retargetable compilers were considered: GCC, LCC, and SDCC. After considering the strengths and weaknesses of the three compilers, and the perceived difficulty of retargeting each compiler for the new architecture, SDCC was chosen as the base compiler. This chapter describes the modifications necessary to the base SDCC compiler that are required to target a new architecture. The February 26th, 2004 release of version 2.4 was used as the basis for this port. The development environment consisted of Visual Studio 6.0 on a computer running Windows XP, and as a result some of the described changes may not apply to other development environments or platforms.

3.1 – Decision to use SDCC

SDCC is an open source, retargetable, optimizing ANSI-C compiler [8] designed for 8-bit microprocessors. It supports global sub-expression elimination, loop optimizations (loop invariant, strength reduction of induction variables and loop reversing), constant folding and propagation, copy propagation, dead code elimination, jump-tables for `switch` statements, a programmable register allocation scheme, a customizable peephole optimizer using a rule-based substitution mechanism, and it allows inline assembly code to be embedded anywhere in a function. The specifics of the aforementioned optimizations are covered in detail in the SDCC documentation. Retargeting is accomplished by writing C code to translate the SDCC object code data structures to assembly instructions for the target architecture. It also offers the flexibility

to provide custom translations for a list of function calls that are specified in the compiler.

SDCC was chosen due to its many optimizations, ease of handling the predefined RTP-specific function calls, and relative ease in developing the register allocation and code generation schemes. Retargeting SDCC involves writing, or rewriting C code. No other languages need to be learned, with the peephole rule language as a possible exception.

3.2 – Base SDCC Complier Functionality

The SDCC compiler uses seven phases to compile C source code to optimized assembly. The first 4 phases are architecture independent, while the last 3 phases are almost wholly dependent on the target architecture. The phases (which are briefly described in the SDCC documentation [8]) are:

- **Parsing**. In this phase, the C source is parsed, and the abstract syntax tree (AST) is generated. Syntax and semantic checking are done in this phase, as well as some high level optimizations.
- **iCode Generation.** This phase takes the AST from the first phase, and generates three-operand intermediate codes (iCodes) for an abstract architecture with unlimited registers. Each of the registers is designated a unique name iTempXXX, where XXX represents a numeric value. If desired, a human readable version of the iCodes can be printed to a file using the `--dumpraw` compiler flag.

- **Code Optimizations.** This phase converts iCodes into basic blocks. Basic blocks are blocks of sequential code which are guaranteed to execute without jumps or branches. This attribute makes them easy to analyze and optimize. The basic blocks are put through data and control flow analysis to perform the following optimizations: local and global common subexpression elimination, dead code elimination, and loop optimizations. This sequence of optimizations is repeated each time the loop optimizations result in changes to the basic blocks.
- **Live Range Computation.** This phase determines when the iTemps are used, from initial assignment until final use.
- **Register Allocation.** The register allocation phase is actually two stages [9]: 1) register allocation, where the set of variables that will exist in registers is determined, and 2) register assignment, where specific registers are chosen for each variable. Architecture-specific expression folding, or register packing, is done in this phase which reduces register pressure. This phase uses the live ranges computed in the previous stage to assign remaining iTemps to physical registers on the target architecture. Code from similar architectures may be used as a basis for register allocation of a new architecture.
- **Code Generation.** This phase maps the iCodes to assembly instructions. Very little code from similar architectures may be used within this phase. However, the general methodology of assigning assembler operands to individual iCodes, as done in previously implemented architectures, is a good starting point for code generation.

- **Peephole Optimizations.** This phase uses a rule-based matching system to optimize certain sequences of assembly code. Few rules from existing architectures will apply to new architectures. Appendix A lists the peephole optimizations for the RTP architecture.

The base SDCC compiler lexes and parses the C source file using FLEX, YAK, and BISON. It then builds the abstract syntax tree (AST) from the output of the parser. Details regarding lexical analysis and parsing will not be discussed in this document, nor will details concerning the generation of the AST.

There are many options in the base SDCC compiler that are architecture specific. These options modify the handling of the code during code generation, with parameters such as memory models, data type representation sizes, and the various segments of memory in the target architecture. They also specify architecture specific command-line option handling, and pragma processing.

3.3 – Required Changes to Base SDCC Compiler

The SDCC compiler was originally designed to target 8-bit architectures, and byte addressable memory schemes. The RTP architecture is a 16-bit architecture, with byte addressable (and word aligned) memory. Memory and register allocation, and memory addressing all required several changes to the base SDCC compiler. Wherever possible, the RTP port avoids making changes to the base SDCC compiler source files, as they are used by the other supported architectures. The changes that must be made to the base compiler are always preceded by `if (TARGET_IS_RTP) {}` to guarantee that other ports will not be affected. A more rigorous handling of 16-bit architectures would

include modifications to the PORT structure with options for memory width and alignment, and register sizes. This approach was not taken, as the RTP port is not a part of the official SDCC source tree, and such an approach would require a multitude of changes that would make it difficult to sync with the latest stable branch of the source tree.

The base SDCC compiler assumes an 8-bit architecture, and therefore allocates memory for data structures and registers on the byte-level. This is problematic when targeting the RTP's 16-bit architecture, as register allocation and memory addressing must be done on the 2-byte word level to avoid memory misalignment exceptions. A modification to SDCCmem.c is required for the functions `allocParm`, `allocLocal`, and `deallocLocal`, so that the amount of memory allocated for character variables is padded when it will cause a non-character data element to have a misaligned memory address. To further guarantee that non-character data is accessed on word boundaries, when code is emitted to allocate memory for any data element, an assembler pragma ".even" is emitted before the allocation.

For register allocations, each call to `getSize()` (a helper function that returns the size of an element in the symbol table) is halved, except in the case of character variables. With an 8-bit scheme `getSize()` returns the number of bytes for each data type, which directly corresponds to the number of 8-bit registers required. A 16-bit architecture requires the number of 16-bit words for register allocation and memory addressing, including stack offset pointers.

Word-aligned memory requires changes to the symbol table function for struct size computations. The base SDCC compiler allows structs and arrays to be placed at

any offset in memory, and allows them to have odd valued sizes. If a member of a struct is not a character, it requires that it be word aligned for memory access. The symbol table function `compStructSize()` computes the size of a struct, and therefore was modified to take into account word alignment for non-character elements. The size of the entire struct is also modified to guarantee an even number of bytes.

In addition to the above mentioned changes, several modifications must be made to the port.h and SDCCmain.c files. At the very beginning of port.h, all of the supported target architectures are given a unique ID. A new entry must be added at the end of the list for the new architecture. After the assignments of unique IDs are several macros (`TARGET_IS_<port>`) that are used to test which architecture is being targeted. A new macro should also be created at the end of this list. The last few lines in port.h provide each of the supported architectures with an `extern PORT <port>_port` declaration for use throughout the base SDCC compiler. An additional declaration should be provided for the new architecture. The final necessary change is in SDCCmain.c, beginning at approximately line 275. At this point in the code an array of all targetable architectures is defined, and the new architecture should be appended to the end of this list as well.

If desired, SDCC_vc.h can be modified such that only one architecture is supported by the compiled binary. This is done by modifying the end of the file to contain a `#define OPT_DISABLE_<port>` line for all unsupported ports defined in port.h.

Targeting the SDCC compiler to a specific architecture for code generation requires the creation of the following 6 files:

- main.c, main.h – These files define the built-in functions that should be trapped by the compiler, the default port variables, and the peephole optimization rule files. Also, keywords that pertain to the architecture can be specified here in order to be properly handled by the lexer. Some port specific functions are also defined here if needed, as well as any declarations and definitions that need a global architectural scope. The main.c source file also contains code to process `pragma` statements.
- ralloc.c, ralloc.h – These files are used to describe the layout of the register file, and the manner in which registers should be allocated for code generation. Also specified are general purpose and scratch register allocation, and any special purpose registers that may be required for the architecture. The live ranges for registers are computed by the base SDCC compiler during creation of iCodes and basic blocks, and subsequently used within ralloc.c for the register allocation and assignment. Any port specific register packing is done in ralloc.c.
- gen.c, gen.h – These files are used for the assembly code generation. An in-depth understanding of the ISA for the target architecture is a pre-requisite. A familiarity with the target architecture's memory addressing modes is also a pre-requisite. An understanding of the data structures that the base SDCC compiler generates and the way they map to the original source code is extremely helpful. The gen.h header file contains the definition of assembly operands and operand types, while gen.c contains the assembly code generation routines.

The changes required for code generation will be covered in this chapter, while Chapter 3 discusses the changes required for trapping the predefined RTP function calls.

3.4 – The SDCC-RTP Compiler: Code Generation

Porting of the SDCC compiler is accomplished in the last 3 stages discussed in Section 3.2. The remainder of this chapter will discuss the details of assigning port options, implementing a register allocation scheme, generating assembly code, and a method of generating peephole optimizations to optimize and reduce code size.

3.4.1 – Port Options (main.*)

The source files main.c and main.h are the files that pertain to port-specific options. The header file is used to declare any variables that require a global scope for the architecture specific files. In the RTP port, there are no declarations or definitions required in main.h.

The main.c source file is used to declare architecture-specific variables and functions. The most important aspect of main.c is the declaration of all port specific options, primarily the variables within the PORT structure. These variables are used throughout the base SDCC compiler to determine the methods for handling iCode generation, memory segmentation, and several optimizations. The prototypes of the built-in functions that will require special handling are defined within a special structure called `builtins`, which is discussed in more detail in Chapter 3. Any special keywords that should be parsed and appropriately handled by the code generator are also defined in main.c. Finally, several helper functions are defined that are called by the base compiler to parse command line options, parse `#pragmas`, and appropriately handle parameter passing via registers.

The comments contained within the structure definition in port.h are extremely helpful. Another useful resource for assigning port options is to refer to a previously targeted architecture's PORT declaration as an example.

3.4.2 – Register Allocation (ralloc.*)

The main function for register allocation is _<port>_assignRegisters (eBBlock **ebbs, int count), which is a function call specified in the architecture's PORT structure. This function is called by the base SDCC compiler, and takes as parameters a pointer to a list of basic blocks, and the number of basic blocks that require register assignment. Registers must be allocated per given function, so the list of basic blocks passed in will only contain the basic blocks for the current function.

A basic block is composed of various elements, most of which aid in control flow and dataflow analysis. The elements of the basic block that factor into register allocation are 1) iCode *sch, which is a linked list of the iCodes contained by the block, and 2) the integers fSeq and lSeq, which are used in conjunction with the seq field of each iCode to determine the locality of iCodes with respect to basic blocks.

Table 3.1: Register Assignments

Registers	Purpose
R0-R3	Scratch registers (caller saved)
R4-R8	General purpose registers (callee saved)
R9 (and R10)	Return value registers
R9-R12	Parameter passing registers
R13	Stack pointer
R14	Frame pointer
R15	Return register

The RTP register file is made of 16 16-bit registers, with the registers assigned as listed in Table 3.1. Note that R10 is only needed as a return register for 4-byte variables. The register types and register file layout are defined in ralloc.h.

The sequence outlined below is architecture independent, but the implementation of the register packing and register allocation functions that are called is highly architecture specific. Register allocation for the RTP architecture is done by taking each of the following steps in order:

- Pack the registers for each basic block, using architecture-specific optimizations to reduce register requirements. Specific optimizations are discussed following this list.
- Recompute the live ranges for all basic blocks, in the event that register packing altered the positions (or existence) of some variables. This is done through a call to a helper function within the base SDCC compiler, `recomputeLiveRanges (ebbs, count)`.
- Analyze the recomputed live ranges to determine the type and number of registers required (register allocation).
- Assign physical registers to the variables that require them (register assignment).
- Create the register mask, and update the corresponding field for each iCode. The register mask is used to determine all registers that are active during the use of the specified iCode, and can be used for other optimizations during code generation.

- Call the helper function `redoStackOffsets()`, which will update the stack size and stack offsets required by the function, now that registers are given a definite assignment.
- Get a pointer to the first iCode in the chain for the basic block, by calling `iCodeLabelOptimize (iCodeFromeBBlock (ebbs, count)).`
- Call the code generation procedure `gen<port>Code (iCode *ic)` to generate code for the current function.

The code found in the AVR-specific register packing was used as a basis for the RTP register packing, and is very straightforward. The following paragraphs will discuss the various register packing scenarios.

First, true symbols (symbols that do not have the iTemp field set) that are used in an assignment to an iTemp (i.e. iTempXX = TrueSym OP *operand*), and are subsequently used in an assignment such as TrueSym = iTempXX. The latter assignment can be replaced with the former assignment, which is the definition of iTempXX. This will free up the registers used in the second assignment operation, and will most likely shorten the live range of such registers.

Second, there are several cases where register use is unnecessary, such as the address of a true symbol, rematerializable data, and addition and subtraction operations that use a rematerializable operand in conjunction with a literal. In these cases, the operands in question need not be stored in registers because they can very easily be determined. Rematerializable data and operands are by definition such data and operands that are the result of an operation that can be computed at compile time, and therefore need not be assigned to registers.

Thirdly and lastly, in register packing, there are several cases where the register is needed for only one use, and possibly can be optimized away. In the case of variable casting for integral promotion, for instance, if the register use being analyzed is the only use of the arithmetic operation involved in the cast, then the cast can be replaced by the result of the arithmetic operation. In the case of return values, redundant moves can be optimized away.

After register packing is performed, register allocation and assignment must be performed. Register allocation involves checking all of the various live ranges for registers that are in use. For all iTemps that are needed for a given live range, the number of required registers is computed based on the size of the data stored by the iTemp. The type of register required is also determined at this stage. If the value will be needed across function calls, then a general purpose register is preferred to avoid excessive saving and restoring of registers. Otherwise, a scratch register is adequate.

Finally, register assignment takes place. All iTemps that require registers are assigned to physical registers in the architecture. When the number of physical registers required for a given live range exceeds the number of physical registers available, iTemps are said to "spill" onto the stack, which results in an increased stack size, and assignment of stack offsets to iTemps.

3.4.3 – Code Generation (gen.*)

After registers have been assigned, the next step is to generate code. Code generation is the mapping of intermediate code instructions to machine-specific assembly code instructions. Each intermediate code instruction has a specific operation that needs

to be performed. In the RTP architecture there are 40 operation codes that are recognized by the code generation algorithm. These operation codes are listed in Table 3.2. There are several other operation codes defined in sdccy.h that may be recognized by other architectures that provide support for them. If the options to transform the various comparison operations to equivalent forms are enabled in main.c, even fewer operation codes will be necessary.

Table 3.2: RTP Recognized iCode Operation Codes

Operation Code	**Notes**
`!`	Logical NOT
`~`	Bitwise complement
`UNARYMINUS`	Unary minus
`IPUSH`	Push onto stack
`IPOP`	Pop from stack
`CALL`	Procedure call
`PCALL`	Procedure call via pointer
`FUNCTION`	Function startup boilerplate
`ENDFUNCTION`	Function cleanup boilerplate
`RETURN`	Return from procedure/function
`LABEL`	Generate label
`GOTO`	Goto label
`+, -, *, /, %`	Add, subtract, multiply, divide, modulo
`>, <, LE_OP, GE_OP, NE_OP, EQ_OP`	Comparisons {>, <, <=, >=, !=, ==}
`AND_OP, OR_OP`	&& and ||
`^, |, BITWISEAND`	Bitwise XOR, OR, AND
`INLINEASM`	Generate inline assembly
`LEFT_OP, RIGHT_OP`	Left and right shifting
`GET_VALUE_AT_ADDRESS`	Get pointer value
`=`	Set pointer value or Assign
`IFX`	If X statement
`ADDRESS_OF`	Generate address of variable
`JUMPTABLE`	Create jump table
`CAST`	Variable casting
`RECEIVE`	Parameters being received
`SEND`	Parameters being passed
`ARRAYINIT`	Array initialization

The gen.h header file contains definitions for the various assembly operand types (literal, register, pointer, etc), and a structure that contains all pertinent information about the assembly operand, including type, memory segment, size, and value. Each iCode references up to three operands: result, left, and right. The code generator determines which operands are valid based on the operation code, and creates one or more assembly operands for the corresponding iCode operand.

All ANSI-C compliant source code[1] can be broken down into a combination of the operation codes listed in Table 3.2. In most cases, only a single iCode is needed to generate code. In the few remaining cases, it may be necessary to look at the next iCode in the chain for example to determine the branch target after a comparison operation.

Figures 3.1 through 3.3b illustrate an example of the flow from C code to assembly code for the RTP processor. Figure 3.1 shows the original C source code that will be used in this example. There are several iCode operations in this short example: assignment, increment/decrement, multiplication, comparisons, and a function call. The variable `c` is a global variable, so it will not be allocated a register. In the `doubleit` function, `op` is passed in via r9, the first parameter passing register, and since r9 is also the return value register, no additional register allocation is required. In the `main` procedure, `a`, `i`, and `result` are local variables and will be allocated registers. The variable `b` is not allocated a register because its value is a constant (rematerializable), and is replaced inline with the scalar value 2.

[1] There are several deviations from ANSI-C compliance. See section 8.2 of the SDCC documentation for further details.

```
 1: char c;
 2:
 3: int doubleit (int op) {
 4:   return (op*2);
 5: }
 6:
 7: int main() {
 8:
 9:  int a, b, i, result;
10:
11:  a = 1;
12:  b = 2;
13:  c = 'q';
14:  result = 0;
15:
16:  for (i = 0; i < 10; i++)
17:    a = a * b;
18:
19:  while (c != 'm')
20:    c--;
21:
22:  if ( a < 1024)
23:    result = 1;
24:
25:  switch(c) {
26:    case 'q' :
27:      a = doubleit(a);
28:      break;
29:    case 'm':
30:      result = 2;
31:      break;
32:    default:
33:      result = 3;
34:  }
35:  if (a != 2048)
36:    result = 4;
37:
38:  Return result;
39:} /* main */
```

Figure 3.1: C source code for RTP code generation example

When this code is first parsed by SDCC-RTP, the iCode chains contained in Figure 3.2 are created (comments added in parentheses), with the exception that LABEL iCodes for entry into `doubleit` and `main` have been omitted. The iCodes in each

```
Chain 1:
 1: FUNCTION (doubleit)
 2: RECEIVE (parameter)
 3: LEFT_OP (left shift)
 4: RETURN (return value from doubleit)
 5: ENDFUNCTION

Chain 2:
 1: FUNCTION (main)
 2: '=' (assignment for variable 'a')
 3: '=' (assignment for variable 'c')
 4: '=' (assignment for variable 'result')
 5: '=' (assignment for variable 'i')
 6: LABEL 1a (for loop jump target)
 7: '*' (multiply operation)
 8: '-' (decrement variable 'i')
 9: IFX (comparison on 'i'; branch LABEL 1b; jump LABEL 1a)
10:   LABEL 1b (for loop completed jump target)
11: LABEL 2a (while loop jump target)
12: NE_OP (comparison of c != 'm')
13:   IFX (branch LABEL 2b)
14: '-' (decrement variable c)
15: GOTO (jump to LABEL 2a)
16: LABEL 2b ()
17: '<' (comparison of a < 1024)
18:   IFX (branch LABEL 3)
19: '=' (assignment for result = 1)
20: LABEL 3 (comparison jump target)
21: EQ_OP (c == 'q')
22:   IFX (branch LABEL 4a)
23: EQ_OP (c == 'm')
24:   IFX (branch LABEL 4b)
25: GOTO (jump to LABEL 4c - default switch target)
26: LABEL 4a ()
27: SEND (set up parameters for doubleit)
28: CALL (call doubleit)
29: '=' (assign a = doubleit(a))
30: GOTO (LABEL 4d)
31: LABEL 4b
32: '=' (assign result = 2)
33: GOTO (LABEL 4d)
34: LABEL 4c
35: '=' (assign result = 3)
36: LABEL 4d (switch jump target)
37: NE_OP (compare a < 2048)
38    IFX (branch LABEL 5)
39: '=' (assign result = 4)
40: LABEL 5 (comparison jump target)
41: RETURN (return value of 'result')
42: ENDFUNCTION (main)
```

Figure 3.2: iCode chain for RTP code generation example

chain are processed sequentially. There are some cases where iCodes are created during code generation. One example of this can be found in Chain 2 iCode 10, which has been underlined. In this case, the IFX iCode creates a LABEL iCode to use as a branch target. Note that this iCode is not inserted into the chain, but is simply created and used dynamically.

It should also be noted that in Figure 3.2, the iCodes that are indented 2 spaces are either created during code generation (line 10), or are processed jointly with the iCodes that precede them (iCodes 13, 18, 22, 24, and 38). This will result in different sequences of assembly instructions being generated for the same iCode operation when processed jointly than when processed individually.

Figure 3.3 contains portions of the assembly code generated from the iCode chains in Figure 3.2. First is listed the header and initialization code as well as the doubleit function. The iCodes that are listed in Chain 1 from Figure 3.2 are included in assembly comments with the corresponding lines of the doubleit function. Figure 3.3 also contains an excerpt from the main function, and lists the corresponding iCodes from Chain 2 in Figure 3.2. For a full listing of the generated assembly code, including peephole rule optimization comments, please see Appendix B.

It is recommended in porting the SDCC compiler to maintain a separate code generation procedure for each different iCode operation. This allows for easy debugging of generated assembly code, and quickest time to release. The tradeoff in this implementation decision is sub-optimal code, which may be optimized via additional peephole rules.

```
      .module example1
      .globl _main
      .globl _doubleit
      .globl _c
      .area DSEG  ;(DATA)
_c::
      .even
      .ds 1

      .area GSFINAL     ;(CODE)
.globl __sdcc_init_data
__sdcc_init_data:
      ret   ;return to caller
      .area CSEG  ;(CODE)

_doubleit:
      push  r14          ;(1.1)
      mov   r14,r13      ;(1.1)
      sll   r9,0x1       ;(1.2,1.3,1.4)
_ret_doubleit:           ;(1.5)
      pop   r14          ;(1.5)
      ret                ;(1.5)


; excerpt from _main:
      mov   r4, 0x1      ;(2.2)
      mov   r0,0x71      ;(2.3)
      mova  r1, _c       ;(2.3)
      st    r0,0(r1)     ;(2.3)
      mov   r5, 0x0      ;(2.4)
      mov   r6, 0xa      ;(2.5)
L00016:                  ;(2.6)
      mov   r0,0x2       ;(2.7)
      mul   r4,r0        ;(2.7)
      sub   r6, 0x1      ;(2.8)
      be    L00027       ;(2.9)
      jmp   L00016       ;(2.9)
L00027:                  ;(2.10)
L00003:                  ;(2.11)
      mova  r0, _c       ;(2.12)
      ld    r1, 0(r0)    ;(2.12)
      cmp   r1, 0x6d     ;(2.12)
      be    L00005       ;(2.13)
      mova  r2, _c       ;(2.14)
      ld    r1, 0(r2)    ;(2.14)
      sub   r1, 0x1      ;(2.14)
      st    r1, 0(r2)    ;(2.14)
      jump  L00003       ;(2.15)
L00005:                  ;(2.16)
      mova  r0,0x400     ;(2.17)
      cmp   r4,r0        ;(2.17)
      bge   L00007       ;(2.18)
      mov   r5, 0x1      ;(2.19)
L00007:                  ;(2.20)
```

Figure 3.3: Assembly code for RTP code generation example

3.4.4 – Peephole Rules (peeph.def)

Optimal code generation is an NP-complete problem. Devising a code generation algorithm that always generates the most optimal code is impractical if not impossible. A simple way to improve sub-optimal code is to pass the generated assembly language instructions through a peephole optimizer [12]. A peephole optimizer recognizes certain instruction patterns, and replaces them with optimized instruction patterns. Each of these replacement patterns is called a peephole rule. This optimization process is given the name "peephole" because it is limited to analyzing a small portion of the code at any given time.

One method for creating a set of peephole rules is to examine the assembly source code generated for several different source files, and look for patterns of instructions that occur frequently. There may be instruction sequences that are not only sub-optimal, but entirely unneeded. A peephole rule should be created for this sequence to eliminate it entirely. A good rule of thumb in both hardware and software design is to make the common cases fast, while ensuring that the difficult cases remain correct.

There are other cases where the compiler emits code that does not take into account all of the side effects of certain instructions. For example, there may be cases where the first instruction sets the flags required for a branching operation, but the compiler produces a compare instruction in addition to the branch. The compiler may be altered to take this into account, but more likely the lower cost alternative is to use peephole rules to optimize this code sequence.

The built in peephole optimizer for the SDCC compiler is used at compile time, and therefore cannot comprehend optimizations that span two separate iCode chains.

This particular shortcoming makes itself manifest in global declaration code, especially if multiple global variables are being initialized to the same value. Because of this limitation, a separate Perl-based peephole optimization program was designed to further reduce the code footprint. Perl was chosen due to its excellent ability to parse and manage text.

A certain subset of peephole rules can be applied to all architectures. The current set of peephole rules for the RTP architecture are listed in Appendix A.

Chapter 4: The SDCC-RTP Compiler

One particular advantage of SoPCs is the ability to customize the system architecture for each different application. As real-time embedded applications are compiled for the RTP architecture, information about the specific resource needs of each task in the application is collected and stored so that the system generator can instantiate application-specific custom hardware. The SDCC-RTP compiler targets the RTP instruction-set architecture, which contains special instructions that facilitate the efficient use of the custom hardware. The number of processors, the number of tasks, the devices, and the required system resources must all be determined statically at compile-time because only the minimal hardware necessary will be instantiated by the system generator. The dynamic creation or deletion of tasks cannot be supported as it is in software-based architectures.

4.1 – Built-in Functions

The built-in functions are used to statically declare the hardware resources, devices, and tasks of the system at compile time. The definitions associated with any hardware element must be declared in a global.h header file. The template for the global.h header file defines the different resource and device types available for the RTP architecture. As new devices or resources are implemented, additional items can be

added to the global header file definitions. The global header file is also used to specify all device and resource descriptor numbers as they are unique in the global scope. The device I/O space of each processor is private, but actual device numbers must remain globally unique. This is done by treating the device numbers as (processor number, I/O address) pairs. The global IDs are not limited to 0..255 like I/O addresses are. Figure 4.1 shows the template for the global.h header file. This global header file is included by the source files for all processors in the system, allowing them to reference global devices and resources.

```
#ifndef __GLOBAL_H__
#define __GLOBAL_H__

enum resource_type {
  ZRESOURCE, MUTEX, DIS_EVENT, CON_EVENT,
  SEMAPHORE, TIMER, INTERRUPT, READER, WRITER };

enum device_type {
  ZDEVICE, FIFO_READER, FIFO_WRITER, TRANSMITTER, RECEIVER,
  SCRATCHMEM, FP_ADD, FP_MUL };

#pragma NUMPROCS_4

/* Resources */
/* Mutexes */
#define MUTEX1    1
#define MUTEX2    2

/* Devices */
/* FIFOS */
#define FIFO1_QU1 1
#define FIFO2_QU2 2

#endif
```

Figure 4.1: The global.h Header File Template

The specification of resource and device numbers (also called device descriptors in some operating systems) is done via #define RES_NAME <NUM> statements. The preprocessor replaces all #defines with their definition, which means the compiler will see these values as if they were literals, and not variables. Literals are passed directly through the compiler's AST, and are much easier for the compiler to use to build its intermediate task-resource symbol table than integer variables. Were the device and resource numbers to be specified as variables, it would be virtually impossible to share global descriptor numbers between different source files, as each processor has its own data memory.

All processors must have their code contained within a separate source file, and must specify their processor number with a #pragma PROC<num> line. The separate source files are required because each processor has its own code memory. An advantage of using separate source files is that the main procedure becomes the idle task for the given processor. The PROC<num> pragma is parsed by the compiler and sets a global RTP processor variable. This pragma also creates an assembly directive of the form .proc_num PROC_NUM that allows the assembler/linker to produce a memory size generator directive. After parsing this pragma, the RTPsys.gen generator directive file is either created if it does not exist, or loaded into the intermediate symbol table if it does exist. To avoid unnecessary recompilation of unchanged processors, only the values for the current processor are updated in the directive file. In global.h, the total number of processors must be specified by the #pragma NUMPROCS_<num> line.

If a built-in function is called without a #pragma PROC<num> line, the compiler will terminate with a fatal error, and will print an error message to the application engineer.

The application engineer declares hardware resources, devices, and tasks through the use of the following function calls in the application code:

1) int OpenResource(int TYPE, int glob_rd_num, int task_id);
All tasks that use any hardware resource must declare such use by means of this function call. This function specifies that a resource of type TYPE must be generated for the system. The global resource descriptor number and the id number of the task that will use the resource are also passed in as parameters. Resources are not restricted to a single processor, nor to a single task within a single processor. The only error checking that is done by the compiler is to verify that the TYPE specified is consistent across all processors and corresponds to an existing hardware resource component available within the RTP VHDL library. There are no limitations to the number of processors or tasks that can use a given resource as long as they call this function before use.

2) int OpenDevice(int TYPE, int dev_num, int task_id, int base_addr, int size, int glob_rd_num, int off_chip_pin_num);
All tasks that share the same I/O devices must be assigned to the same processor. There are exceptions to this rule in the case of hardware FIFOs and scratchpad memory. In the RTP VHDL library, a hardware FIFO is composed of both a reader device and a writer device. These FIFO readers and writers can reside on the I/O buses of the same or

different processors. The OpenDevice function specifies that a device of type TYPE must be generated for the system. The global device descriptor number, the task id number, and the base I/O address are also passed in as parameters. An optional size parameter is also passed in for devices that can vary in size, which currently only pertains to FIFO and scratchpad memory devices. The final two parameters are also optional for devices that will utilize a resource or an off-chip pin. The compiler checks to see that the device does not already exist on another processor, and if it does, prints an error message to the user. On a given processor there is no limitation to the number of tasks that can share a device as long as they call this function before use. Only one task can use a specific device at a given time, therefore if a device is shared between tasks a mutex resource should be used to ensure exclusive access to the device.

<u>3) int CreateTask</u> `((void *) t_main, int t_priority, int stack_size, (void *) arg1, (void *) arg2);`

This function takes five parameters: a function pointer to the task's main routine, the initial task priority id (which is also the task id), the required stack size, and two pointer-sized arguments for the task's main routine. When the compiler parses this function call, each parameter is passed along to a function call in the RTOS, `_createtask`, which initializes the task table, and starts the task. The task table contains the stack size, task main routine start location, and two arguments for each task on the processor. The application engineer is the best judge for which tasks require the highest priority, and which tasks should be run concurrently. This is why the priority must be specified with

task creation. The compiler will catch an attempt to assign the same priority to two tasks on the same processor, and will print a corresponding error message.

4.1.1 – Compiler Support for Built-in Functions

The base SDCC compiler provides support to define a list of functions with their prototypes that will be given special support in the compiler. The original intent of this support was to bridge functions implemented in hardware to application code; however, the SDCC-RTP compiler uses this list of predefined functions as the means by which to trap the three special functions listed above and use their parameters to create the intermediate symbol table described in the following section.

The built-in functions are declared using the structure type definition listed in Figure 4.2. The final entry must be given as `{NULL, NULL, 0, {NULL}}` to specify the end of the array. As mentioned in Figure 4.2, the comments listed before the helper function `typefromStr` describes the manner in which return and parameter types should be specified within the structure.

```
typedef struct builtins {
  char *name; /* name of built-in function */
  char *rtype; /* return type given as string */
  int  nParms; /* number of parameters (max 8) */
  char *parm_types[MAX_BUILTIN_ARGS]; /*param type as string*/
} builtins; /* see typefromStr for more details */
```

Figure 4.2: Built-in Function Structure Type Definition

When the code generation algorithm sees a SEND iCode operation code, it checks to see if the iCode's `builtinSEND` field is masked. If so, the code generation

algorithm will call a special routine that handles the built-in functions, called `genBuiltIn`. In the general case, this routine should do the following:

- Call `getBuiltinParms`, which will populate an array of operands that contains the parameters passed into the built-in function.
- Compare the built-in function name against the list of built-in functions to determine which to generate code for.
- Handle the built-in function in the appropriate manner.

In the case of the SDCC-RTP compiler, the final step builds the intermediate symbol table entry for the current function, and provides a return value to the callee. In the case of the `CreateTask` built-in function, the RTOS function `create_task` is also called in order to initialize the task's startup table, which contains stack size, arguments, and initial PC address. The assembly code for the RTOS is included in Appendix C.

4.2 – Intermediate Symbol Table

A small intermediate symbol table (IST) of generator directives is created and modified using the globally shared definitions in global.h and calls to the above three functions so that the proper generator directive file can be created (updated) upon successful compilation. The intermediate symbol table is contained in a single global file RTP_ist.dat. This file is stored in binary format as human readability is not required. As mentioned earlier, this symbol table only modifies entries for the currently specified processor. The format of the IST is found in Figure 4.3.

```
struct _RTP_TaskInfo {
  int proc_id;
  int task_id;
  int task_priority;
  int stk_size;
  struct _RTP_TaskInfo *nextTask;
};

struct _RTP_directives {
  int res_or_dev; /* 0 = res, 1 = dev */
  int type;
  int res_dev_num;
  int task_id;
  int proc_id;
  int size; /* if needed, for device */
  int res_id; /* if needed, for device */
  int off_chip_pin_num; /* if needed, for device */
  struct _RTP_directives *nextDir;
};
```

Figure 4.3: Intermediate Symbol Table for Generator Directives

4.3 – Generator Directives

The three built-in function calls create entries in the IST, which in turn produce specific generator directives that signal to the RTPGen hardware generator the nature and grouping of the tasks, devices, and system resources. The RTP architecture uses a system specific customized task-resource matrix (see Fig. 2.1) for allocating system resources to tasks, keeping track of task state, and maintaining mutual exclusion for shared resources.

The compiler generated directive file is created upon successful compilation of each processor. The file is created directly from the IST, and is done in ASCII format. This lends itself to human readability so that the application engineer can review the final directive file. The format of the ASCII directive file is as follows, with each directive line terminated by a newline character:

```
P_<NUM_PROCS>
M_<PROC_ID> :CODE_SIZE :CODE_FNAME :DATA_SIZE :DATA_FNAME
T_<ID> :Proc_ID :TASK_PRIORITY
R_<glob_RD_num> :TYPE :PROC_ID :TASK_ID
D_<glob_DD_num> :TYPE :PROC_ID :TASK_ID :BASE_ADDR :SIZE
:RES_ID :PIN_NUM
```

The P_ directive specifies to the generator the number of processors in the system. The M_ directive is created by the assembler/linker to provide the code and data memory sizes and filenames required for the given processor. The T_ directive specifies to the generator the task modules that need to be created for the given processor at the specified priority level. The R_ directive specifies a task-resource node in the task-resource matrix for the given processor and task. The D_ directive specifies a device that is accessed by the given processor and task, with its base address and optional size parameter specified. The resource id and pin number parameters are also optional for devices. The P_, R_ and D_ directives describe to the generator each hardware unit that must be instantiated in the top level VHDL file.

Chapter 5: The Assembler and Linker

After the C source files have been compiled to assembly language instructions, the assembler and the linker take the assembly files created by the compiler and generate machine code and data memory images that are downloaded onto an FPGA. This chapter will first discuss the implementation details of the assembler followed by the implementation details of the linker.

5.1 – The Assembler

The purpose of the assembler is to translate the assembly instructions created by the compiler into machine code instructions. The as-rtp assembler is based on the ASXXXX assembler, a freeware retargettable assembler included in the SDCC source tree. The ASXXXX assembler consists of two main parts: 1) the "generic" part that remains unchanged between architectures, and 2) the architecture specific code that specifies the manner in which the byte-codes should be emitted.

The generic part of the ASXXXX assembler parses and lexicographically analyzes the assembly source code, and builds the symbol table. In order to properly interface with the architecture-specific labels and keywords that should be recognized, a mnemonic structure must be created that lists all of the recognized mnemonics, and the corresponding bit-patterns that should be generated for each mnemonic. These

mnemonics are categorized by syntactic groupings, i.e., arithmetic, control, or branch operations.

The architecture specific code must do the following:

- Receive the mnemonic from the generic part to lookup the syntactic group.
- Reads the correct number of operands based on syntactic group.
- Determine the operands for the op-code, using a helper function to retrieve operand types.
- Determine the actual op-code (byte-code) based on operand types, i.e., register-register or register-literal.
- Emit byte-code, or relative code if relativeness cannot yet be determined.

The assembler creates a .rel file, which is the relocatable code file that the linker will process. The format of the .rel file can be seen in Table 5.1. The Assembler begins

Table 5.1: Assembler Relocatable File Format

Line Type	**Format** **Meaning**	**Example(s)**
XH (1st line)	%c%c Hex format, MSB first	`XH`
H (2nd line)	H %d areas, %d global symbols Summary	`H 5 areas, 4 global symbols`
M (3rd line)	M <module name> Module name	`M example`
A	A <seg> size %d flags %d Area (segment) information	`A CSEG size 6E flags 0`
S	S <symbol> [Ref|Def|Abs]%d Global Symbol in current segment	`S _main Def0008`
T	T aa <addr> <inst> <inst> <inst> True code emitted	`T 03 0000 1500E 0400E (code)` `T 03 0000 F3 3A 77 22 (data)`
R	R aa <addr> <how> <symbol> Relocatable patch instructions	`R 03 0001 AREA_MOVA SSEG` `R 03 0007 JUMP _ex1_code`

the .rel file with 3 lines that describe the contents of the remainder of the file to the linker. All global symbol definitions pertain to the most recent segment specified on an A line. T lines contain either code or data emissions, within relative code or data space of the specified segment. If the relativity of the code is not yet known, then an R line is emitted to signal to the linker that the code must be patched at link-time. Table 5.2 lists the six Relocatable patch instructions.

Table 5.2: Assembler Relocatable Patch Instructions

Patch Instruction	Resolution
`DATA`	Patches a 16-bit address reference in a data segment
`MOVA`	Patches 2 16-bit addresses in a "MOVA Rx, Symbol" instruction
`MOVL`	Patches lower 16-bit address in a "MOVA Rx, Symbol" instruction
`MOVH`	Patches upper 16 bit address in a "MOVA Rx, Symbol" instruction
`JUMP`	Patches a 12-bit absolute address in a jump or call instruction
`BRCH`	Patches a 10-bit PC-relative address in a branch instruction

The code for a given source file may refer to symbols (either code or data) that exist in a separate source file or segment. The assembler emits code on a per-segment, per-source-file basis, which is why the relativity of the code may be unknown at time of assembly.

5.2 – The Linker

The purpose of the linker is to take the relative code emitted by the assembler and to patch up each relocatable address with an actual address. The final output of the linker is executable machine code and data memory images for each processor required by the

system. In comparison to the amount of code required by the assembler, the linker is very simplistic.

The linker utilizes a module/segment approach to link the assembled code. Each .rel source file is considered a module, and within each module there are different segments for code and data. See table 5.3 lists the valid segments that have been predefined in both the assembler and the linker.

Table 5.3: RTP Segments

Segment	Description	Type
`CSEG`	Code Segment	Code
`GSINIT`	Global and Static Initialization Segment	Code
`GSFINAL`	Global and Static Final Segment	Code
`DSEG`	Data Segment	Data
`SSEG`	Stack Segment	Data

For each module, the linker starts to build pieces of each defined segment as they are encountered. If the linker finds multiple pieces of the same segment in different locations within the same module, the assembled code or data is appended to the current contents of that segment. Within each segment will be the three line types described in the previous section as R, S, and T. True code and global symbols are placed in the memory images as they are, and data structures are created for the relocatable code. When all modules have been read in, the linker uses the specific patch instructions to iterate through all of the relocatable code, and assign to each an actual memory location, as all true code and global symbols now have specific memory addresses.

After all of the relocatable code has been relocated, what is left is a true image of code and data memory. The linker then outputs a single CODE segment, which is created with ordering of the sub-segments as follows:

1. GSINIT code, if present
2. GSFINAL code, if present
3. CSEG code, if present

The DATA segment is created such that the DSEG and SSEG are together in the same file. The total length of the DSEG and the total length of the SSEG are added together and rounded to the nearest 1K bytes, as the DATA on the FPGA must fit into an even number of block RAMs. The DSEG is placed at the bottom of this allocated DATA segment, and the SSEG is placed at the top of the DATA segment, to allow the stack to grow downward. If the RTP platform were enhanced to support `malloc` or similar memory allocation mechanisms, then an additional HSEG (Heap segment) would need to be defined, and would be placed in the DATA segment just above the DSEG to allow the heap to grow upward.

The Linker can take command line arguments to specify the file format of the output code and data files as either Intel .hex format, which can be used in an RTP simulator, or as Xilinx .coe and .xcp format, which are used as input files for the RTPGen Hardware Generator.

Chapter 6: The RTPGen Hardware Generator

Once compilation from C to RTP assembly instructions has been done, the system generator will have the necessary directives to begin architecting the system. Figure 6.1 shows the complete dataflow for C-to-FPGA compilation. The RTPGen Hardware Generator takes as inputs the assembly files and the generator directive file produced by the SDCC-RTP compiler, and ultimately outputs several VHDL and EDIF files. This process will be described in more detail throughout this section.

The ability to customize the system architecture for each different real-time application is exploited by having the compiler statically analyze the application code to determine:

- how many processors are required,
- the number of tasks,
- the static assignment of each task to a specific processor,
- the default task priority within each processor,
- what resources are needed by each task (what nodes are needed in the task-resource matrix),
- the amount of code and memory for each processor, and
- what I/O devices are used by each processor.

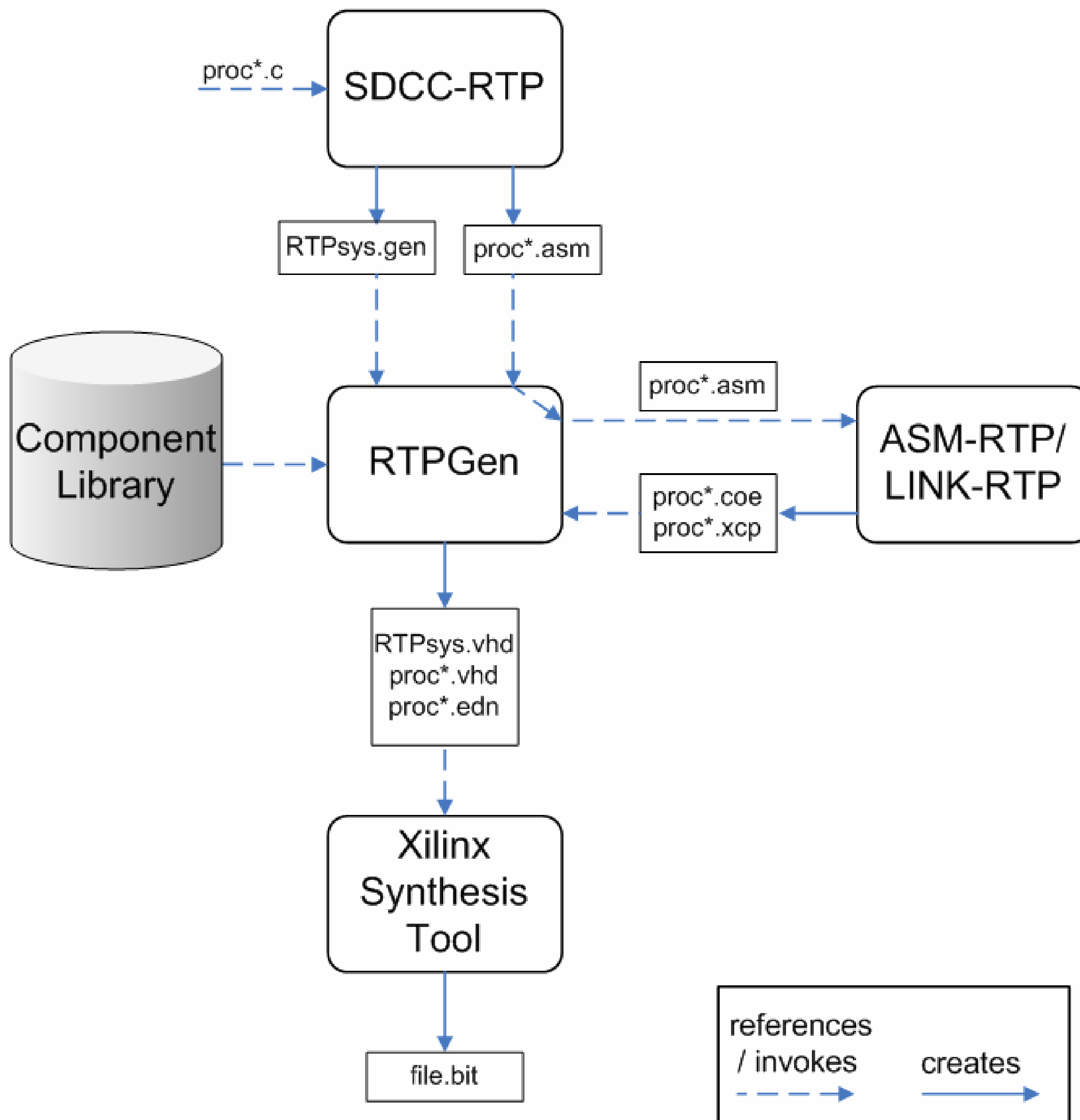

Figure 6.1: Dataflow for C-to-FPGA Compilation

Once the compiler (and in the case of code and memory size, the assembler) has extracted these parameters, the RTPGen hardware generator can proceed with architecting the system. After parsing the generator file, the RTPGen hardware generator passes all of the assembly files to the assembler and linker to produce one set of files per processor, which includes code and data memory files. The RTPGen hardware generator

takes the output of the linker and runs the .coe and .xcp files through the Xilinx coregen to generate the memory image files for each processor.

A complete basic set of the resource and device modules, as well as the RTP processor, has been compiled as a library of VHDL and EDIF entities. The top level VHDL file created by the RTPGen hardware generator calls the elements in the library and the elements created by the Xilinx coregen program to assemble the entire architecture. These entities are instantiated and integrated together in order to be synthesized specifically for the given application.

6.1 – Instantiation of Processors

The RTPGen hardware generator takes as its main input the RTPsys.gen generator directive file. This file is described in detail in Section 4.3. As the generator directive file is parsed, each device and/or processor is instantiated in memory. The assembly source files for each processor must be contained in a file named proc<proc_num>.asm. The data and instruction memory images for each processor, once assembled and linked, will be contained in files named proc<proc_num>.[data|code].<extension>. The M_ directive in RTPsys.gen will specify these filenames. When the M_ directive is parsed, the processor interface and declaration are instantiated, as well as the corresponding data and memory block rams and interfaces.

The RTP processor contains ports to interface with: data memory, instruction memory, the scheduler, I/O devices, resources, tasks, the Task Resource Matrix, and specific nodes in the Task Resource Matrix. The hardware implementation of the RTP processor, resources, devices, and the Task Resource Matrix is the subject of [10].

6.2 – Creation of Tasks, Resources, and Devices

When the RTPGen hardware generator processes a T_ directive, a task interface and declaration are instantiated, allocating the task to a specific processor, and assigning the task the specified priority. All of the tasks assigned to the same processor share the same code and data memory. A hardware scheduler determines which tasks will execute on which processor at a given time. Each task contains signals to interface with: a specific processor, the scheduler, and a resource node.

The R_ and D_ directives specify resources and devices that need to be instantiated. Resource modules are instantiated in a manner described in the following section that allows them to be shared between the tasks on all processors that need them. Resource modules interface with processors and resource nodes. Device modules are used for I/O, and are instantiated along the processor I/O bus of the specified processor only. They are not shared between processors. Devices interface with processors and off-chip inputs/outputs.

6.3 – RTP Simptris Example

To better illustrate the RTP C-to-FPGA system, this section will walk through an example embedded design. The basis of this example will be the BYU ECEn 425 Simptris Lab. Simptris is a tetris-like game, but with only two types of pieces: corner pieces and straight pieces. It uses an interrupt scheme to signal when new pieces arrive, pieces touch the bottom of the screen, lines are cleared, etc. For the purpose of this example what would have been an interrupt in Simptris will be signaled via semaphores.

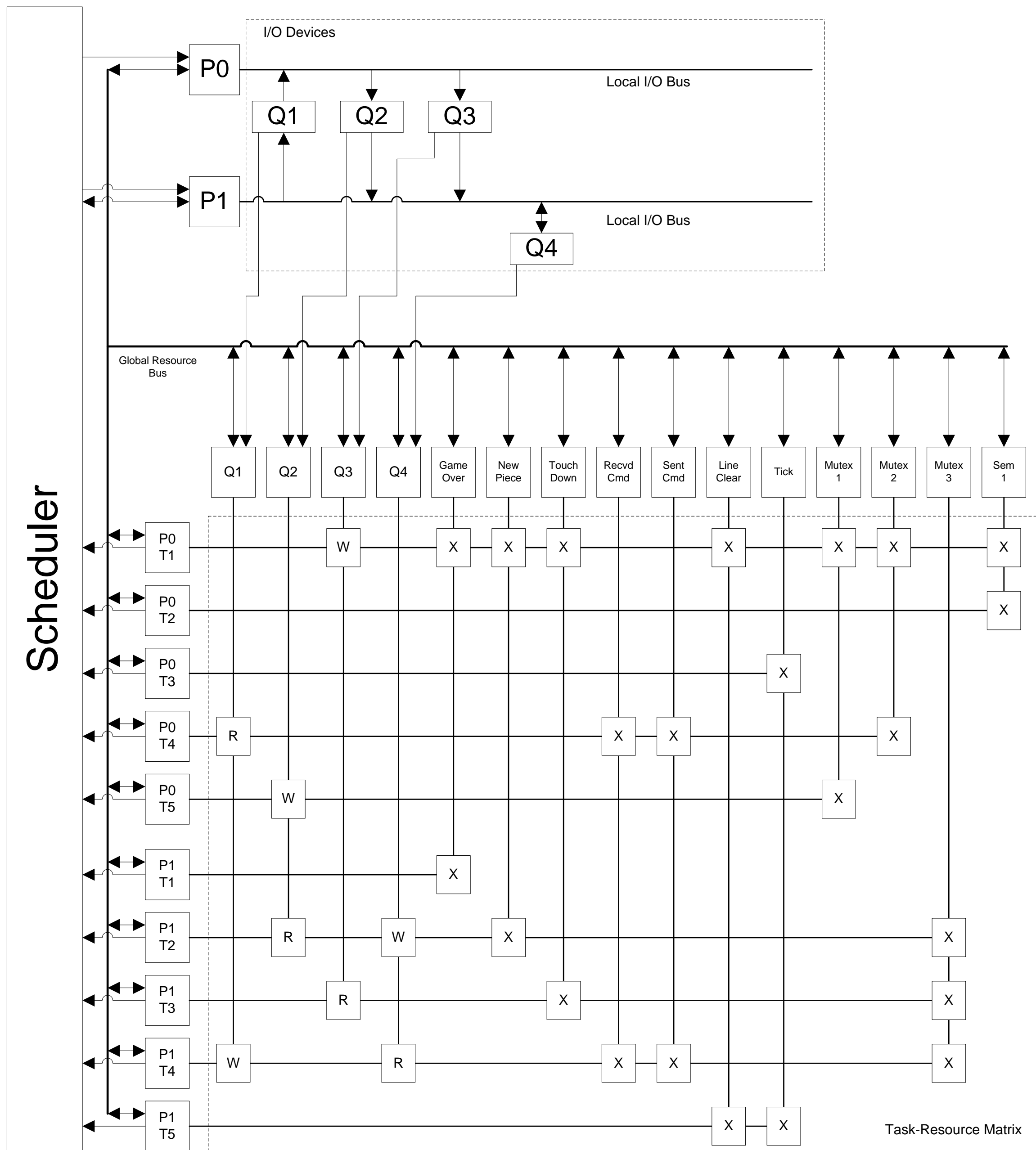

Figure 6.2: RTP System Diagram for Simptris Example

One processor (P0) will generate all of the interrupts (i.e. act as the game engine), while a second processor (P1) will service all of the interrupts (i.e. play the game).

For this example, the RTP system diagram can be seen in Figure 6.2. Code and data block RAMs are not shown. Hardware FIFOs are used to pass messages between

the two processors, and between two of the tasks on the second processor. Several mutex resources are used by both processors to protect shared memory locations. The boxes labeled “X” signify nodes in the TRM. For the sake of space in the diagram, the FIFO_READER and FIFO_WRITER resources have been combined into one entry in the figure, with the corresponding resource nodes being marked “R” or “W” to distinguish between the two. P0T0 and P1T0 are the idle tasks for each processor and are therefore not shown in the system diagram.

Resources labeled “Q1”, “Q2”, “Q3”, and “Q4” are all queue resources, connected to hardware FIFOs for message passing. The resources labeled “Game Over”, “New Piece”, “Touch Down”, “Recvd Cmd”, “Sent Cmd”, “Line Clear” and “Tick” are interrupts, implemented in this example as semaphores. “Mutex 1, “Mutex 2”, and “Mutex 3” are mutex resources used to protect shared memory, and “Sem 1” is a semaphore resource used in processor 0 as a means of stepping the game engine. A more detailed description of what each resources is used for is included in the description of each task below.

The global system header file, as seen in Figure 6.3 contains defines for all of the resource and device types and IDs. Each FIFO device has a single device ID, however both a FIFO_READER and FIFO_WRITER device type must be created for each FIFO in order to both read from and write to the queue.

```
enum resource_type {
  ZRESOURCE, MUTEX, DIS_EVENT, CON_EVENT,
  SEMAPHORE, TIMER, INTERRUPT, READER, WRITER };

enum device_type {
  ZDEVICE, FIFO_READER, FIFO_WRITER, TRANSMITTER, RECEIVER,
  SCRATCHMEM, FP_ADD, FP_MUL };

#pragma NUMPROCS_2

/* Resources */
/* Mutexes */
#define MUTEX1             1
#define MUTEX2             2
#define MUTEX3             3

/* Semaphores */
#define SEM1               5
#define SEM2               6

/* Readers / Writers */
#define QU1_R              7
#define QU1_W              8
#define QU2_R              9
#define QU2_W              10
#define QU3_R              11
#define QU3_W              12
#define QU4_R              14
#define QU4_W              15

/* Interrupts */
#define INT_GameOver       35
#define INT_NewPiece       36
#define INT_TouchDown      37
#define INT_RecvdCmd       38
#define INT_LineCleared    39
#define INT_Tick           40
#define INT_SentCmd        41

/* Devices */
/* FIFOS */
#define FIFO_QU1  1
#define FIFO_QU2  2
#define FIFO_QU3  3
#define FIFO_QU4  4
```

Figure 6.3: RTP Simptris global.h

The ten tasks shown in Figure 6.2 along with a brief description of what each task does and the other tasks with which it communicates are:

- P0T1: Core task. This task is responsible for generating the "Game Over", "New Piece", "Touch Down" and "Line Clear" interrupts. It communicates directly with P0T2 via "Sem 1" to know when the game should process the playing field, with P0T4 via "Mutex 2" for processing slide and rotate commands and with P0T5 via "Mutex 1" for adding new pieces to the playing field. Before signaling "Touch Down", the information regarding the piece that has touched down is placed in Q3.
- P0T2: Game Tick Task. This task communicates with P0T1 via "Sem 1" that the game should process any pending commands, and move each piece on the playing field downward.
- P0T3: Tick task. This task is used to generate the "Tick" interrupt, which is used by P1 to keep track of statistics.
- P0T4: Command task. This task blocks on the "Sent Cmd" interrupt. Once this interrupt has been signaled this task reads the command from Q1, which is populated by P1. "Mutex 1" is used to protect the memory where the pending commands are stored until they can be processed by P0T1. The "Recvd Cmd" interrupt is then generated after a small delay.
- P0T5: Piece task. This task generates new pieces for the playing field. As the number of lines cleared increases, the delay between new pieces decreases. This task communicates with P0T1 after each new piece is created. It then places the piece information in Q2 to later be consumed by P1.

```
/* Create Resources and Nodes */
/*P0 T1 */
_OpenResource(MUTEX, MUTEX1, 1);
_OpenResource(MUTEX, MUTEX2, 1);
_OpenResource(SEMAPHORE, Sem_GameTick, 1);
_OpenResource(WRITER, QU3_W, 1);
_OpenResource(SEMAPHORE, INT_NewPiece, 1);
_OpenResource(SEMAPHORE, INT_TouchDown, 1);
_OpenResource(SEMAPHORE, INT_LineCleared, 1);
_OpenResource(SEMAPHORE, INT_GameOver, 1);
/*P0 T2 */
_OpenResource(SEMAPHORE, SEM1, 2);
/*P0 T3 */
_OpenResource(SEMAPHORE, INT_Tick, 3);
/*P0 T4 */
_OpenResource(SEMAPHORE, SEM2, 4);
_OpenResource(SEMAPHORE, INT_RecvdCmd, 4);
_OpenResource(READER, QU1_R, 4);
_OpenResource(MUTEX, MUTEX2, 4);
/*P0 T5 */
_OpenResource(MUTEX, MUTEX1, 5);
_OpenResource(WRITER, QU2_W, 5);

/* Create Devices */
_OpenDevice(FIFO_READER, FIFO_QU1, 4, 0, 8, QU1_R, NULL);
_OpenDevice(FIFO_WRITER, FIFO_QU2, 5, 8, 8, QU2_W, NULL);
_OpenDevice(FIFO_WRITER, FIFO_QU3, 1, 16, 32, QU3_W, NULL);

/* Create Tasks */
task_error = _CreateTask((void *) Core_task, 1, 60, 0, 0);
if (task_error) return task_error;

task_error = _CreateTask((void *) GameTick_task, 2, 30, 0, 0);
if (task_error) return task_error;

task_error = _CreateTask((void *) Tick_task, 3, 30, 0, 0);
if (task_error) return task_error;

task_error = _CreateTask((void *) Command_task, 4, 30, 0, 0);
if (task_error) return task_error;

task_error = _CreateTask((void *) Piece_task, 5, 30, 0, 0);
if (task_error) return task_error;
```

Figure 6.4: RTP Simptris proc0.c

- P1T1: GameOver task. This task blocks on the "Game Over" interrupt. As this is the highest priority task on P1, the other tasks on P1 will be blocked until the system is reset once this interrupt is received.
- P1T2: NewPiece task. This task blocks on the "New Piece" interrupt. Upon receiving this interrupt the piece information is retrieved from Q2, and the commands to be performed on the piece are communicated to P1T4 via Q4. "Mutex 3" is used to protect the shared memory region in P1 for tracking which pieces are still in play.
- P1T3: TouchDown task. This task blocks on the "Touch Down" interrupt. Once received, it retrieves the information regarding which piece is out of play from Q3, and then blocks on "Mutex 3" to update P1's piece tracking variables.
- P1T4: Command task. This task blocks on the "Recvd Cmd" interrupt. When signaled, it blocks on Q4 until a new command is ready to be sent to P0. To determine if the piece is still in play, "Mutex 3" is used to ensure that the most current piece information is available. If the piece is not in play, the task blocks again on Q4 until a new command is ready. Once a command is received for a piece in play, the command is placed in Q1 and the "Sent Cmd" interrupt is used to signal P0 that a new command is ready.
- P1T5: Stats task. This task keeps track of game statistics, such as the number of lines cleared and the number of game ticks signaled. It uses non-blocking resource acquisition to allow it to process both "Line Clear" and "Tick" interrupts.

```
/* Create Resources and Nodes */
/*P1 T1 */
_OpenResource(SEMAPHORE, INT_GameOver, 1);
/*P1 T2 */
_OpenResource(MUTEX, MUTEX3, 2);
_OpenResource(SEMAPHORE, INT_NewPiece, 2);
_OpenResource(READER, QU2_R, 2);
_OpenResource(WRITER, QU4_W, 2);
/*P1 T3 */
_OpenResource(MUTEX, MUTDX3, 3);
_OpenResource(READER, QU3_R, 3);
_OpenResource(SEMAPHORE, INT_TouchDown, 3);
/*P1 T4 */
_OpenResource(MUTEX, MUTEX3, 4);
_OpenResource(SEMAPHORE, SEM2, 4);
_OpenResource(SEMAPHORE, INT_RecvdCmd, 4);
_OpenResource(READER, QU4_R, 4);
_OpenResource(WRITER, QU1_W, 4);
/*P1 T5 */
_OpenResource(SEMAPHORE, INT_Tick, 5);
_OpenResource(SEMAPHORE, INT_LineCleared, 5);

/* Create Devices */
_OpenDevice(FIFO_WRITER, FIFO_QU1, 4, 0, 8, QU1_W, NULL);
_OpenDevice(FIFO_READER, FIFO_QU2, 2, 8, 8, QU2_R, NULL);
_OpenDevice(FIFO_READER, FIFO_QU3, 3, 16, 32, QU3_R, NULL);
_OpenDevice(FIFO_WRITER, FIFO_QU4, 4, 58, 8, QU4_R, NULL);
_OpenDevice(FIFO_READER, FIFO_QU4, 2, 58, 8, QU4_W, NULL);

/* Create Tasks */
task_error = _CreateTask((void *) GameOver_task, 1, 12, 0, 0);
if (task_error) return task_error;

task_error = _CreateTask((void *) NewPiece_task, 2, 12, 0, 0);
if (task_error) return task_error;

task_error = _CreateTask((void *) TouchDown_task, 3, 12,0, 0);
if (task_error) return task_error;

task_error = _CreateTask((void *) Command_task, 4, 12, 0, 0);
if (task_error) return task_error;

task_error = _CreateTask((void *) Stats_task, 5, 0, 0, 0);
if (task_error) return task_error;
```

Figure 6.5: RTP Simptris proc1.c

```
P_2
M_0 :4096 :proc0.code.hex :2048 :proc0.data.hex
T_1 :0 :1
T_2 :0 :2
T_3 :0 :3
T_4 :0 :4
T_5 :0 :5
R_1 :1 :0 :1
R_2 :1 :0 :1
R_5 :4 :0 :1
R_12 :8 :0 :1
R_36 :4 :0 :1
R_37 :4 :0 :1
R_39 :4 :0 :1
R_35 :4 :0 :1
R_5 :4 :0 :2
R_40 :4 :0 :3
R_6 :4 :0 :4
R_38 :4 :0 :4
R_7 :7 :0 :4
R_2 :1 :0 :4
R_1 :1 :0 :5
R_10 :8 :0 :5
D_1 :1 :0 :4 :0 :8 :7 :0
D_2 :2 :0 :5 :8 :8 :10 :0
D_3 :2 :0 :1 :16 :32 :12 :0
M_1 :4096 :proc1.code.hex :2048 :proc1.data.hex
T_1 :1 :1
T_2 :1 :2
T_3 :1 :3
T_4 :1 :4
T_5 :1 :5
R_35 :4 :1 :1
R_3 :1 :1 :2
R_36 :4 :1 :2
R_9 :7 :1 :2
R_15 :8 :1 :2
R_3 :1 :1 :3
R_11 :7 :1 :3
R_37 :4 :1 :3
R_3 :1 :1 :4
R_6 :4 :1 :4
R_38 :4 :1 :4
R_14 :7 :1 :4
R_8 :8 :1 :4
R_40 :4 :1 :5
R_39 :4 :1 :5
D_1 :2 :1 :4 :0 :8 :8 :0
D_2 :1 :1 :2 :8 :8 :9 :0
D_3 :1 :1 :3 :16 :32 :11 :0
D_4 :2 :1 :4 :58 :8 :14 :0
D_4 :1 :1 :2 :58 :8 :15 :0
```

Figure 6.6: RTP Simptris RTPsys.gen

After compiling and assembling the code for both processors, the RTPsys.gen file contains the lines listed in Figure 6.6. With this input file, the RTPGen hardware generator can create the top-level VHDL file for the system. A portion of the VHDL file created by the system generator can be seen in Figures 6.7, 6.8, and 6.9.

```
---------------------------------------------------------------------------
-- processor interface  Processor 0
---------------------------------------------------------------------------
-- scheduler interface
signal p0_tid     : std_logic_vector(3 downto 0)
-- instruction memory interface
signal p0_iaddr   : std_logic_vector(11 downto 0);
signal p0_iout    : std_logic_vector(17 downto 0);
signal p0_ien     : std_logic;
-- data memory interface
signal p0_daddr   : std_logic_vector(14 downto 0);
signal p0_din     : std_logic_vector(15 downto 0);
signal p0_dout    : std_logic_vector(15 downto 0);
signal p0_we      : std_logic_vector(1 downto 0);
-- i/o device interface
signal p0_xid     : std_logic_vector(7 downto 0);
signal p0_xrd     : std_logic;
signal p0_xwr     : std_logic;
signal p0_xin     : std_logic_vector(15 downto 0);
signal p0_xout    : std_logic_vector(15 downto 0);
-- resource-matrix interface
signal p0_rid     : std_logic_vector(7 downto 0);
signal p0_req     : std_logic;
signal p0_gnt     : std_logic;
signal p0_min     : std_logic_vector(15 downto 0);
signal p0_mout    : std_logic_vector(15 downto 0);
-- resource commands
signal p0_enable  : std_logic;
signal p0_disable : std_logic;
signal p0_sig1    : std_logic;
signal p0_sig2    : std_logic;
signal p0_read    : std_logic;
signal p0_write   : std_logic;
signal p0_release : std_logic;
signal p0_reset   : std_logic;
signal p0_lock    : std_logic;
signal p0_nb_lock : std_logic;
-- task commands
signal p0_ptid    : std_logic_vector(7 downto 0);
signal p0_rd_time : std_logic;
signal p0_wr_time : std_logic;
signal p0_rd_prio : std_logic;
signal p0_wr_prio : std_logic;
---------------------------------------------------------------------------
-- task interface : Task 1
---------------------------------------------------------------------------
signal p0t1_mout     : std_logic_vector(15 downto 0);
signal p0t1_priority : std_logic_vector(3 downto 0);
signal p0t1_ready    : std_logic;
signal p0t1_sel      : std_logic;
signal p0t1_readyin  : std_logic;
signal p0t1_grantin  : std_logic;
```

Figure 6.7: RTP Simptris VHDL Declarations for Processor 0

Figure 6.7 shows the signal declarations for processor 0 and one of its tasks. Figure 6.8 shows the instantiation of processor 0. Finally, Figure 6.9 shows the instantiation of processor 0 task 1, and the scheduler for processor 0.

```
---------------------------------------------------------------------------
-- processor 0 (5 tasks)
---------------------------------------------------------------------------
-- processor
P0 : processor_sch
generic map (
  NTASKS => X"5",
  PROC_ID => X"0"
)
port map (
  -- globals
  clk                => clk,
  reset              => rst,
  -- data memory
  DataMemInput       => p0_dout,
  DataMemOut         => p0_din,
  DataMemAddressOut  => p0_daddr,
  DataMemWE          => p0_we,
  -- inst memory
  PCimem             => p0_iaddr,
  ImemOutIR          => p0_iout,
  InstMemEnable      => p0_ien,
  -- device interface
  IOinput            => p0_xin,
  IODataOut          => p0_xout,
  IOAddressOUT       => p0_xid,
  EnableIOOut        => p0_xrd,
  IOwe               => p0_xwr,
  -- scheduler interface
  SchedulerTaskIn    => p0_tid,
  -- task interface
  TaskIdAddress      => p0_ptid,
  TRM_read_priority  => p0_rd_prio,
  TRM_read_time      => p0_rd_time,
  TRM_write_priority => p0_wr_prio,
  TRM_write_time     => p0_wr_time,
  --resource interface
  TRM_request        => p0_req,
  ResourceGranted    => p0_gnt,
  ResourceAddressOut => p0_rid,
  TRMDataOut         => p0_mout,
  TaskResourceMatrixInput => p0_min,
  TRM_disable        => p0_disable,
  TRM_enable         => p0_enable,
  TRM_lock           => p0_lock,
  TRM_nb_lock        => p0_nb_lock,
  TRM_release        => p0_release,
  TRM_read           => p0_read,
  TRM_write          => p0_write,
  TRM_reset          => p0_reset,
  TRM_sig1           => p0_sig1,
  TRM_sig2           => p0_sig2
);
```

Figure 6.8: RTP Simptris VHDL Instantiation of Processor 0

```
------------------------------------------------------------------------------
-- task 1
------------------------------------------------------------------------------
P0t1 : task
generic map (
  PROC  => X"0",
  TASK  => X"1"
)
port map (
  -- globals
  clk        => clk,
  msec_tic   => msec_tic,
  rst        => rst,
  PPPP       => p0_ptid(7 downto 4),
  TTTT       => p0_ptid(3 downto 0),
  data_in    => p0_mout(15 downto 0),
  data_out   => p0t1_mout(15 downto 0),
  -- commands from processor
  read_time  => p0_rd_time,
  write_time => p0_wr_time,
  read_prio  => p0_rd_prio,
  write_prio => p0_wr_prio,
  nb_lock    => p0_nb_lock,
  -- interface to task scheduler
  priority   => p0t1_priority,
  ready_out  => p0t1_ready,
  -- interface to task-resource nodes
  ready_in   => p0t1_readyin,
  sel_task   => p0t1_sel,
  grant_in   => p0t1_grantin
);

------------------------------------------------------------------------------
-- scheduler for processor 0
------------------------------------------------------------------------------
P0_sc : scheduler
generic map (
  PROC_ID   => X"0",
  NUM_TASKS => X"5"
)
port map (
  Priority1 => p0t1_priority,
  Ready1    => p0t1_ready,
  Priority2 => p0t2_priority,
  Ready2    => p0t2_ready,
  Priority3 => p0t3_priority,
  Ready3    => p0t3_ready,
  Priority4 => p0t4_priority,
  Ready4    => p0t4_ready,
  Priority5 => p0t5_priority,
  Ready5    => p0t5_ready,
  Priority6 => gnd4,
  Ready6    => gnd,
<through Task15>
  -- ReadyTask ID
  ReadyTaskID => p0_tid
);
```

Figure 6.9: RTP Simptris VHDL Instantiation of P0 Task 1 and Scheduler

In summary, the system generator uses the source files global.h, proc0.c and proc1.c to produce the top-level structural VHDL file for a custom architecture. This

custom architecture contains exactly the hardware necessary to execute the system provided in this example.

Chapter 7: Conclusions and Future Work

The SDCC-RTP C to FPGA design flow enables rapid development of System-on-Programmable-Chip designs. It is a self-contained set of development tools that allows the design engineer to provide C source code and a global.h file that specify the required number of processors, tasks, and resources, and then utilizes this information to build the top-level structural VHDL file. Targeting FPGAs allows the designer to quickly and cost-efficiently develop a multi-processor real-time embedded SoPC.

7.1 – Contributions Revisited

The introduction of this thesis detailed six contributions that were made. These contributions were defining the RTP system-level architecture, porting of the SDCC compiler to the RTP architecture, creation of a set of peephole optimizations for the RTP architecture to reduce code size, extending the SDCC compiler to interface with the RTPGen hardware generator, development of the RTPGen hardware generator, and a document that can be used for porting the SDCC compiler to new architectures.

The RTP system architecture is described in Chapter 2. The details of developing the SDCC-RTP code generator are contained in Chapters 3 and 4. The details of the RTP assembler are found in Chapter 5. Code generation and assembly are necessary to allow the RTP processor to execute code.

The set of peephole optimization rules for the RTP architecture can be found in Appendix A. These rules help to minimize the required code size for the generated assembly instructions from the compiler. A discussion of peephole optimization rules is found in Section 3.4.4.

The RTPGen hardware generator can take the code generated by the SDCC-RTP compiler coupled with the directive file and create an optimized SoPC design. This process is described in Chapter 6, including an example real-time system to better illustrate the RTP system architecture.

A helpful document that can be used to help others port the SDCC compiler to new architectures is found in Chapter 3, which is also intended to be published as a self-contained document. Chapter 3 can also be used as an additional reference for a course on compilers and code generation.

7.2 – Future Work

Compiling code optimally is an NP-complete problem. There are many factors that play into optimal code generation, such as optimal register allocation and assignment, or optimal code sequencing based on future operations (compiler look-ahead depth). Any of these areas could be explored in greater detail.

One optimization that would significantly reduce code size would be to initialize the global data upon assembly declaration. This would eliminate the need to use code space to perform the initializations.

Another area that can potentially result in a significant reduction of code size is code compression with hardware support for decompression [13,14]. This methodology would require both hardware and compiler support.

7.3 – Final Words

The primary motivation of this thesis is the belief that FPGA advances in size and speed continue to make compelling arguments for development of customized application-specific processing elements. I believe I have provided the initial framework for developers to design such a system, and have the sincere hope that others will continue to improve upon my work.

BIBLIOGRAPHY

[1] V. Mooney and D. Blough, "A Hardware-Software Real-Time Operating System Framework for SoCs," *IEEE Design & Test of Computers*, pp. 44-51, Nov-Dec 2002.

[2] V. Mooney, "Hardware/Software Partitioning of Operating Systems [SoC Applications]," *Proc. of the Design, Automation and Test in Europe Conference and Exhibition*, pp. 338-339, 2003.

[3] M. Boden, J. Schneider, K. Feske, and S. Rulke, "Enhanced Reusability for SoC-based HW/SW Co-design," *Proc. of the Euromicro Symposium on Digital System Design*, pp 94-99, Sep. 4-6, 2002.

[4] Xilinx, http://www.xilinx.com

[5] M.B. Gokhale, J.M. Stone, J. Arnold, and M. Kalinowski, "Stream-oriented FPGA computing in the Streams-C high level language," *IEEE International Symposium on FPGAs for Custom Computing Machines,* 2000.

[6] J. Frigo, M. Gokhale, D. Lavenier, "Evaluation of the Streams-C C-to-FPGA Compiler: An Applications Perspective," *Proceedings of the Ninth ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA),* pp. 134-140, Monterey, CA, February 2001.

[7] Celoxica, http://www.celoxica.com

[8] S. Dutta, "SDCC Compiler User Guide", http://sdcc.sourceforge.net

[9] A. Aho, R. Sethi, J.D. Ullman, *Compilers: Principles, Techniques, and Tools*, pp. 541-542, Addison-Wesley, 1986.

[10] S. Isaacson, "Unpublished", M.S. Thesis, BYU.

[11] S. Isaacson, D. Wilde, "The Task-Resource Matrix: Control for a Distributed Reconfigurable Multi-Processor Hardware RTOS", *ERSA '04,* July 2004.

[12] W.M. McKeeman, "Peephole Optimization", *Communications of the ACM*, v.8 n.7, pp. 443-444, July 1965

[13] K.D. Cooper, N. McIntosh, "Enhanced Code Compression for embedded RISC processors", *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pp.139-149, May 1999

[14] A. Wolfe and A. Chanin, "Executing Compressed Programs on an Embedded RISC Architecture", *Proceedings the 25th Annual International Symposium on Microarchitecture*, MICRO'25, pp. 81-91, December 1992

[15] D. Sun, D. Blough and V. Mooney, "Atalanta: A New Multiprocessor RTOS Kernel for System-on-a-Chip Applications.", Technical Report GIT-CC-02-19, http://www.cc.gatech.edu/content/view/1041/, 2002.

[16] J. Labrosse, *MicroC/OS: Real-Time Kernel II: The Real-Time Kernel*, R&D Books, Lawrence, KS, October 1998.

[17] M. Young, "Unpublished", M.S. Thesis, BYU.

[18] A. Agarwal, J. Kubiatowicz, D. Kranz, B. Lim, D. Yeung, G. D'Souza, and M. Parkin, "Sparcle: An Evolutionary Processor Design for Large-Scale Multiprocessors," *IEEE Micro*, Vol. 13 Issue 3, pp. 48-61, June 1993

[19] T. Nakano, A. Utama, M. Itabashi, A. Shiomi, and M. Imai, "Hardware Implementation of a Real-time Operating System," *Proceedings of TRON'95*, IEEE, pp. 34-42, 1995.

[20] J. Ito, T. Nakano, Y. Takeuchi, and M. Imai, "Effectiveness of a High Speed Context Switching Method Using Register Bank," *IEICE Trans. Fundamentals*, Vol. E81-A, No.12, pp. 2661-2667, Dec. 1998.

[21] P.H. Shiu, Y. Tan, and V.J. Mooney III, "A Novel Parallel Deadlock Detection Algorithm and Architecture," *Proc. Int'l Symp. Hardware/Software Codesign*, ACM Press, New York, 2001, pp. 30-36

Appendix A: Peephole Rules for the SDCC-RTP Compiler

```
replace restart {
  pop       %1
  push      %1
} by {
  ; Peephole 1     removed pop %1 push %1
}

replace restart {
  pop       %1
  mov       %2,%3
  push      %1
} by {
        ; Peephole 2     removed pop %1 push %1
  mov       %2,%3
}

replace restart {
  add %2, %1
  mov %1, %2
} by {
  ; Peephole 3    removed mov %1, %2
  add %1, %2
}

replace restart {
  sub %1,%2
  cmp %1,0
  be        %3
} by {
  sub %1,%2
  ; Peephole 4    removed cmp %1, 0
  be        %3
}

replace restart {
  push      %1
  pop %1
} by {
  ; Peephole 5    removed push %1 pop %1
}


replace restart {
  mov       r1, %1
```

```
  mov       %2, r1
} by {
   ; Peephole 6   removed redundant mov r1, %1
  mov %2, %1
}

replace restart {
  mov       r0, %1
  mov       %2, r0
} by {
  ; Peephole 7    removed redundant mov r0, %1
  mov %2, %1
}

replace restart {
  mov       r0, %1
  add       %2, r0
} by {
  ; Peephole 8a   removed mov r0, %1 (add)
  add %2, %1
}

replace restart {
  mov       r0, %1
  addc      %2, r0
} by {
  ; Peephole 8b   removed mov r0, %1 (addc)
  addc      %2, %1
}

replace restart {
  mov r0,%1
  sub %2,r0
} by {
  ; Peephole 9a removed mov r0 %1 (sub)
  sub %2, %1
}

replace restart {
  mov r0,%1
  subc      %2,r0
} by {
  ; Peephole 9b removed mov r0 %1 (subc)
  subc      %2, %1
}

replace restart {
  mov r0,%1
  cmp %2,r0
} by {
  ; Peephole 10a removed mov r0 %1 (cmp)
  cmp %2, %1
}

replace restart {
  mov r0,%1
  cmpc      %2,r0
```

```
} by {
  ; Peephole 10b removed mov r0 %1 (cmpc)
  cmpc      %2, %1
}

replace restart {
  mov r0,%1
  and %2,r0
} by {
  ; Peephole 11a removed mov r0 %1 (and)
  and %2, %1
}

replace restart {
  mov r0,%1
  andc      %2,r0
} by {
  ; Peephole 11b removed mov r0 %1 (andc)
  andc      %2, %1
}

replace restart {
  mov r0,%1
  or        %2,r0
} by {
  ; Peephole 12a removed mov r0 %1 (or)
  or        %2, %1
}

replace restart {
  mov r0,%1
  orc %2,r0
} by {
  ; Peephole 12b removed mov r0 %1 (orc)
  orc %2, %1
}

replace restart {
  mov r0,%1
  xor %2,r0
} by {
  ; Peephole 13a removed mov r0 %1 (xor)
  xor %2, %1
}

replace restart {
  mov r0,%1
  xorc      %2,r0
} by {
  ; Peephole 13b removed mov r0 %1 (xorc)
  xorc      %2, %1
}

replace restart {
  mov r0,%1
  cmp %2,r0
} by {
```

```
  ; Peephole 14a removed mov r0 %1
  cmp %2, %1
}

replace restart {
  mov r0,%1
  cmpc      %2,r0
} by {
  ; Peephole 14b removed mov r0 %1
  cmpc      %2, %1
}

replace restart {
  ld        r0, %1(%2)
  mov %3,r0
} by {
  ; Peephole 15 removed mov %2, r0
  ld        %3, %1(%2)
}
```

Appendix B: Assembly for RTP Code Generation Example

```
;--------------------------------------------------------
; File Created by SDCC : FreeWare ANSI-C Compiler
; Version Real Time Processor 1.0Beta Mon Jan 30 22:07:11 2006

;--------------------------------------------------------
   .module example1

;--------------------------------------------------------
; Public variables in this module
;--------------------------------------------------------
   .globl _main
   .globl _doubleit
   .globl _c
;--------------------------------------------------------
;  ram data
;--------------------------------------------------------
   .area DSEG     ;(DATA)
_c::
   .ds 1
;--------------------------------------------------------
; overlayable items in  ram
;--------------------------------------------------------
   .area OSEG     ;((UNUSED in RTP) OVR,DATA)
;--------------------------------------------------------
; external initialized ram data
;--------------------------------------------------------
;--------------------------------------------------------
; global & static initialisations
;--------------------------------------------------------
   .area GSFINAL  ;(CODE)
.globl __sdcc_init_data
__sdcc_init_data:
   ret      ;return to caller
;--------------------------------------------------------
; code
;--------------------------------------------------------
   .area CSEG     ;(CODE)
;------------------------------------------------------------
;Allocation info for local variables in function 'doubleit'
;------------------------------------------------------------
;op                       Allocated to registers r9
;------------------------------------------------------------
;  example1.c 3
;  -----------------------------------------
```

```
;   function doubleit
;  ----------------------------------------
_doubleit:
   push     r14
   mov      r14,r13
;  example1.c 4
   sll      r9,0x1
_ret_doubleit:
   pop      r14
   ret
;------------------------------------------------------------
;Allocation info for local variables in function 'main'
;------------------------------------------------------------
;a                          Allocated to registers r4
;b                          Allocated to registers
;i                          Allocated to registers
;result                     Allocated to registers r5
;------------------------------------------------------------
;  example1.c 7
;  ----------------------------------------
;   function main
;  ----------------------------------------
_main:
   push     r4
   push     r5
   push     r6
   push     r14
   mov      r14,r13
   push     r15
;  example1.c 11
; Peephole 7      removed redundant mov r0, 0x1
   mov      r4, 0x1
;  example1.c 13
   mov      r0,0x71
   mova     r1, _c
   st r0,0(r1)
;  example1.c 14
; Peephole 7      removed redundant mov r0, 0x0
   mov      r5, 0x0
;  example1.c 16
; Peephole 7      removed redundant mov r0, 0xa
   mov      r6, 0xa
L00016:
;  example1.c 17
   mov      r0,0x2
   mul      r4,r0
;  example1.c 16
; Peephole 9a removed mov r0 0x1 (sub)
   sub      r6, 0x1
; Peephole 4    removed cmp r6, 0
   be L00027
   jmp      L00016
L00027:
;  example1.c 19
L00003:
   mova     r0, _c
   ld r1, 0(r0)
```

```
; Peephole 15    removed mov r1, r0
; Peephole 10a removed mov r0 0x6d (cmp)
   cmp      r1, 0x6d
   be L00005
;  example1.c 20
   mova     r2, _c
   ld r1, 0(r2)
; Peephole 9a removed mov r0 0x1 (sub)
   sub      r1, 0x1
   st r1, 0(r2)
   jump     L00003
L00005:
;  example1.c 22
   mova     r0,0x400
   cmp      r4,r0
   bge      L00007
;  example1.c 23
; Peephole 7      removed redundant mov r0, 0x1
   mov      r5, 0x1
L00007:
;  example1.c 25
   mova     r0, _c
   ld r1, 0(r0)
; Peephole 15    removed mov r1, r0
; Peephole 10a removed mov r0 0x6d (cmp)
   cmp      r1, 0x6d
   be L00008
   mova     r0, _c
   ld r1, 0(r0)
; Peephole 15    removed mov r1, r0
; Peephole 10a removed mov r0 0x71 (cmp)
   cmp      r1, 0x71
   be L00009
   jump     L00010
;  example1.c 27
L00008:
   push     r14
   push     r0
   push     r1
   push     r2
   push     r3
   mov      r9,r4
   call     _doubleit
   mov      r6,r9
   pop      r3
   pop      r2
   pop      r1
   pop      r0
   pop      r14
   mov      r4,r6
;  example1.c 28
   jump     L00011
;  example1.c 30
L00009:
; Peephole 7      removed redundant mov r0, 0x2
   mov      r5, 0x2
;  example1.c 31
```

```
   jump     L00011
;  example1.c 33
L00010:
; Peephole 7      removed redundant mov r0, 0x3
   mov      r5, 0x3
;  example1.c 34
L00011:
;  example1.c 35
   mova     r0,0x800
   cmp      r4,r0
   be L00013
;  example1.c 36
; Peephole 7      removed redundant mov r0, 0x4
   mov      r5, 0x4
L00013:
;  example1.c 38
   mov      r9,r5
_ret_main:
   pop      r15
   pop      r14
   pop      r6
   pop      r5
   pop      r4
   ret
   .area CSEG     ;(CODE)
```

Appendix C: RTP RTOS Assembly Source Code

```
	.title Real-Time Processor Operating System
	.subtitle composed by Matt Young, the Programmator
	.module rtos
	;; NOTE - moved timeout value to r10 in re seek from r12, adjust
not done yet
	; compiled with command: "as-rtp -xlos rtos.asm"

	; resource constants
	suspend = 1
	scheduler = 0
	resourceRMW = 2
	BLOCK = 3
	NONBLOCK = 4

	; return codes
	ERROR_CODE = 1
	AOK_CODE = 0

	;; TASK FLAGS
	READY_FLAG = 0x10

	.area GSINIT

	.org 0

	and R3, 0		; clear unused registers
	and R4, 0
	and R5, 0
	and R6, 0
	and R7, 0
	and R8, 0
	and R11, 0
	and R12, 0

	.globl __sdcc_init_data

	;; the very first thing to do is call __sdcc_init_data if we are
task 0

	r_prio	R0	; PPPP TTTT FFFF XXXX	XXXX=Priority FFFF =
Flags PPPP = processor TTTT = Task
	srl	R0, 12
	and	R0, 0xF		; R0 contains TTTT
	cmp	R0, 0
	bne	_nott0
	mova	R15, _nott0
	mova	R0, __sdcc_init_data
	jre	0(R0)
```

```
_nott0: r_prio    R0    ; PPPP TTTT FFFF XXXX   XXXX=Priority FFFF =
Flags PPPP = processor TTTT = Task
      mov   R1, R0
      srl   R1, 12
      and   R1, 0xF           ; PPPP
      mova  R2, my_proc_id
      st    R1, 0(R2)   ; store PPPP in my_proc_id


      ; store PPPP in proc_id
      srl   R0, 8
      and   R0, 0xF           ; TTTT in R0
      sll   R0, 3 ; 8 * TTTT(8 = sizeof task table element
      mova  R1, tasktable
      add   R0, R1      ; &tasktable[TTTT] in DSEG

      ld    R1, 2(R0)         ; R1 = current stack size
      mova  R2, current_stack ; pointer to current top of stack
      ld    R14, 0(R2)        ; init R14 = SP
      rsub  R1, R14           ; R1 <- R14 - R1 = new top of stack
      st    R1, 0(R2)   ; save new top of stack

      ld    R9,  4(R0)  ; arg1
      ld    R10, 6(R0)  ; arg2
      mov   R13, R14    ; init FP
      ld    R1,  0(r0)  ; task procedure address

      mova  R15, _gsi1  ; but since call R doesnt exist, simulate it
      jre   0(R1)       ; using 3 instructions: mova(2) and jre(1)


_gsi1:      call  _task_sus         ; call RTOS

forever:jump      forever

      ; end of GSINIT segment

      .area CSEG

      ;; regarding queues and mailboxes
      ;; 0 offset address of queue or mailbox with empty and full flags
      ;; 1 offset on gives us the message
      ;; so, will the empty and full flags go directly into the task
resource mat
      ;; in other words, do we even need OS to check empty and full
flags?
      ;; all right, more here
      ;; when i am writing to a queue, i have already obtained the write
mutex
      ;; however, when the queue become full, i attempt to lock on that
      ;; same mutex to block until the queue is non full
      ;; Does this even work? Is the write mutex lost when the queue
fills?

      ;; and are queues LIFOs, FIFOs? what? ie when they fill where do i
write
      ;; to once it isnt full anymore?

      ; external address


      ;;  for nb_lock, r_pri, w_pri
      ;; task id (8 bits), status bits (4 bits), priority (4 bits)
```

```
      ;; R0-R3  scratch registers
      ;; R4-R8 preserved
      ;; R9-R12 parameter registers/return values
      ;; R13 frame pointer
      ;; R14 stack pointer
      ;; R15 return address

;; TASKS

; rtp_task_sleep
;;; task_slp (sleep_time)
.globl _task_slp
_task_slp:  w_time            R9    ;puts us to sleep for R9 seconds
            r_prio            R1;  read task priority
            and         R1,0xEF;  reset ready flag
            w_prio            R1;  suspend until timeout expires
            or          R1, READY_FLAG;  set ready flag
            w_prio            R1;  task is now ready again
            rel         suspend ;  clear request of suspend
            jre         R15   ;return


; rtp_task_suspend - suspends forever
;;task_sus()
.globl _task_sus
_task_sus:  mov         R0, 0x0000
            w_time            R0    ;write a 0 to timeout count, no
expire
            r_prio            R1
            and         R1, 0xEF;  clear out the ready bit
            w_prio            R1;  set the task to non-ready, tasks
suspends
      ;; ok, this code segment only reached if another task unsuspends
us
            jre         0(R15) ; only get here if suspend resource
                               ;; is reset by another task

;rtp_task_reference - one instruction , probably doesnt need own
function
;; task_ref()
.globl _task_ref
_task_ref:  r_prio      R9    ;reads task reference info into R1
            jre         R15

.globl _lock
;rtp_lock - disables preemption for processor - are we gonna do this
_lock:      ds          ;; disable scheduler
            ret
;;may not need this


;; This function could better be built into the scheduler.
;; A flag in the scheduler could disable scheduling.
;; Maybe the scheduler itself could be addressed as a resource?
;; disable scheduler  ; allow the running task to run until it blocks
;; When the running task blocks, this flag clears itself
;; anything requesting the TRM will automatically re-enable the
scheduler

.globl _unlock
;rtp_unlock - reenables preemption
_unlock:    es          ;; enable scheduler
            ret
```

```
;;may not need this


;; RESOURCES
.globl _re_sig
;rtp_resource_signal - releases a resource
;;res_sig(resource_id)
_re_sig:    rel         R9    ;releases resource passed in in R9
            jre         0(R15)      ;return

;rtp_resource_gain- non-blocking lock on resource, event, etc
;; using R9 to pass back a parameter here
;;res_gain(resource_id)
.globl _re_gain
_re_gain:   nb_lock           R1, R9      ;can we acquire the resource
            r_pri       R0    ; read our task state
            srl         R1, 8 ;  move task ids to lower
            srl         R0, 8 ; byte to enable compare
            cmp         R1, R0      ;comparison to check if we got it
            mov         R9, AOK_CODE
            be          GOT   ;return to post routine if we got lock
            mov         R9, ERROR_CODE ;move error code into R1
GOT:        jre         0(R15)      ;return




;;rtp_pi_seek(resource_id, time_out_val)
.globl _re_pi_seek
_re_pi_seek:      nb_lock           R1, R9      ;can we acquire the
resource
            push        R1 ;  store res info on stack
            r_pri       R0    ; read our task state
            srl         R0, 8 ; move task ids into
            srl         R0, 8 ; lower byte for comp
            cmp         R0, R1      ;comparison to check if we got it
            be          END_PI      ;no need to PI if we got it
            r_pri       R0    ; read our task state
            load        R1, 0(R14) ; restore res owner state
            and         R1, 0xF     ;  mask out everything but owner pri
            and         R0, 0xF     ; mask out everything but my
priority
            cmp         R1, R0
            bgtu        _re_seek; branch and never return,owner has pri
            ld          R1, 0(R14); restore R1 (resource owning task)
            mova        R3, 0xFFF0
            and         R1, R3
            or          R1, R0; put our priority in resource own task
            w_prio            R1 ; bump up pri of res owning task
            push        R15 ;  we will kill our return address
            call        _re_seek
            ld          R15, 0(R14)
            ld          R1, 2(R14) ; instead of two pops
            add         R14, 0x04 ; we have now saved an inst
            w_prio            R1;  write the orig prior field still
      ;; stored in R1 (from read) back into task R1 (R1 is unmodified)
END_PI:           return


;rtp_resource_seek - blocking lock on resource
;; res_seek(resource _id,time_out_val)
.globl _re_seek
_re_seek:   w_tim       R10   ; r10 has time out value
```

```
            lock        R9    ;blocking lock on resource
            r_pri       R0    ;check ready flag to see if time out
            mov         R1, AOK_CODE
            w_time            0x0 ; clear timeout counter
            and         R0, READY_FLAG
            bne         GOTIT ; brach off the and setting the flags
            rel         R9 ;  must release request if we didnt get
            mov         R1, ERROR_CODE ;move error code into R9
GOTIT:            mov         R9, R1
            jre         0(R15) ;return


;; EVENTS
;rtp_event_clear -clears one or more event flags
;; event_clr(event_id, mask)
.globl _clear
_clear:           w_time            0x00 ; clear timer, no expire needed
            lock        resourceRMW
            read        r0, r9      ;
            and         r0, r10     ; mask events
            write       R9, R0  ;write them back
            rel         resourceRMW
            return


;rtp_event_signal - sets one or more events
;;;event_signal(event_id, mask)
.globl _signal
_signal:    w_time            0x00 ;  clear timer
            lock        resourceRMW
            read        r0, r9      ;
            or          r0, r10
            write       R9, R0 ;signals event
            rel         resourceRMW
            return


;rtp_res_ref general resource reference
;res_ref(event_id)
.globl _res_ref
_res_ref:   read        R9, R9
            jre         0(R15)


;MAILBOXES


;rtp_mailbox_post-dont block if we cant send
;;can call from C
;mail_post(mailboxwriteres, mailbox_addr, message)
.globl _mail_post
_mail_post: mov         R12, R9     ;  copy write resource since it will
die
            push        R15
            call        _re_gain  ;check if empty (empty flag in R9)
            cmp         R9, ERROR_CODE;comparison to check if got mutex
            be          DONEMAILPOST
            out         R11, R10 ;write message stored in msg reg R12
            rel         R12; release full flag
DONEMAILPOST:     pop         R15
            jre         0(R15) ;return


;rtp_mailbox_send -block until it can send
```

```
;;can call from C
;;; mail_send(mailboxwriteres,time_out_val, mail_box_addr, message)
.globl _mail_send
_mail_send: mov         R3, R15 ; return address will get killed
            mov         R2, R9 ;  writre resource will get killed
            call        _re_seek
            cmp         R9, ERROR_CODE
            be          DONEMAILSEND
            out         R12, R11 ; write message stored in msg reg(R11)
            rel         R2 ; release mailboxwriteres
DONEMAILSEND:     move        R15, R3     ; restores return address
            jre         0(R15) ;return


;rtp_mailbox_gain - dont block, return error code if empty
;; can call from C
;;; mail_gain(mailboxreadres, memory_address)
.globl mail_gain
_mail_gain: mov         R12, R15
            mov         R11, R9 ; read resource will get stomped on
            call        _re_gain
            mov         R15, R12 ;  restore return address
            cmp         R9, ERROR_CODE ;comparison to check if full
            be          DONEMAILGAIN
            read        R3, R11 ;  read resource to get mailbox address
            in          R2, R3 ;read message into msg reg(R10)
            st          R2, 0(R10) ; store msg in R2 to memory
            rel         R11 ; release read resource
DONEMAILGAIN:     jre         0(R15) ;return

;rtp_mailbox_seek block on an empty mailbox
;; can call from C
;;; mail_seek(mailboxreadres,timeout_val, memory_address)
.globl _mail_seek
_mail_seek: mov         R12, R15
            mov         R3, R9 ;  save location of read resource
            call        _re_seek
            mov         R15, R12 ; restore return address
            cmp         R9, ERROR_CODE ;see if we acquired the mutex
            be          DONEMAILSEEK
            read        R12, R3     ; read read resource to get address
            in          R2, R12
            st          R2, 0(R11)
            rel         R3 ;  release read resource
DONEMAILSEEK:     jre         0(R15) ;; return


;rtp_mailbox_reference - retrieves a message, doesnt remove it
;; can call from C
;;; mail_ref(mailboxfull,time_out_val, memory_address)
.globl _mail_ref
_mail_ref:  mov         R12, R15
            mov         R3, R9
            call        _re_seek
            mov         R15, R12
            cmp         R9, ERROR_CODE; did we get it??
            be          DONEMAILREF ; end if we didnt
            read        R12, R3
            in          R2,R12
            st          R2, 0(R11)
            rel         R3 ;; its still full
DONEMAILREF:      jre         0(R15) ;return
```

```
;; QUEUES


;rtp_queue_write-write to the queue until message is done
;;; qu_write(queue_write_res, timeout_val, message_address, message_len)
;;; R1 has the queue_address, R2 num bytes written, R3 has block or
nonblock
.globl _qu_write
_qu_write:  ld          R0, 0(R11)
            out         R0, R1 ;write message stored in R0 to queue
            add         R2, 2 ; increment num bytes written
            sub         R12, 1 ;see if message is finished
            be          WRDONE  ; branch to done if R11 is 0
            push        R9
            push        R15
            cmp         R3, BLOCK
            be          BLOCKINGW
            call        _re_gain
            jmp         REUNITEW
BLOCKINGW:  call        _re_seek; relock a resource we already have
                          ;; this will block if queue is full
REUNITEW:   pop         R15
            cmp         R9, ERROR_CODE; was resource acquired?
            be          WRDONE
            pop         R9
            add         R11, 0x2 ;;move to next address in mem
            jump        _qu_write ; write again
WRDONE:           jre         0(R15)


;rtp_queue_read-read from the queue until message is done
;;; qu_read(queue_read_res,timeout_val,mem_store_address, message_len)
;;; R1 has the queue_address, R2 num bytes written, R3 has block or
nonblock
.globl _qu_read
_QU_READ:   in          R0, R1 ;read message into reg
            st          R0, 0(R11) ; store msg in R0 to memory
            add         R2, 1
            sub         R12, 1;see if message is finished
            be          RDDONE  ; branch to done if R6 is 0
            push        R9
            push        R15
            cmp         R3, BLOCK
            be          BLOCKINGR
            call        _re_gain
            jmp         REUNITER
BLOCKINGR:  call        _re_seek; relock a resource we already have
                          ;; this will block if queue is empty
REUNITER:   pop         R15
            cmp         R9, ERROR_CODE; was resource acquired?
            be          RDDONE
            pop         R9
            add         R11, 0x2; increment memory location
            jump        _qu_read ; read again
RDDONE:           jre         0(R15);; return


;rtp_queue_post- non-blocking send message to queue
;;; qu_post(queue_write_res, message_address, message_len)
.globl _qu_post
_qu_post:   push        R15   ;  return address gonna get whacked
            mov         R2, R9 ; queue_write resource will die
            call        _re_gain ;can we write to the queue?
            cmp         R9, ERROR_CODE ;did we get mutex??
```

```
            mov         R9, 0x0     ; wont affect flags, 0 bytes were
writ
            be          DONEQUPOST ;branch to done if didnt get lock
            mov         R9, R2      ; move write_res back into param reg
            and         R2, 0x0     ;  clear R2 to store bytes written
            mov         R3, NONBLOCK ; this is a nonblocking call
            read        R1, R9 ; read queue_address into R1
            call        _qu_write; time to write
            rel         R9 ; release write flag
            mov         R9, R2 ;  move num bytes written to return reg
DONEQUPOST: pop         R15;  restore the return address
            jre         0(R15) ;; return


;rtp_queue_send -block until it can send
;; qu_send(queue_write_res, timeout_val, message_address,message_len)
.globl _qu_send
_qu_send:   push        R15   ;  return address gonna get whacked
            mov         R2, R9 ; queue_write resource will die
            call        _re_seek     ;can we write to the queue?
            cmp         R9, ERROR_CODE ;did we get mutex??
            mov         R9, 0x0     ; wont affect flags, 0 bytes were
writ
            be          DONEQUSEND ;branch to done if didnt get lock
            mov         R9, R2      ; move write_res back to R9
            and         R2, 0x0     ; zero out R2 to store bytes written
            mov         R3, BLOCK
            read        R1, R9 ; read queue_address into R1
            call        _qu_write; time to write
            rel         R9
            mov         R9, R2 ; mov num bytes written to return reg
DONEQUSEND: pop         R15 ;  restore return addr
            jre         0(R15) ;; return



;rtp_queue_gain-non-blocking read
;;; qu_gain(queue_read_res, NULL,message_address,message_len)
.globl _qu_gain
_qu_gain:   push        R15
            mov         R2, R9
            call        _re_gain    ;can we read from the queue?
            cmp         R9, ERROR_CODE ;did we get mutex??
            mov         R9, 0x0     ; wont affect flags, 0 bytes were
read
            be          DONEQUGAIN ;branch to done if didnt get lock
            mov         R9, R2 ;  restore read_resource
            and         R2, 0x0     ;  R2 will store bytes read
            mov         R3, NONBLOCK
            read        R1, R9 ; read queue_address into R1
            call        _qu_read; time to read
            rel         R9
DONEQUGAIN: pop         R15
            jre         0(R15);; return


;rtp_queue_seek -block until read
;; qu_seek(queue_read_res,timeout_val,mem_store_address,message_len)
.globl _qu_seek
_qu_seek:   push        R15
            mov         R2, R9
            call        _re_seek    ;can we read from the queue?
            cmp         R9, ERROR_CODE ;did we get mutex??
```

```
            mov         R9, 0x0     ; wont affect flags, 0 bytes were
read
            be          DONEQUSEEK ;branch to done if didnt get lock
            mov         R9, R2 ;  restore read_resource
            and         R2, 0x00    ;  R2 will store bytes read
            mov         R3, BLOCK
            read        R1, R9 ; read queue_address into R1
            call        _qu_read; time to read
            rel         R9
DONEQUSEEK: pop         R15
            jre         0(R15);; return


;; system timer functions
;; rtp_get_time- will return value of universal timer in r10
;;; get_time(timer_address)
.globl _get_time
_get_time:  in          R10, R9
            return

;;; rt_set_time  - will set the value of the universal timer with value
in R9
;;; set_time(timer,_address,new_timer_value)
.globl _set_time
_set_time:  out         R10, R9
            return


.globl _create_task
      ; create_task(function, proc_id, task_id, task_priority,
stacksize, arg1, arg2)
      ;             R9        R10      R11      R12           -2(R14)  -
4(R14) -6(R14)
_create_task:
      mova  R1, my_proc_id
      ld    R0, 0(R1)
      cmp   R0, R10           ; PPPP = my_proc_id?
      be    _ct1        ; yes good, no bad
      mov   R9, 1       ; error code
      ret               ; return error
_ct1: mova  R0, tasktable     ; &tasktable[0]
      mov   R1, R11           ; task_id
      sll   R1, 3       ; 8 * task_id
      add   R0, R1            ; &tasktable[task_id]
      ld    R1, 0(R0)   ; tasktable[task_id].function
      cmp   R1, 0       ; = 0?
      be    _ct2        ; yes good, no bad
      mov   R9, 2       ; if task already inited: error code
      ret               ; return error
_ct2: st    R9, 0(R0)   ; setup tasktable[task_id]
      ld    R1, 0(R14)  ; stacksize
      st    R1, 2(R0)   ; tasktable[task_id].stacksize = stacksize
      ld    R1, 2(R14)  ; arg1
      st    R1, 4(R0)   ; tasktable[task_id].arg1 = arg1
      ld    R1, 4(R14)  ; arg2
      st    R1, 6(R0)   ; tasktable[task_id].arg2 = arg2

      mov   R1, R10           ; PPPP
      and   R1, 0xF           ; PPPP
      sll   R1, 4       ; PPPP 0000
      and   R11, 0xF    ; TTTT
      or    R1, R11           ; PPPP TTTT
      sll   R1, 8       ; PPPP TTTT 0000 0000
      or    R1, 0x10    ; PPPP TTTT 0001 0000 (set task ready flag)
```

```
      and   R12, 0xF    ; XXXX
      or    R1, R12           ; PPPP TTTT FFFF XXXX
      w_prio      R1          ; set priority and ready (unblocks task)
      return


      .area DSEG
      .even
my_proc_id: .dw -1            ; get inited by GSINIT
current_stack:    .dw stacktop
.globl _main
tasktable:  .dw _main   ; task 0 - branch address
            .dw 60            ; task 0 - stack size
            .dw 0       ; task 0 - arg1 (argc) to main()
            .dw 0       ; task 0 - arg2 (argv) to main()
            .blkw 15*4  ; reserve room for the other 15 tasktable
entries (4 words each)
      ; 66 bytes total of data memory used by _gsinit() and
create_task()

      .area SSEG        ; placed at top of memory
stacktop:               ; first non-existant memory location
```

Appendix D: RTP Instruction Set Architecture

Mnemonic	Operands	Description	Operation	FLAGS
Arithmetic And Logic Instructions				
add R, L	Reg, L	add	[C | R] <= [Reg] + L(sign extended)	Updated
add R1, R2	Reg, Reg	add	[C | R1] <= [R1] + [R2]	Updated
addc R, L	Reg, L	add with carry	[C | R] <= [Reg] + L + C	Updated
addc R1, R2	Reg, Reg	add with carry	[C | R1] <= [R1] + [R2] + C	Updated
sub R, L	Reg, L	subtract	[C | R] <= [Reg] - L	Updated
sub R1, R2	Reg, Reg	subtract	[C | R1] <= [R1] - [R2]	Updated
subc R, L	Reg, L	subtract with carry	[C | R] <= [Reg] - L(sign extended) - C	Updated
subc R1, R2	Reg, Reg	subtract with carry	[C | R1] <= [R1] - [R2] - C	Updated
rsub R. L	Reg, L	reverse subtract	[C | R1] <= -[R1] + L	Updated
rsub R1, R2	Reg, Reg	reverse subtract	[C | R1] <= -[R1] + [R2]	Updated
rsubc R, L	Reg, L	reverse subtract with carry	[C | R1] <= -[R1] + L	Updated
rsubc R1, R2	Reg, Reg	reverse subtract with carry	[C | R1] <= -[R1] + [R2]	Updated
mul R, L	Reg, L	Multiply word	[T,R] <= [Reg]*L, signed	Updated
mul R1, R2	Reg, Reg	Multiply word	[T,R1] <= [R1]*[R2], signed	Updated
mulb R, L	Reg, L	Multiply low-byte	[Reg](15..0) <= [Reg](7..0)*L, signed	Updated
mulb R1, R2	Reg, Reg	Multiply low-byte	[R1] <= [R1](7..0)*[R2](7..0), signed	Updated
mulu R, L	Reg, L	Multiply unsigned word	[T,R] <= [Reg]*L, unsigned	Updated
mulu R1, R2	Reg, Reg	Multiply unsigned word	[T,R1] <= [R1]*[R2], unsigned	Updated
mulub R, L	Reg, L	Multiply unsigned low-byte	[Reg](15..0) <= [Reg](7..0)*L, unsigned	Updated
mulub R1, R2	Reg, Reg	Multiply unsigned low-byte	[R1] <= [R1](7..0)*[R2](7..0), unsigned	Updated
and R, L	Reg, L	And literal	[Reg] <= [Reg] AND (0,L)	Updated
and R1, R2	Reg, Reg	And literal	[R1] <= [R1] AND [R2]	Updated
andc R, L	Reg, L	And with carry	[Reg](7..0) <= [Reg](7..0) AND L	Updated
andc R1, R2	Reg, Reg	And with carry	[R1](7..0) <= [R1](7..0) AND [R2](7..0)	Updated
or R, L	Reg, L	Or	[Reg] <= [Reg] OR (0,L)	Updated
or R1, R2	Reg, Reg	or	[R1] <= [R1] OR [R2]	Updated
orc R, L	Reg, L	Or with carry	[Reg](7..0) <= [Reg](7..0) OR L	Updated
orc R1, R2	Reg, Reg	Or with carry	[R1](7..0) <= [R1](7..0) OR [R2](7..0)	Updated
xor R, L	Reg, L	xor	[Reg] <= [Reg] XOR (0,L)	Updated
xor R1, R2	Reg, Reg	xor	[R1] <= [R1] XOR [R2]	Updated
xorc R, L	Reg, L	xor with carry	[Reg](7..0) <= [Reg](7..0) XOR L	Updated
xorc R1, R2	Reg, Reg	xor with carry	[R](7..0) <= [R](7..0) XOR [R2](7..0)	Updated
Branch Instructions				
call	L	call	[R15] <= PC, PC <= L (relocatable)	Unchanged
jump	L	jump	PC <= L (relocatable)	Unchanged

jre L(R)	Reg, L	Jump register	PC <= L + [Reg]	Unchanged
jre R2(R1)	Reg, Reg	jump register	PC <= [R1] + [R2]	Unchanged
ret		Subroutine return	PC <= [R15]	Unchanged
cmp R, L	Reg, L	compare with carry	FLAG <= [Reg] - L(zero extended)	Updated
cmp R1, R2	Reg, Reg	compare with carry	FLAG <= [R1] - [R2]	Updated
cmpc R, L	Reg, L	compare with carry	FLAG <= [Reg](7..0) - L	Updated
cmpc R1, R2	Reg, Reg	compare with carry	FLAG <= [R1](7..0) - [R2](7..0)	Updated
be		branch if equal	if Z=1 then PC <= PC+L	Unchanged
beu		branch if equal, unsigned	if ZU=1 then PC <= PC+L	Unchanged
bge		branch if greater or equal	if N=0 then PC <= PC+L	Unchanged
bgeu		branch if greater or equal, unsigned	(NU=0=always) PC <= PC+L ; jump relative	Unchanged
bgt		branch if greater	if C=0 then PC <= PC+L	Unchanged
bgtu		branch if greater, unsigned	if CU=0 then PC <= PC+L	Unchanged
ble		branch if lesser or equal	if C=1 then PC <= PC+L	Unchanged
bleu		branch if lesser or equal, unsigned	if CU=1 then PC <= PC+L	Unchanged
blt		branch if lesser	if N=1 then PC <= PC+L	Unchanged
bltu		branch if lesser, unsigned	(NU=1=never); never jump; nop	Unchanged
bne		branch if not equal	if Z=0 then PC <= PC+L	Unchanged
bneu		branch if not equal, unsigned	if ZU=0 then PC <= PC+L	Unchanged
bnv		branch if not overflow	if V=0 then PC <= PC+L	Unchanged
bnvu		branch if not overflow, unsigned	if VU=0 then PC <= PC+L	Unchanged
bv		branch if overflow	if V=1 then PC <= PC+L	Unchanged
bvu		branch if overflow, unsigned	if VU=1 then PC <= PC+L	Unchanged
Data Transfer Instructions				
mov R, L	Reg, L	Move	[Reg] <= L, sign extended	Unchanged
mov R1, R2	Reg, Reg	Move	[R1] <= [R2], sign extended	Unchanged
mova R, L	Reg, L	16 bit mov	R <= L (16 bit), pseudo op	Unchanged
movh R, L	Reg, L	Move hi-byte	[Reg](15..8) <= L, [Reg](7..0) unaffected	Unchanged
movh R1, R2	Reg, Reg	Move hi-byte	[R1](15..8) <= [R2], [R1](7..0) unaffected	Unchanged
movu R, L	Reg, L	Move unsigned	[Reg] <= L, zero extended	Unchanged
movu R1, R2	Reg, Reg	Move unsigned	[R1] <= [R2], zero extended	Unchanged
movuh R, L	Reg, L	Move unsigned hi-byte	[Reg](15..8) <= L, [Reg](7..0) <= 0	Unchanged
movuh R1, R2	Reg, Reg	Move unsigned hi-byte	[R1](15..8) <= [R2], [R1](7..0) <= 0	Unchanged
ld R1, L(R2)	Reg, Reg, L	load	[R1] <= Data(L + [R2])(15..0)	Unchanged
ldb R1, L(R2)	Reg, Reg, L	load byte	[R1] <= Data(L + [R2])(7..0)	Unchanged
st R1, L(R2)	Reg, Reg, L	store	Data(L + [R1])(15..0) <= [R2]	Unchanged
stb R1, L(R2)	Reg, Reg, L	store byte	Data(L + [R1])(7..0) <= [R2]	Unchanged
in R, L	Reg, L	In port	addr <= L, [Reg] <=in	Unchanged
in R1, R2	Reg, Reg	In port	addr <= [R2], [R1] <=in	Unchanged
out R, L	Reg, L	Out port	addr <= L, out <= [Reg]	Unchanged
out R1, R2	Reg, Reg	Out port	addr <= [R2], out <= [R1]	Unchanged
push R	Reg	push on stack	[SP] <= [Reg]	Unchanged
pop R	Reg	pop off stack	[Reg] <= [SP]	Unchanged
xch R, T	Reg, L	Exchange	[T,Reg] <= [Reg,T]	Unchanged
Bit Instructions				
clrc		clear carry	or r0,0	C,UC <-- 0

clrt		clear transfer	sll r0,0	T <-- 0
rol R, L	Reg, L	Rotate left	[T,Reg] <= ([0,Reg] << L) | ([0,Reg] << (16-L)) >> 16	Updated
rol R1, R2	Reg, Reg	Rotate left	[T,R1] <= ([0,R1] << R2) | ([0,R1] << (16-R2)) >> 16	Updated
ror R, L	Reg, L	Rotate Right	[Reg,T] <= ([0,Reg] << (16-L)) | ([0,Reg] << L) >> 16	Updated
ror R1, R2	Reg, Reg	Rotate right	[R1,T] <= ([0,R1] << (16-R2)) | ([0,R1] << R2) >> 16	Updated
slc R, L	Reg, L	Shift left with carry	[T,Reg] <= ([0,Reg] << L) | [0,T]	Updated
slc R1, R2	Reg, Reg	Shift Left with carry	[T,R1] <= ([0,R1] << R2) | [0,T]	Updated
sll R, L	Reg, L	Shift left logical	[T,Reg] <= [0,Reg] << L	Updated
sll R1, R2	Reg, Reg	Shift Left Logical	[T,R1] <= [0,R1] << R2	Updated
sra R, L	Reg, L	Shift right arithmetic	[Reg,T] <= [SE,Reg] << (16-L)	Updated
sra R1, R2	Reg, Reg	Shift Right Arithmetic	[R1,T] <= [SE,R1] << (16-R2)	Updated
src R,L	Reg, L	Shift right with carry	[Reg,T] <= ([0,Reg] << (16-L)) | [T,0]	Updated
src R1,R2	Reg, Reg	Shift Right with carry	[R1,T] <= ([0,R1] << (16-R2)) | [T,0]	Updated
srl R, L	Reg, L	Shift right logical	[Reg,T] <= [0,Reg] << (16-L)	Updated
srl R1, R2	Reg, Reg	Shift Right Logical	[R1,T] <= [0,R1] << (16-R2)	Updated
Control and Special Purpose Functions				
nop		No Operation		Unchanged
disable	Reg or L	Disable Resource	disable resource {[Reg], L} (disable interrupt {[Reg], L})	Unchanged
enable	Reg or L	Enable Resource	enable resource {[Reg], L} (enable interrupt {[Reg],L})	Unchanged
ds		Disable Scheduling	(this processor only)	Unchanged
es		Enable Scheduling	(this processor only)	Unchanged
lock	Reg or L	Lock Resource	task[this].resource[{[Reg],L}].req <= 1, block until req==0 or timeout=1	Unchanged
nb_lock R, L	Reg, L	Non-Blocking Lock	attempt to lock L, [Reg] <- info on task that owns resource L	Unchanged
nb_lock R1, R2	Reg, Reg	non-blocking lock	attempt to lock [R2], [R1] <- info on owner of resource [R2]	Unchanged
r_prio	Reg	Read Priority	[Reg] <= this.task.priority	Unchanged
r_time	Reg	Read Timeout	[Reg] <= this.task.timeout	Unchanged
w_prio	Reg or L	Write Priority	task{[Reg], this}.priority <= {[Reg], L}	Unchanged
w_time	Reg or L	Write Timeout	this.task.timeout <= {[Reg], L}	Unchanged
read R, L	Reg, L	Read resource status	[Reg] <= status of resource L	Unchanged
read R1, R2	Reg, Reg	Read resource status	[R1] <= status of resource [R2]	Unchanged
write R, L	Reg, L	Write resource Status	status of resource L <= [Reg]	Unchanged
write R1, R2	Reg, Reg	Write resource Status	status of resource [R2] <= [R1]	Unchanged
rel	Reg or L	Release Resource	task[this].resource[{[Reg], L}].gnt <= req <=0	Unchanged
rst	Reg or L	Reset Resources	task[*].resource[{[Reg], L}].req <= gnt <= 0	Unchanged
sig1	Reg or L	Send Signal 1	send sig 1 to resource {[Reg], L} (set int. {[Reg], L})	Unchanged
sig2	Reg or L	Send Signal 2	send sig 2 to resource {[Reg], L} (reset int. {[Reg], L})	Unchanged