2007-07-12

Hardware Support for a Configurable Architecture for Real-Time Embedded Systems on a Programmable Chip

Spencer W. Isaacson
Brigham Young University - Provo

HARDWARE SUPPORT FOR A CONFIGURABLE ARCHITECTURE FOR REAL-TIME EMBEDDED SYSTEMS ON A PROGRAMMABLE CHIP

by

Spencer William Isaacson

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

Spencer William Isaacson

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
Date	James K. Archibald
Date	Michael J. Wirthlin

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of Spencer W. Isaacson 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

HARDWARE SUPPORT FOR A CONFIGURABLE ARCHITECTURE FOR REAL-TIME EMBEDDED SYSTEMS ON A PROGRAMMABLE CHIP

Spencer William Isaacson

Department of Electrical and Computer Engineering

Master of Science

Current FPGA technology has advanced to the point that useful embedded SoPCs can now be designed. The Real Time Processor (RTP) project at Brigham Young University leverages the advances in FPGA technology with a system architecture that is customizable to specific applications. A simple real-time processor has been designed to provide support for a hardware-assisted real-time operating system providing fast context switches. As part of the hardware RTOS, the following have been implemented in hardware: scheduler, register banks, mutex, semaphore, queue, interrupts, event, and others. A novel circuit called the Task-Resource Matrix has been created to allow fast inter/intra processor communication and synchronization.

ACKNOWLEDGMENTS

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

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 Introduction

It is hard to find an embedded system designer that doesn't want better performance, lower cost, and more predictable behavior from his embedded design. More and more processing power is needed for larger and more complex applications that are breaching the world of the embedded market. The software RTOS market has relied heavily on increased processing power to increase the productivity of its software. Few new mechanisms for increasing real-time stability have entered the market without the aid of specialized hardware. Direct Message Passing implemented in Windriver's VxWorks is an attempt at decreasing RTOS overhead giving more processor time to tasks without specialized hardware.[22] Message passing, however, is only one of the many elements of the RTOS that consume processor cycles. An alternative to improved software algorithms is to implement RTOS elements in hardware. These elements become especially suited for system on a chip designs.

It is becoming common for processors, memory, and custom hardware to all be integrated on a single System-on-a-Chip (SoC). 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 Systems-on-a-Programmable-Chip (SoPCs). Several recent innovations in hardware/software co-design target SoCs in an effort to improve embedded system performance and design [1, 2, 3]. As FPGAs increase in gate

count and memory capacity, SoPCs will continue to increase in popularity. Implementing a function in specialized hardware, rather than in software running on a general purpose processor, is faster and more predictable [5]. On an FPGA, where hardware elements are abundant and memory space is limited, it is the software that is more expensive, giving specialized hardware functions the triple benefit of being faster, more predictable, and cheaper.

Previous work has shown numerous advantages of implementing a Real Time Operating System (RTOS) in hardware [5,6,7]. With abundant hardware resources, an FPGA is a perfect target for a hardware-based OS. Implementing an RTOS primarily in hardware mitigates a primary weakness of FPGAs: the lack of memory space. Traditionally inter-task communication and synchronization are accomplished through the use of data primitives implemented in shared memory and controlled by the RTOS. These primitives are not only costly in processor cycles, but are also costly in memory usage. This is further complicated when working with multiple processing cores. Moving communication and synchronization primitives to hardware not only reduces memory usage but also reduces processor cycles.

Task switches in software can enfetter hundreds of processor cycles per task switch. While the delay of a task switch is very dependant on software algorithms and hardware implementations, it can quickly consume many of the processor cycles available to tasks.

The Real Time Processor (RTP) project aims to provide a new architecture for embedded system co-design. The system architecture includes a custom tailored processor, a hardware scheduler, and an advanced task and resource management

structure called the Task-Resource Matrix (TRM) that implements the synchronization primitives in hardware for a variety of resource types.

The RTP architecture aims to create a simple-to-use customizable embedded system utilizing fast inter-task and inter-processor communication, fast context switching, and RTOS primitives all implemented in hardware. This reduces the OS software core to only around 400 instructions. In comparison, the ThreadX kernel is around 2KB with the absolute minimal functions of an RTOS. If you add mutex, semaphore and queue support the size of the RTOS grows to 4.5K.[20] The uC/OSII minimally configured and targeted to a special processor its minimal size is still over 2KB.[21] Even when Xilinx and Realfast collaborated to build a hardware assisted RTOS for the Microblaze processor, the software portion of the RTOS is still "a couple of KiloBytes".[4][6] The final design goal of the RTP architecture is a multiprocessor real-time embedded system on a single programmable chip. An ideal processor would therefore have enough power to run multiple tasks with reasonable performance, yet be lightweight enough that several could easily fit on a single FPGA.

Figure 1-1 shows an overview of the RTP architecture. The architecture is a multiprocessor system, containing an array of 1 to 16 processors. The processor utilizes fast context switching by maintaining state information for each task. The architecture limits the number of tasks to 16 per processor due to the manner in which state information is saved. More tasks can be hosted by using additional processors. The small footprint of the RTP encourages the use of multiple processors. Each processor has its own local memory for code and data, shared by all tasks on that processor.

Task-Switching within the processor is controlled by the hardware scheduler implemented as part of the RTP project. The scheduler provides the processor with the identifier of the highest priority ready task.

Each processor also has its own local I/O space of up to 256 ports. A 16-bit local I/O bus is connected to all of the peripherals used by that processor, such as serial ports, queues, and network interfaces. Peripherals that can be shared by two or more tasks, whether they are on a single processor or on different processors, need to be synchronized among the competing tasks. This is done using system "resources" in this architecture. System resources include mutexes, semaphores, timers, 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 TRM, shown in Figure 1-1, is an innovative feature of this architecture that controls the sharing of resources in a unified way. The matrix is sparsely populated, meaning that a resource node, the connection between a task and a resource, is only populated if that task needs access to that resource. The resource node keeps track of pending requests and grants for a resource. The TRM provides important hardware assistance to the RTOS by keeping track of the tasks that are waiting for specific resources. This new circuit provides information to the scheduler about what tasks are "blocked" waiting for any un-granted resources. A shared global resource bus is used to access system resources and task modules. Since this bus is shared, processors must arbitrate to use it.

The processor interface is designed to operate with the TRM as well as communicate with resource modules and peripherals.

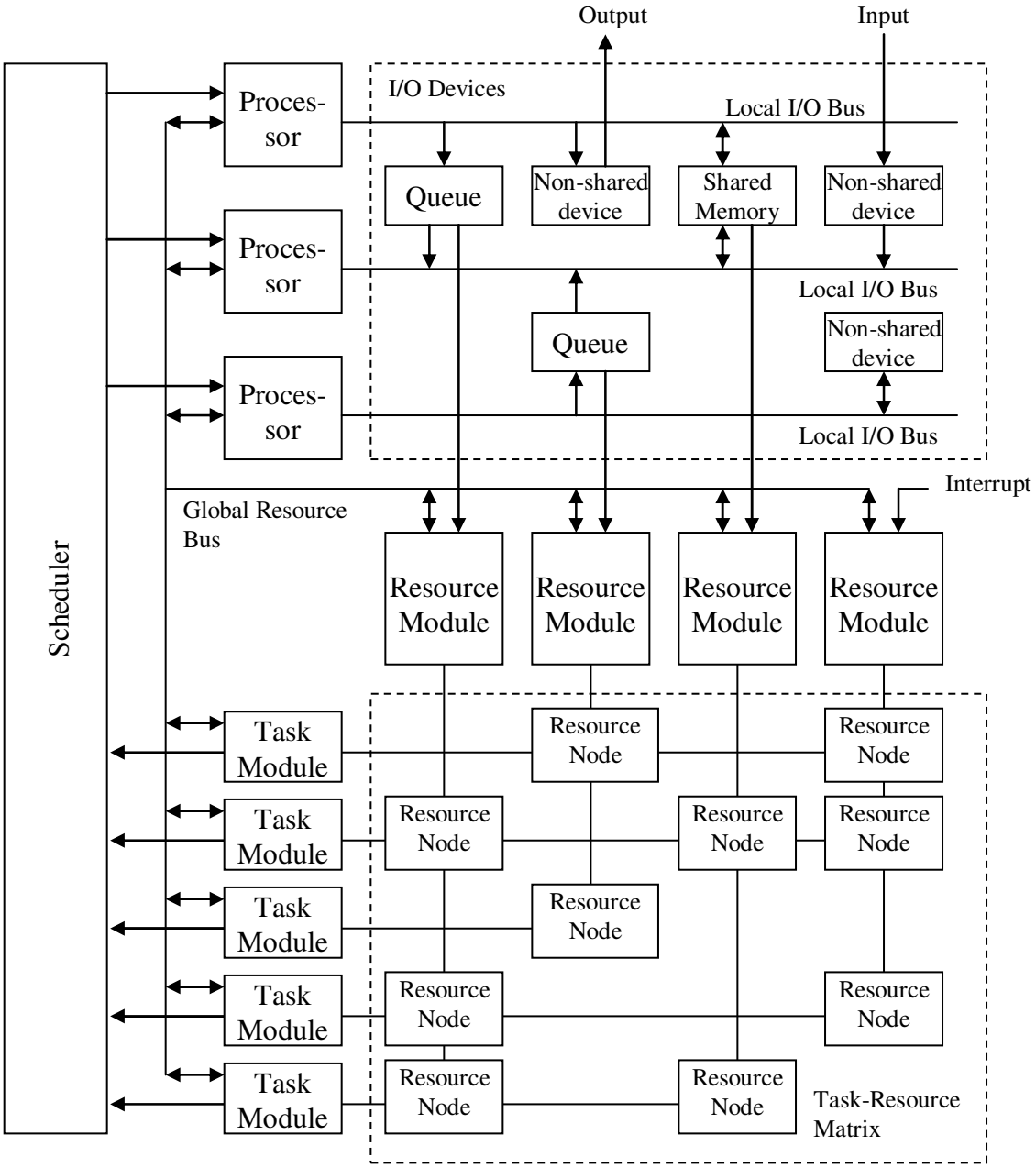

Figure 1-1:RTP Project Architecture

1.1 Contributions

My contributions to the research of this project are as follows:

1. I designed and implemented the Task-Resource Matrix, a new and innovative feature of the RTP system. I also published and presented a peer-reviewed paper on the Task-Resource Matrix.[19]
2. I helped extend a standard RISC instruction set architecture with new instructions that accelerate execution of a real-time system enabling the RTOS to be written in about 400 instructions. This is less than half the size of any other RTOS that I have found, even when compared with a hardware assisted RTOS.

This thesis gives the details of the hardware implemented as part of the RTP project. It is divided into the following sections: Chapter 2 elaborates on the Real Time Processor, Chapter 3 gives details on the hardware scheduler implemented, Chapter 4 describes the Task Resource Matrix, and Chapter 5 is the conclusion.

2 The Real Time Processor (RTP)

Advances in FPGA technologies have allowed for microprocessors implemented as part of FPGA fabric known as soft-core processors. Increasing amounts of FPGA fabric has made it possible to include both soft-processors and custom logic within the same device, enabling entire embedded systems on a single chip. Having the ability to design a complete system on a programmable chip with costs starting at less than twelve dollars have lead to a trend of more systems developed for FPGAs using soft-core processors.[4] Most current FPGA soft-core processors are fashioned after general purpose ASIC processors with little possibility for customization. Recently, soft processor cores have been improved with the ability to add customized instructions of varying clock cycle latencies.

In 2004, Altera® introduced its Nios® II soft processor with capabilities of having up to 256 application-specific instructions. The processor core has three versions: fast core (Nios II/f) for high performance design, the economy core for lower cost solutions and the standard Nios core. The previous version of the soft processor core only had one application-specific instruction that had the limitation of one processor clock cycle latency. [15]

Xilinx® has also introduced two soft processors, the Picoblaze and the MicroBlaze. The newest version of the PicoBlaze processor has the capability of

addressing 1024 18-bit instructions, has16x 8-bit general purpose registers, a stack depth of 31, a 64 byte scratch pad memory, and can address 256 input devices and 256 output devices. Previous versions of the PicoBlaze did not have any memory at all (not even the small stack), and could only address 256 instructions total. A major benefit of these streamlined features is the small footprint of the processor. On a Spartan-3 FPGA with all of the features, the PicoBlaze uses 96 slices. On a Spartan-IIE with more limited functionality, it only requires 76 slices.[4]

The MicroBlaze soft processor is on the other edge of the spectrum. It requires 1050 slices on a Spartan-IIE and comes with a higher level of functionality, including 32x32bit registers, 32 bit arithmetic, and a host of configurable intellectual property cores to add on. This IP has a great advantage over traditional off-the-shelf processor design in that the ability exists to include synthesized hardware elements such as UARTs and giga-bit Ethernet cores, but only when necessary.[4] None of the soft core processors mentioned utilizes fast-context switching. The MicroBlaze, PicoBlaze, and Nios processors limit the customization with the IPcores and some application specific instructions. The processor implemented as part of the RTP project goes a step further.

2.1 RTP System

The RTP system is meant to provide the hardware support that an embedded system's application would need without cluttering the system with un-needed elements. Application code is analyzed to find the tasks and the resources that those tasks need in order to communicate with the rest of the system, communicate with the other tasks, and synchronize themselves with all of the elements of the system. The analysis of these

elements is done at compile time. The output of the analysis gives the number of tasks necessary, the number of processors needed to implement these tasks, the communication primitives needed, i.e. queues, semaphores, mailboxes, etc., and the connections each task needs to each element.

For example, consider a system with three tasks, A, B and C. Task A reads data from an input port, and passes that information to Task B with a queue. Task A knows new data is ready via an interrupt set externally. Task C sets an output flag only when task A and B are finished with their "munching" of the data. In this example, the number of tasks is obvious, what is less obvious is how many processors are needed, and the priority of these tasks. Because there are only 3 tasks, this can easily be implemented on only one processor. If that task A is very time critical and needs to be able to service its interrupt very quickly, this would still not be a problem on one processor because context switching between tasks takes only 1 clock cycle. But to be sure, you can always implement two or even three processors and have A on its own processor. The user has the ability to define this during the compilation stage. The small footprint of the RTP processor encourages the use of multiple processors. The RTP system allows for 16 tasks per processor and up to 16 processors. This provides a total of 256 tasks within the system if required. One limitation of the design is the static allocation of tasks and resources.

Task A requires access to three elements or communication primitives: an interrupt, the write port of a queue, and a semaphore to notify C that it is finished. Task B needs access to the read port of the queue, and a semaphore to notify C. Task C needs access to both semaphores, but not the read queue and the interrupt. This type of

information is used in the construction of the RTP system and modifies the structure of the Real-Time Processor, Scheduler(s) and the Task-Resource Matrix. All of these structures are described in this document.

The processor interface in this system should not only be able to access the normal instruction and data memories (a Harvard memory architecture), but also access the I/O ports, and the Task-Resource Matrix. Figure 2-1 shows the interface of the Real-Time Processor. All the inputs and outputs of the RTP are listed. The signals are grouped according to the block with which they interface. The groupings of signals include interfaces to instruction memory, data memory, and I/O devices. The interface to the Task-Resource Matrix actually consists of three sub-interfaces: resource control, task control and node control. Each of these interfaces is described in more detail within the Task-Resource Matrix chapter. The dedicated input from the scheduler is the task ID of the ready task which has the highest priority for that processor.

The driving thoughts behind the architecture of the Real-Time Processor, or RTP, were simplicity and speed. Making the processor as simple as possible helps not only to reduce the amount of physical resources due to a decrease in specialized hardware, but also reduces the complexity of the controlling algorithms. However, making the processor faster requires some specialized hardware. The next sections will detail the architecture of the pipe stages, the specialized context switching hardware, and the arithmetic capabilities of the processor.

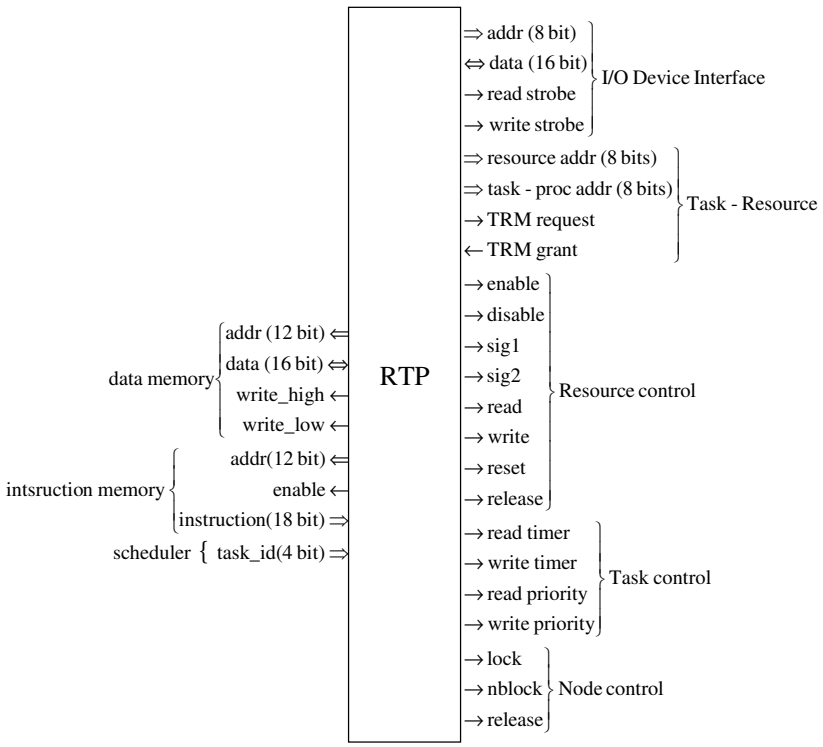

Figure 2-1: The Real Time Processor Interface

2.2 The Pipe

In keeping with the goal of simplicity, the natural conclusion was to make a RISC processor with some architecture-specific instructions. The processor was divided into only 3 pipe stages. This allows the cycles per instruction through this pipe to be 1 and greatly simplifies any forwarding logic. The first stage is instruction fetch. In this stage the appropriate program counter is accessed, the address is latched in the instruction memory and the program counter is incremented.

The second stage is the decode/execution/memory access stage. In this stage, the instruction is decoded, the registers and flag set are sourced and the results of any

mathematical operation are calculated. This includes offsets to memory accesses and branch target addresses. (The context switching section below will detail how this address is saved.) As part of the decode/execution/memory access stage, the task resource matrix access request is sent as well. The access is arbitrated external to the processor by the TRM arbiter.

The final stage is the write back stage. In this stage, the results of the previous stage are multiplexed with any external accesses and stored back into the registers.

The worst case path of the processor is the path from calculating the branch target address in stage 2 to the instruction fetch in stage 1. In order to maintain a CPI of 1 and not stall the pipe, this worst case path was implemented. In hind sight, this is probably a bad idea.

This can be improved. Throw out the idea of a CPI of 1 and accept that stalls are going to have to happen. The number of stages can be increased to 5 to better balance the load of the stages and to decrease the length of the critical paths. This will allow the frequency of the processor to be improved greatly and increase the overall throughput of the processor. An increase in the number of stages would also simplify some of the context switching specific hardware. However, the attendant cost will be increased with forwarding logic and increased control logic resulting in a larger footprint for the processor mitigating the benefit of adding multiple processors to the end system.

2.3 Context Switching Hardware

Fast context switching is one of the primary mechanisms for speeding up the entire embedded system. Fast context switching is not new.[6][9] Several commercially

available processors implement some sort of fast context switching. The ARM processor core duplicates a subset of registers for the different privileged operating modes. A duplicate register set does not exist for system and user operating modes. In addition, the program counter is never duplicated.[23] Due to this restriction the ARM cpu still takes several cycles to context switch. In contrast, context switching only takes 1 cycle in the RTP system. This is accomplished by the fact that no registers or flag sets have to be saved when switching context. Each task has its own program counter (PC), register set, and flag set. Because the task utilization is monitored at compile time, dynamic task creation or deletion is not supported.

2.3.1 Program Counter

Each task must be able to maintain where it is in execution. The program counter must then either be saved into the data memory or have a specialized register to save itself during a context switch. Figure 2-2 shows the block diagram for the program counter.

Note that the highlighted blue line is the worst case path through the program counter. This is from the output of the execution stage and goes to the input of the instruction memory.

The program counter is 12-bits wide. The program counter is implemented as a dual-ported distributed RAM of size number_of_tasks x 12 bits. Table 2-1 shows the slice count per number of tasks for the program counter. The SliceM count is the number of slices used as distributed memory. The program counter register is dual-ported to take into consideration the case where a context switch occurs after a branch instruction and the newly calculated branch target address must be written to the previously running

task's program counter at the same time that the running task updates its program counter. A simple work-around for this issue would be to not allow the task to switch during the cycle immediately following a branch or any program counter (PC) altering instructions.

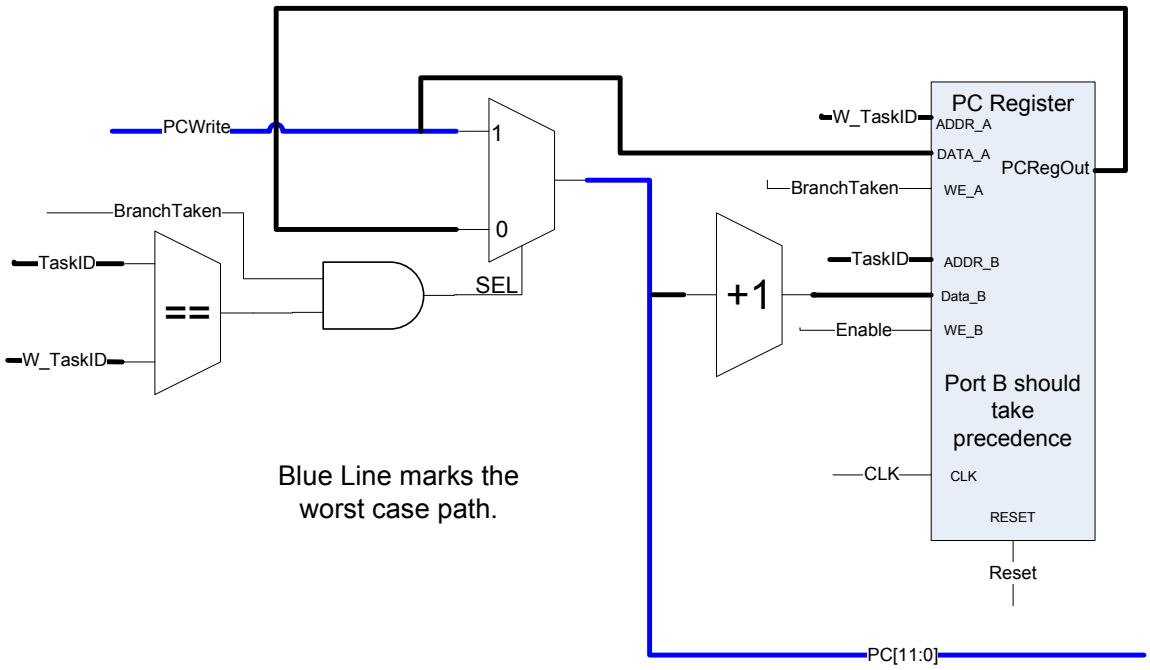

Figure 2-2: Block Diagram of the Program Counter

With the program counter being limited to only twelve bits, task code can branch to any address in the instruction memory space using a literal. However, this limits the address space to 2^{12} or 4096 addressable instructions per processor. The program counter can easily be modified to contain up to sixteen bits, but the ability to address the entire address space with a single literal instruction would be lost. The program counter can be further modified to contain more than sixteen bits, but the ability to address the entire address space with any single instruction would be lost, let alone the ability to address the

space with a single literal instruction. The amount of block memory built into current FPGA technology is the limiting factor for addressable instruction memory space. Future increases in the amount of block RAM will enable the addressing of larger memory spaces.

Table 2-1: PC Slice Count per Number of Tasks

Number of Tasks	Synthesis Slices	Post Implementation	Post Implementation SliceM
1	25	22	
2	39	35	
3	55	49	
4	61	55	
5	82	76	
6	89	81	
7	98	90	
8	106	99	
9	128	131	24
10	136	137	24
11	144	143	24
12	152	152	24
13	173	158	24
14	181	165	24
15	189	172	24
16	197	180	24

2.3.2 Register Set

As already mentioned, in order to make context switching times fast, each task has its own register set. The size of the register set is (number_of_tasks x 16) x 16. The address to the register set is derived from the concatenation of the task ID with the register address. For example, if a task with the task ID of 0010b wants to access a register addressed 0101b, the register address is 00100101b. The reconfigurable nature of the FPGA allows for the implementation of only the absolutely necessary register sets. The hardware generation tools will only include the number of register sets required for

each processor, thus making the footprint of the processor core as small as possible.

Table 2-2 shows the slice count per number of tasks for the register set.

Table 2-2: Register Slice Count per Number of Tasks

Number of Tasks	Synthesis Slices	Post Implementation	Post Implementation SliceM
1	48	32	32
2	60	85	64
3	91	135	96
4	103	172	128
5	151	237	160
6	166	274	192
7	183	315	224
8	200	353	256
9	246	411	288
10	282	465	320
11	319	515	352
12	332	554	384
13	364	603	480
14	382	641	512
15	400	676	544
16	421	714	576

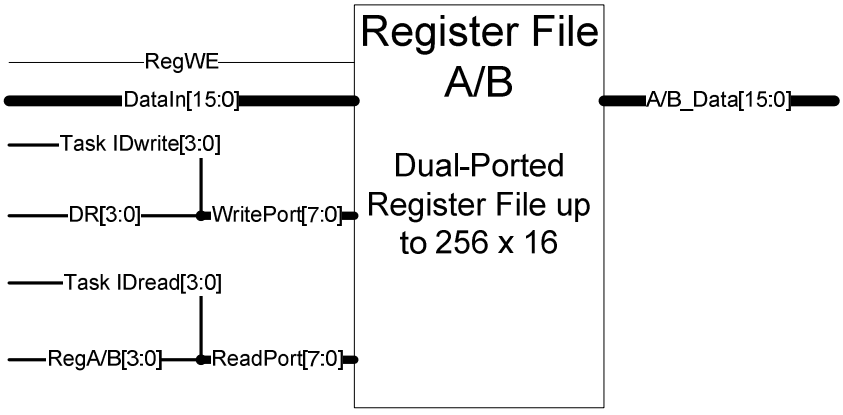

Figure 2-3: Single Register File Block Diagram

The register file is actually implemented as two dual port distributed RAMs to allow access of two registers for reading at the same time of writing contents to the registers. Figure 2-3 shows the instantiation of one of the dual-ported register files. Two of these are needed to complete the register file.

2.3.3 Flag Set

The set of flags crucial to the internal workings of the processor also require separate memory spaces for each task. The flag set is similar to the same implementation of the register file. The flag set, however, does not need to be duplicated like the register file. The flags will only be sourced once for reading and once for writing requiring a single dual-ported distributed RAM when enough tasks have been assigned to the processor, otherwise, simple flip-flops are implemented. The size of the flags' memory requirement is number_of_tasks x 6. Table 2-3 shows the number of slices per number of tasks.

The flag set of the RTP processor includes SIGN, OVERFLOW, and CARRY for signed operations. UOVERFLOW and UCARRY for unsigned operations and ZERO for both signed and unsigned operations. These flags are generated in the execution block of the processor and used in a variety of ways. For example, if the *be* instruction is executed, it looks at the Zero flag to see if the result of the previous arithmetic instruction was zero indicating an equality. Any add, subtract, multiply, compare, and, or, xor, or shift operation will modify the flag set. The set will not change with jumps or other data movement commands.

Table 2-3: Flag register set slice count per task

Number of Tasks	Synthesis Slices	Post Implementation	Post Implementation SliceM
1	4	0*	0*
2	7	11	
3	10	20	
4	20	27	
5	29	40	
6	33	46	
7	36	53	
8	40	59	
9	49	78	12
10	54	85	12
11	56	91	12
12	59	98	12
13	69	105	12
14	73	111	12
15	79	119	12
16	87	128	14

* The implementation tools placed all of the required flip-flops into IOBs.
Slice count is actually 4

The OVERFLOW and UOVERFLOW flags are tied low for logical operations. The OVERFLOW flag for signed operations in the adder/subtractor unit is the logical XOR of the carry in and carry out bits of the full-adder circuit which implements the highest order bit. The UOVERFLOW flag for unsigned operations in the adder/subtractor unit is the carry out bit of the same full-adder circuit. The OVERFLOW flag from the multiplier/shifter signifies that the result of the multiply or the shift operation has overflowed into the top half of the output from the multiplier. For signed operations, it compares all of the upper 16 bits (bits 16 through 31) of the result against the most significant bit of the lower half of the result (bit 15). If all of the bits are 1 or all the bits are 0, the OVERFLOW flag is not set. If any two of the bits differ, the OVERFLOW flag is set. The UOVERFLOW flag is set if any bit in the upper half of the

result (bits 16 through 31) is non-zero. Special precautions are taken for unsigned operations in order to ensure that the result is not mistakenly skewed negative.

Probably the most interesting flag is the CARRY flag. The CARRY flag indicates that there was a carry out of the arithmetic unit (adder/subtractor). This flag allows the arithmetic operations on numbers greater than 16-bits through the use of multiple instructions and special instructions that take advantage of the CARRY bit. These instructions are post-fixed with the letter *c,* such as *addc*, *subc,* etc. The logical *c* instructions are for re-calculating flags across multiple instructions. *orc* will only set the zero flag if the previous result was zero as well. For a full list of the instruction set, please refer to [19].

While it is somewhat trivial to provide more than 16-bit arithmetic support for adds and subtracts, the multiplies and shifts are a little different. They require a full 16-bit 'carry out' register called the "T-reg" that is discussed below.

2.4 Processor Arithmetic

The arithmetic/logical section of the RTP is broken into three main units: the adder/subtractor unit, the shifter/multiplier, and the logical operations unit. The adder/subtractor unit is sixteen bits wide with the single carry out bit. Depending on the instruction, this bit will be latched as part of the flag set for future use. The *addc* and *subc* operations use this bit to support arithmetic operations of more than 16 bits. The adder/subtactor also completes the compare operation, which does not modify any of the registers, but does modify the flags.

The shifter/multiplier takes two sixteen bit inputs and produces a sixteen bit result and a sixteen bit overflow into a temporary register called the T-Reg. The heart of this unit is an 18 x 18 “hard” multiplier that produces a thirty-six bit result. The term “hard” signifies that the multiplier circuit is constructed from non-configurable CMOS circuitry, rather than being implemented from reconfigurable FPGA fabric. For signed multiplication, the sign of the sixteen bit inputs is extended to the full 18 bits of the hard multiplier. The sign of the output is taken from the most significant bit of the output of the multiplier. The T-Reg can be saved for later use when performing arithmetic operations of more than 16 bits. The following code example shows how to complete the multi-word multiplications using the multiplier, T-reg and the *xch* instruction.

```
                    R2R1
                    R4R3x

0      0      R3R1h R3R1l
0      R3R2h  R3R2l 0
0      R4R1h  R4R1l 0
R4R2h  R4R2l  0      0      +
R4     R3     R2     R1

;Make sure all used registers are cleared.
mov R5, R3
mul R5, R1  ; R5 gets R3R1l
xch R6, T   ; R6 gets R3R1h
mov R7, R3
mul R7, R2  ; R7 gets R3R2l
```

```
xch R8, T   ; R8 gets R3R2h
mov R9, R4
mul R9, R1  ; R9 gets R4R1l
xch R10, T  ; R10 gets R4R1h
mov R11, R4
mul R11, R2 ; R11 gets R4R2l
xch R12, T  ; R12 gets R4R2h
;Do the adds now
mov R1, R5  ; R1 gets add of first column (trivial)
mov R2, R6  ;
mov R3, R8  ;
mov R4, R12 ;
add R2, R7  ; Intermediate add of (R3R1h + R3R2l)
addc R3, R10; Intermediate add of (R3R2h + R4R1h + carry out)
addc R4, 0  ; add the carry out of previous operation to R4,
            ; will not carry out.
add R2, R9  ; R2 now has full addition of second column
addc R3, R11; R3 now has full third column plus carries of second
addc R4, 0  ; R4 now has full fourth column plus carries of third
;Result of multiplication is in registers R4, R3, R2, R1
```

Another novelty of this implementation comes not from the multiplication operations, but rather from the shift operations that this unit performs. A barrel shifter is very costly in terms of FPGA resources. A variable shift register is constructed from large multiplexers. A variable shift limited to 4 bits requires four 4-bit multiplexers. A variable shift limited to 5 bits requires five 5 bit multiplexers. For 16-bit shifts, sixteen 16-bit multiplexers are required. Because FPGA resources are made up of multiplexers

and memory cells, one might be led to believe that it is cheap to implement a large multiplexer in FPGA fabric. This, however, is not the case. The most basic FPGA resource is a LUT or look-up table. The LUT is a four input – one output memory cell; it can implement any 4 input function. According to Shannon's expansion theorem, with the addition of another LUT and a multiplexer (muxF5 in Xilinx's case), any 5-bit function can be implemented.

While this proves that as long as an FPGA is large enough, it can perform any combinatorial function, it also shows that the number of LUTs required increases exponentially for each additional input. The number of inputs required for a variable shift of up to 16 bits is minimally 20. If shifts in either direction are required, one more input is required. Yet another bit is required if the shifts can be both arithmetic and logical. Special circuitry exists for single shifts in the FPGA circuitry. However, this circuitry cannot be used to perform variable shifts within a single clock cycle. In order to perform rotating shifts of variable length in a single clock cycle, a large amount of FPGA resources are consumed. 128 LUTs and 16 F5muxes (occupying 34 slices) are required to perform a completely variable 16-bit rotating shift. In order to perform optional rotating shifts in both directions using either arithmetic or logical shifts, 212 LUTs and 13 F5muxes (occupying 120 slices) are required.

An alternative is to use the multiplier to perform shift operations. In ASIC designs, it is not uncommon to perform multiply operations by shifting. In this case we've reversed the operation by using multiplication to perform shifts. A shift of one place to the left is the same as multiplying the operand by 2; a shift of two places to the left is the same as multiplying by 4; a shift of *n* places to the left is the same as

multiplying by 2^n. Right shifting is the same as dividing; however, there is no hard divider in the Spartan 3 line of FPGAs. Right shifts are performed by shifting to the left by 16-*n*, or rather multiplying by 2^{16-n}, and reading the result from the top 16-bits of the 32-bit result. The multiplier actually produces a 36-bit result so we use bits 31 down to 16 for the result of the right shift operation.

Because shifts can overflow by up to 15 bits, the temporary register (T-Reg) that is used as the overflow register for the multiply operations is used to hold the overflow of the shifts. For right-shifts, the T-Reg is written with the lower 16-bits of the result. Using the exchange instruction (*xch*), the T-Reg can be moved to a normal operating register. Although the T-Reg is maintained across context switches, the T-Reg is not maintained across all instructions of the same task. If the designer wishes to later use the contents of the T-Reg, it is recommended that the contents of the register be saved immediately. The T-Reg is used for shift with carry operations as well as for multiplications. In a shift with carry operation, the existing data in the T-Reg is OR-ed into the result of the shift, while the "overflow" of the shift is stored into the temporary register. This operation makes data manipulation on variables larger than 16 bits possible. The following two code examples show how the shift with carry is used to complete both multi-word shifts and rotates:

```
;R3R2R1 shift left by R4, Result in R4R3R2R1
sll R1, R4  ;shift R1 by R4
slc R2, R4  ;shift R2 by R4
slc R3, R4  ;shift R3 by R4
xch R4, T   ;Move overflow from R3 into R4
```

```
;R3R2R1 rotate right by R4, Result in R3R2R1
srl R3, R4  ;shift R3 right by R4
src R2, R4  ;shift R2 right by R4
src R1, R4  ;shift R1 right by R4
xch R5, T   ;store overflow into R5
or R3, R5   ;OR overflow from R1 into R3 completing the rotate
```

The rotate operations are also completed in this functional unit. The rotate is accomplished by performing the requested shift and then OR-ing the two halves of the 32-bit result together. This effectively produces a rotated shift.

The logical unit of the data path performs the logical operations such as *and*, *or*, and *xor*. There are also logical operations, *andc*, *orc*, and *xorc* which utilize the carry. The "with carry" version of these operations uses the previously generated set of flags, as well as the flags generated from this operation, in calculating the new set of flags. This is specifically helpful when manipulating data larger than 16 bits.

3 Scheduler

The purpose of a scheduler in an RTOS is to provide the processing unit with the address of task code that is ready to execute. The term "ready" implies that all inputs necessary to execute the code properly are present. There are no outstanding dependences that would otherwise prohibit the execution of task code. For example, if a task were to only run every millisecond, the dependence is the millisecond tick; the task must wait for the tick in order to execute. A hardware implementation of a scheduler is not different. Its sole purpose is to provide the processor with code (or at least the address of code) that is ready to execute. Many different schemes have been proposed and used as scheduling algorithms.[17] [24]

One of the simplest schemes is round-robin scheduling. In the round-robin scheme, each task is granted a window of execution time. During the tasks allotted time, it has exclusive rights to execute on the processor. If the task is not ready to run, it must busy-wait until it either becomes ready or its allotted time is up (effectively wasting the processor cycles). Variations of the round-robin scheme exist in which only the ready tasks are granted a window of execution.

Another scheme is priority-based scheduling. In this scheme, each task is assigned a priority. The ready task with the highest priority is granted exclusive rights to the processor. When this task is no longer ready, the use of the processor is given to the

ready task with the next highest priority. In a scheduling scheme capable of preemption, when a higher priority task becomes ready while a lower priority task is running, the lower priority task will not be allowed to continue its execution. If, however, the scheme is not preemptive, the lower priority task will be able to continue its execution until it blocks or stops itself from running. Using priority-based scheduling with preemption, lower priority tasks may experience a condition called starvation, where the higher priority tasks utilize all of the available cycles of the processor thereby preventing the lower priority tasks from running. However, with proper design, priority-based scheduling can be very powerful.

Another approach that tries to handle the problem of starvation is the earliest deadline first scheme. In this scheme, the deadline by which a task must run its code must be known, or at least estimated. The rights to the processor are then granted to the task with the earliest deadline. Every task will eventually get to the point where its deadline is the closest. This type of scheduling makes sense for periodic tasks. However, in reality, it is often difficult to determine what deadlines exist for each piece of task code at runtime.

Implementing a scheduler in hardware is not a new idea. In [17] a hardware scheduler is proposed that implements two of the above mentioned schemes (priority-based, and earliest deadline first) plus a third (rate-monotonic) and it can dynamically switch between them. The implementation includes sorted priority and sleep queues, a task table, an interrupt controller and control logic all implemented in hardware. The task table holds information about the task, including the task priority, the period, worst case execution time, type (periodic or aperiodic), whether the task can be preempted or not,

and its status. The scheduler can be controlled either by specialized instructions from the processor or as an I/O device. To signal the processor to switch tasks, the scheduler uses an interrupt. A software routine reads the ID of the task scheduled to run, stores the context of the currently running task and switches context to the task scheduled to run.

The scheduler implemented here as part of the RTP architecture is a priority-based preemptive scheduler. It is capable of handling dynamic changes to priority, thus supporting priority inheritance as a solution to the priority inversion problem. To my knowledge, this is the first dynamic priority scheduler implemented in hardware. Priority inversion occurs when a high priority task is kept from running when a low priority task holds a resource that the high priority task needs.

This scheduler is implemented in hardware. The scheduler requires fixed task IDs, the hardware is optimized for this. The circuit could be further optimized to a simple carry chain if task priorities were fixed also.

The scheduler itself has a simple interface. The priority of the existing tasks and the ready status are used as inputs. The only output of the scheduler is the task ID of the highest priority ready task. Unused inputs, belonging to tasks that do not exist, are tied low. Most synthesizers have the ability to propagate constants and remove unnecessary logic.

Table 3-1 shows the number of slices used in the scheduler per the number of tasks. The case with only 1 task is trivial; it is only a wire. With only two tasks, one being the idle task with a fixed priority, the implementation is trivial as well; the result is still a wire that is the ready bit of the non-idle task.

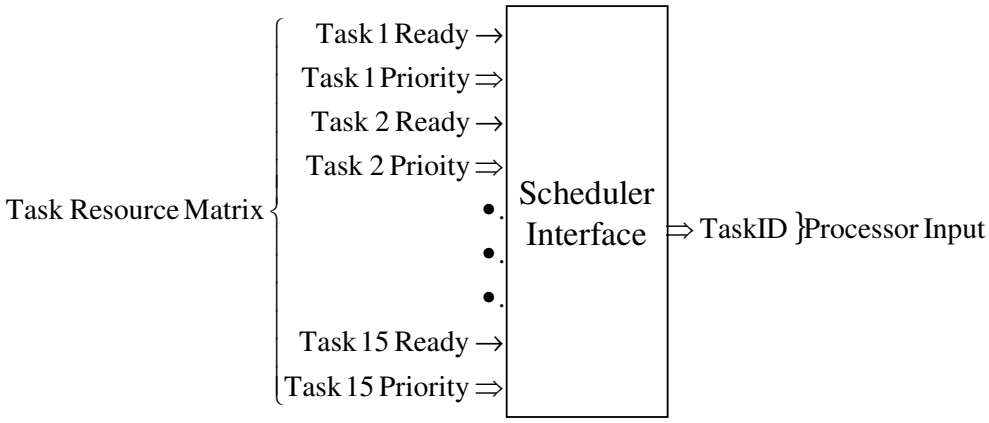

Figure 3-1: Scheduler Interface

Table 3-1: Scheduler slice count per number of tasks

Number of Tasks	Synthesis Slices	Post Implementation
1	0	0
2	0	0
3	8	6
4	12	12
5	23	21
6	29	26
7	37	31
8	42	36
9	50	40
10	55	45
11	65	52
12	71	59
13	74	60
14	83	66
15	88	71
16	96	79

The scheduler can be effectively disabled by the processor with the execution of the disable scheduler (DS) instruction. The task ID input is multiplexed with the previous scheduler input. After a DS instruction, the task ID input is ignored and the previous

input is used. The scheduler can be re-enabled with the enable scheduler (ES) instruction or certain TRM access instructions that should possibly block the execution of the task.

The internal workings of the scheduler take advantage of the fixed task ID for each task. The structure is a comparison tree that should identify the highest priority ready task. The full tree is necessary to implement priority inheritance; otherwise, with static priority assignment, a simple priority encoder would suffice. In order to shorten the propagation time, the ready bit is prefixed to the priority before a comparison is made. This allows the special carry chain logic of the FPGA to be used to give precedence to a lower priority ready task over a high priority non-ready task. By having the ready bit as the most significant bit for the comparison, any ready task is automatically given higher priority than any non-ready task. The alternative is to force a change before the selection is finalized, but that would require an additional stage of delay through a regular LUT, which is slower than the special carry hardware.

There are four levels of logic or stages in the scheduler. The first stage of the scheduler compares the priority associated with tasks 15 against 14, 13 against 12, 11 against 10, 9 against 8, 7 against 6, 5 against 4, 3 against 2 and tasks 1 against 0. Task 1 against 0 is a special case as task 0 is the idle task, which is always ready, and always has a priority of 0. If task 2's priority is greater than task 3's, task 2's ready state and priority pass on to the next stage along with the bit that separates 2 from 3. Task 2 can have a greater priority when it is ready and 3 is not, or when both are ready and task 2 has inherited the priority of a higher priority task (as part of the priority inheritance scheme). As an optimization, we can take note that the task IDs of the tasks that are compared differ only in the least significant bit. Therefore, it is only the least significant bit of the

highest priority ready task that is decided during the first stage. This bit is actually the output of the comparator computing A ≥ B (the carry out of the function A-B). Figure 3-2 shows the schematic for this comparator cell.

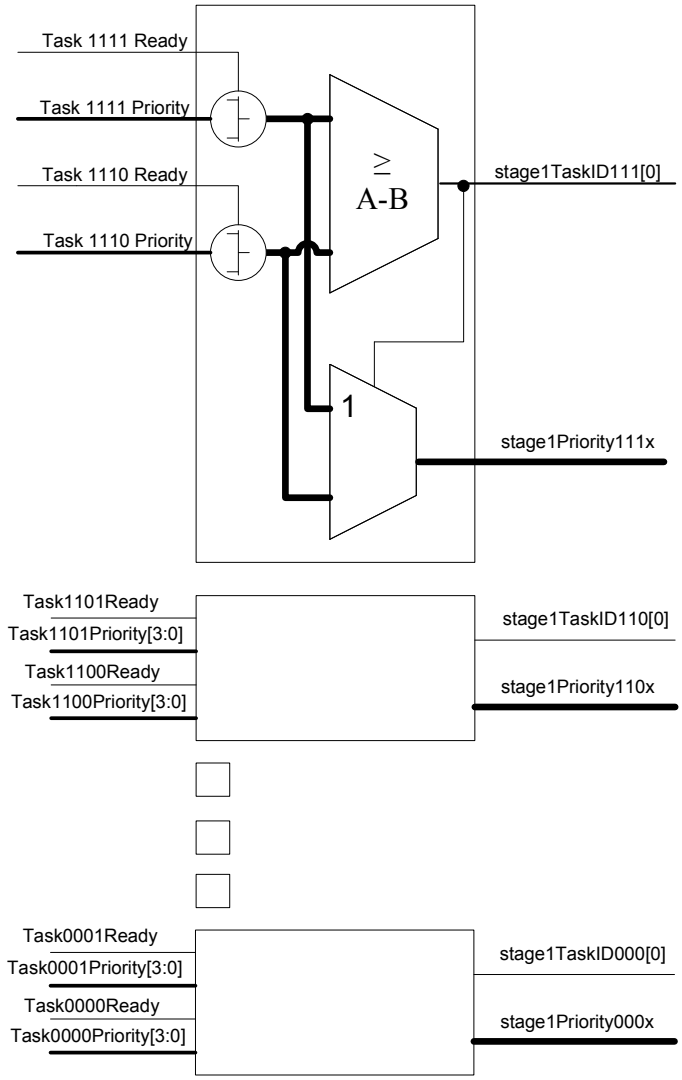

Figure 3-2: First Stage of the Scheduler Comparator Tree (8 total cells)

The second stage computes the higher priority of the stage 1 outputs. The higher priority of tasks 15 and 14 is compared with the higher priority of tasks 13 and 12. Note

that the task ID of the higher priority task between tasks 15 and 14 is *111x* where *x* is decided by the output of the stage 1 comparator. This is similar for all the other outputs of the stage 1 comparators. All of the task IDs are of the form *nnnx* where *nnn* is the

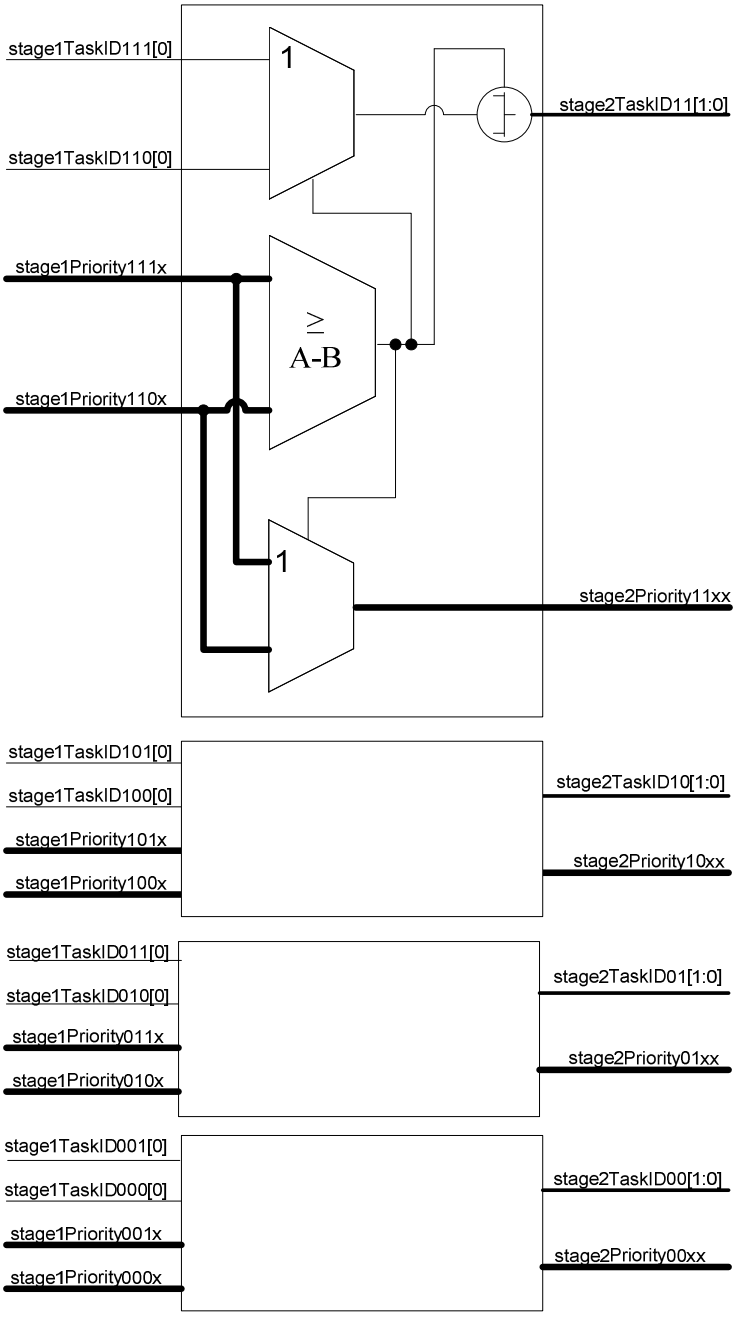

Figure 3-3: Stage 2 of the Scheduler Comparator Tree

most significant bits of the task ID and *x* is the bit determined by the comparator. The comparison of the higher priority of tasks 15 and 14 against tasks 13 and 12 compares the priority of tasks *111x* and *110x*. The second least significant bit is determined in the second round of comparison. The least significant bit then needs to be multiplexed through the second stage onto the third. The circuit of this stage is found in Figure3-3.

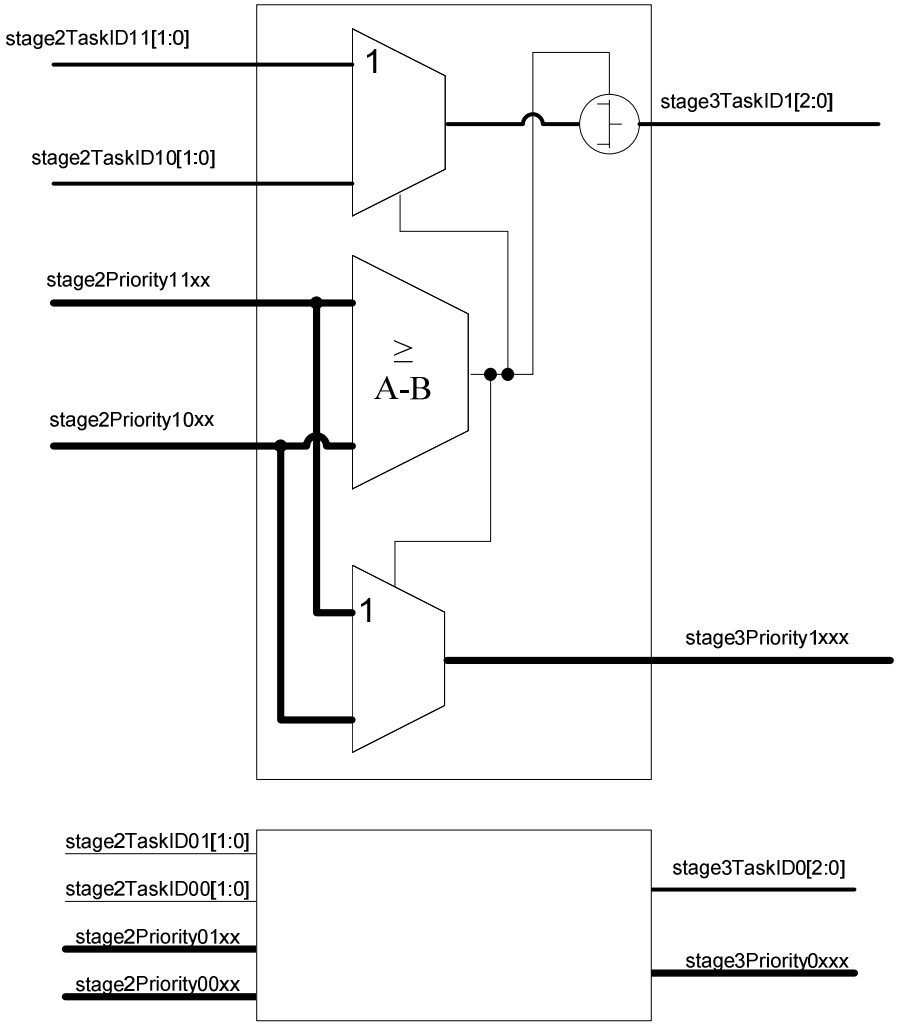

Figure 3-4: Stage 3 of the Scheduler Comparator Tree.

Similar to the stage 2 cell, the stage 3 cells compare the priorities of the highest priority of tasks 15, 14, 13 and 12 to the highest of 11, 10, 9, and 8, with task IDs of *11xx* and *10xx* respectively. The highest priority of tasks 7, 6, 5, and 4 is compared with the highest of tasks 3, 2, 1, and 0 with task ids of *01xx* and *00xx* where *xx* is determined by the first and second stages of the scheduler. The third least significant bit of the task with the highest priority is computed in this stage. The output of the comparator in this stage is the third least significant bit of the task ID. Figure 3-4 gives a schematic for the stage 3 cell of the scheduler.

The final stage of the scheduler compares the task's priority of the highest priority of tasks 8 through 15, with the task's priority of the highest priority of tasks 0 through 7 with task IDs of the form *1xxx* and *0xxx* respectively where *xxx* is determined by stages 1 through 3. The output of this comparator gives the most significant bit of the highest priority ready task. Figure 3-5 shows the schematic of this stage.

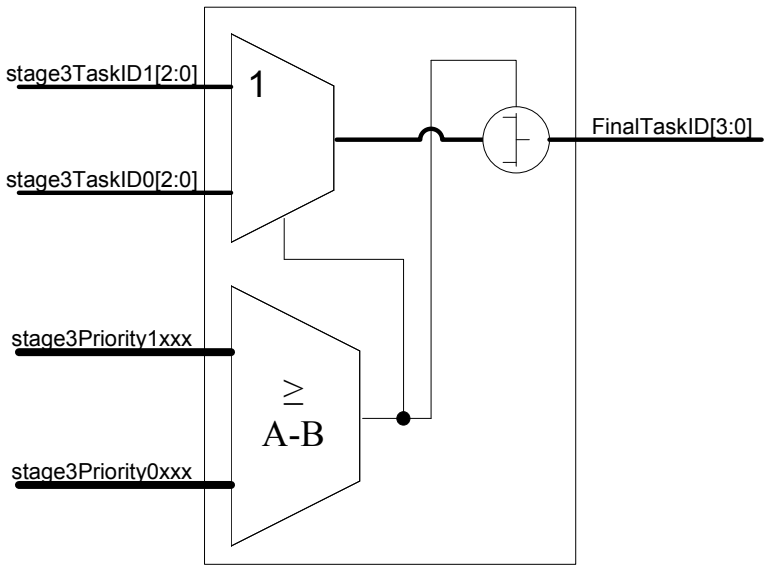

Figure 3-5: Final Stage of the Scheduler Comparator Tree

In [17] a hardware scheduler is proposed that performs priority based, rate-monotonic, and earliest deadline first algorithms and can be dynamically switch between the different methods. In order to keep the size of the RTP scheduler small (using minimum resources on an FPGA) and to minimize the propagation delays, only one method of scheduling is available. It would not be difficult, however, to modify the scheduler to be able to perform modified round robin and earliest deadline first algorithms as well as the priority-based system. Priority-based scheduling makes a lot of sense in this system because the tasks are already assigned an initial priority at compile time. Using the hardware primitives to block execution will prohibit a traditional "round-robin" scheduling algorithm since each task must run during its window of execution and the hardware primitives do not support busy waiting.

Rate monotonic scheduling can also be performed on this system since in its most basic form it relies on priority-based scheduling. The detailed work of scheduling using the rate monotonic algorithm would need to be performed at or before compile time. It would be necessary to know the periodicity, and deadlines. The theoretical algorithm requires neither data dependencies nor resource sharing, thereby making it almost completely useless. In practice, however, rate-monotonic scheduling is often augmented with priority inheritance in order to solve the priority inversion problem. The hardware scheduler implemented as part of the RTP project is capable of priority inheritance. The task of assigning priorities could theoretically be dynamic; however, the RTOS code would have to be augmented to calculate and assign the priorities without taking advantage of the hardware-only implementation of the scheduler. In order to take full

advantage of the hardware-implemented scheduler, the initial priorities would need to be determined at or before compile time.

In a multi-processor system, a unique scheduler must be added for each processor. The scheduler only has inputs for those tasks which are assigned to its processor and only outputs one task ID for the processor assigned to it.

4 The Task-Resource Matrix (TRM)

As part of the RTP project, a Task-Resource Matrix was designed and implemented. The Task-Resource Matrix (TRM) provides the system with control over shared resources. The TRM has connections to the scheduler to provide the priority and ready status of every task. The TRM takes inputs from the processor(s) to control it. The task-resource matrix is a distinguishing feature of the RTP project. A peer-reviewed paper was published and presented as [19]. Much of the contents of this chapter are taken from that paper. Designing a custom processor also allows tailoring of the instruction set to allow instructions which interface with custom RTOS hardware much more efficiently than would be possible with a standard instruction set. The resource-specific control instructions in this new architecture include the following:

- *lock r*: Attempt to reserve system resource *r*. If unsuccessful, the task is blocked until the resource becomes available. This instruction will automatically enable the scheduler input.
- *nb-lock r*: (non-blocking lock) 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 whether or not it successfully locked the resource, or if not, what other task has locked the resource.

- *rel r*: Release system resource *r* so other tasks may reserve it.
- *rst r*: Reset system resource *r* to initial state.

The meaning of the following instruction varies for each resource type:

- *enable r*: Enable system resource *r*.
- *disable r*: Disable system resource *r*.
- *sig1 r*: Send signal 1 to system resource *r*.
- *sig2 r*: Send signal 2 to system resource *r*.
- *read r*: Read the status of system resource *r*.
- *write r*: Write to the status of system resource *r*.

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 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.

These custom instructions facilitate hardware assistance to the RTOS, resulting in a much smaller software kernel. The software kernel has been reduced to around 400 instructions.

The Task-Resource Matrix is organized by rows and columns. The rows represent tasks interfacing with the TRM and the columns represent resources. A task sends a signal to the TRM to lock or reserve a resource for exclusive use. If the resource lock is granted, the task may continue executing and use the resource. If the lock is not

granted, the TRM signals the scheduler to block that task from executing. Additionally, a peripheral may send status information to the TRM. For example, if a queue resource is full, it signals the TRM not to grant access until the queue no longer sends the full flag. The resource nodes in the TRM keep request/grant state of a task for a particular resource.

4.1 Types of Resource Modules

The resource modules record the state of the resources they represent. In order to simplify system generation, a standard resource module interface is used. Figure 4-1 shows this interface.

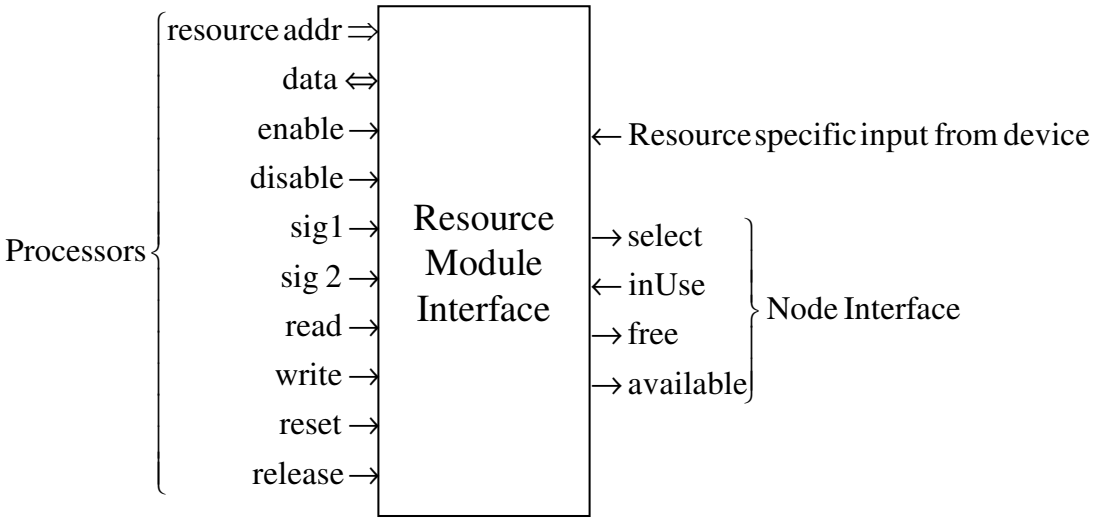

Figure 4-1: The Resource Module Interface

The interface includes signals from the processor, any resource specific inputs and an interface with the resource nodes. The resource address specifies which resource is targeted; the data bus transfers any resource specific information. The remaining signals originating from the processor are control signals asserted directly as a result of executing hardware control instructions previously discussed. The signals *free* and *available* indicate that this resource can be locked by another task. The *inUse* signal indicates that one or more tasks have locked the resource. The select signal is broadcast to all nodes corresponding to this resource. It signifies that this resource is targeted for action by the processor. The resource specific input from a peripheral varies depending on the resource. Not all of the connections are used with every resource; some of the signals may not have meaning.

The following resource modules are implemented for use as part of the RTP project:

Mutex

A mutex can be declared for any shared device. Before accessing a shared device, a task must acquire the mutex lock. Once locked, the task has exclusive privileges to the shared resource. When access to the resource is no longer required, the task releases the mutex lock.

Event

An event is a set of flags that enables the granting of a lock. A conjunctive event requires all of the flags to be set. A disjunctive event requires any one of the flags to be set.

When an event is detected, a lock will be granted to the highest-priority requesting task. The task can then clear the event flags.

Semaphore

This resource is associated with a counter. This resource may be locked (and the count atomically decremented) by multiple tasks as long as the counter is strictly positive (≥ 1). Tasks can also initialize the counter.

Interrupt

Using the task-resource matrix, interrupts are serviced like any other event in the RTP system. On system initialization, an interrupt service routine is initialized and then blocks, waiting for an interrupt. When an interrupt event occurs, the interrupt lock is granted and the service routine proceeds. The interrupt routine releases the lock (re-arming the interrupt), services the interrupt, and then tries to re-acquire the lock, thereby blocking until the next interrupt event.

Reader

Before reading a queue (or mailbox, etc.), a task must acquire a read lock. This lock is only granted if the queue is not empty. Once locked, the task has exclusive read privileges and can read multiple words. When done, the task releases the read lock.

Writer

Before writing a queue (or mailbox, etc.), a task must acquire a write lock. This lock is only granted if the queue is not full. Once locked, the task has exclusive write privileges and can write multiple words. When done, the task releases the write lock.

Global Timer

This resource may be locked by multiple tasks if the timer is zero. The timer may be read and written. Tasks that try to lock a non-zero timer will block until it decrements to zero. It is decremented every millisecond.

4.2 The Task Modules

The task module contains information such as its task-processor id, priority, ready state, and timeout counter. The timeout mechanism allows a task to set a time-limit for which it will wait for a resource. This timeout counter can also be used as a suspend resource for each task. Using this counter, a task can suspend itself for a specified amount of time or indefinitely. The priority can be dynamically changed to support operations such as priority inheritance to solve the priority inversion problem. Figure 4-2 shows the connections to a task module.

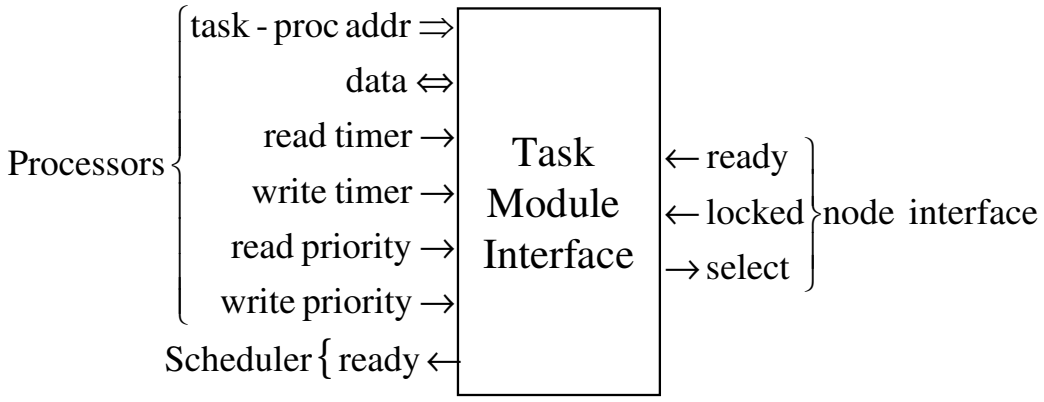

Figure 4-2: The Task Module Interface

The task module interface is also standardized in order to facilitate system generation. The processor specific connections include a task-processor id that addresses the correct task module for use, and a data bus. The remaining signals originating from the processor are control signals asserted as a result of executing hardware control instructions previously discussed. The node interface includes *ready*, *locked* and *select* signals. The *ready* signal alerts the task module and subsequently the scheduler that there are no outstanding resource requests that block execution. The *locked* signal is a logical ORing of all the selected locks in a task row. The *locked* signal is used by the nb_lock instruction to identify which task holds a resource. The currently executing task determines the *select* signal that is used to select a task row for modification by the processor.

4.3 The Resource Node

The resource node is found at intersections of task rows and resource columns. The nodes are sparsely populated and only placed where a task utilizes a resource. In this way, only useful circuitry is built. Figure 4-3 shows the resource node interface.

The *inUse* signal communicates to the resource module the number of resource nodes that have this resource locked. The *free* signal originates in the resource module and communicates to the resource nodes whether or not this resource is free to lock. It also prioritizes which task is granted access when multiple tasks are requesting the resource.

The *ready* and *locked* signals chain together nodes in the same task row. *Ready* is the logical NAND of all the request flags in the row and is used to remove the task from

the ready queue. *Locked* is a logical OR of the selected grants used by the nb_lock instruction to identify which task holds a resource. The confluence of the task and resource select lines enables a node state change.

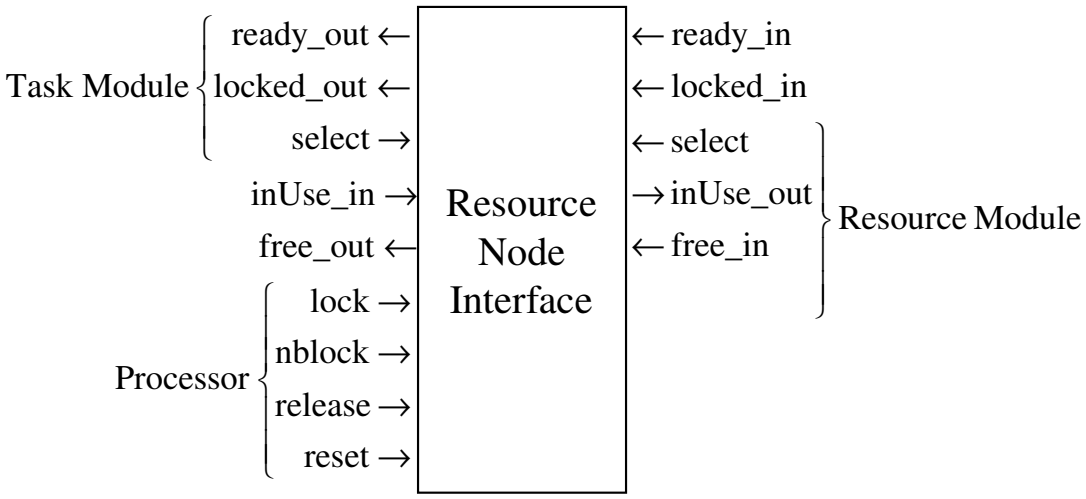

Figure 4-3: The Resource Node

The resource node keeps track of pending requests and grants for a resource. The node maintains the state by keeping a request flag and a grant flag. These flags are state variables of a state machine internal to the resource node. Figure 4-4 graphically shows the state machine of most resource nodes. Control signals such as *lock*, *nb_lock*, and *release* modify the state of the node. On a *lock*, the request flag is set. The task will be blocked until the lock is granted (i.e. the request flag is reset and the grant flag is set) or until its timeout counter expires. The request flag is used in combination with the *ready_in* signal to generate the *ready_out* which will eventually propagate to the task module. While the request flag is asserted, the node will signal the task module and the

scheduler to remove the task from the ready queue. If the resource is free to acquire, the lock is immediately granted without stalling the task. On an *nb_lock* signal, the task will never stall. The state of the resource and the task ID of the task which has locked the resource are returned to the processor. If the task is free to acquire, the grant flag will be set. If not, there is no change in state. A *release* signal releases the lock held by the task by resetting the grant flag. The *reset* signal resets both the request and grant flags.

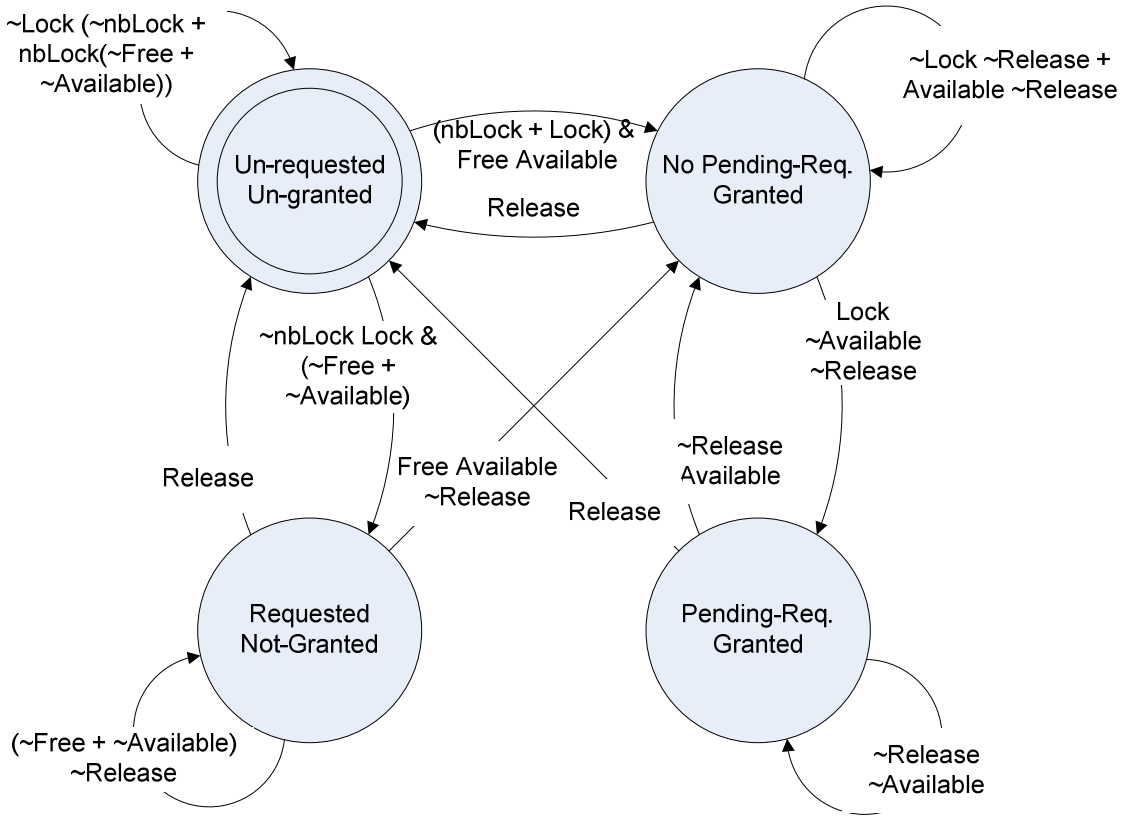

Figure 4-4: State Machine of the Resource Node.

There are three main types of resource nodes: a single acquisition node, a multiple acquisition node, and an interrupt node. A single acquisition node assures that only one

task maintains ownership of any given resource at any given time. A multiple acquisition node allows multiple locks to be granted. While the first two types of resource node differ only in the connections between nodes and modules, the third type of resource node is markedly different and serves a different purpose. The interrupt node's state machine is simpler than the state machine in Figure 4-4.

The interrupt node should only be used by tasks that are Interrupt Service Routines (ISR). In general, ISRs should not depend on shared resources. A traditional interrupt event typically triggers a jump to an ISR address in the interrupt vector table. The traditional ISR saves context, does some sort of decoding to ascertain what event triggered the interrupt request (IRQ) and services the interrupt. In the RTP architecture, an IRQ is handled by an interrupt resource. The task that is designated an ISR for a particular IRQ depends on the interrupt resource associated with that IRQ. On startup, the ISR runs initialization code and then attempts to lock the interrupt resource. Unless the interrupt event has already occurred, the ISR will block. On an interrupt event, the ISR unblocks, re-arms the interrupt resource module and runs until it retries to lock the interrupt resource. If an interrupt occurs while the ISR is running, the resource module remembers an event has occurred and immediately grants the lock. There is no interrupt vector table; there is no saving and restoring of context. Using this method of interrupt handling within the scope of the RTP architecture allows a designer to guarantee worst-case interrupt latency.

The resource modules and resource nodes vary for each type of resource. However, the interconnections between them are standard. The task modules also

interface with the resource nodes in a standard fashion, although some of the connections may not be used. The goal of standard connections is to simplify system generation.

The combination of resource module, task module, and resource node allows fast inter-processor communication to take place. For example, a typical mutex release may result in a context switch. The entire process of task switching from mutex release to a new task running requires only 4 clock cycles. During the first cycle, a task sends a release command to the TRM targeting the specified mutex. On the next cycle, the release is processed, a pending task is granted the mutex, and the task module is signaled that the grant has been completed. Arbitration through the scheduler takes place during the third cycle. On the fourth cycle, instructions from the newly readied task are fetched for execution on the processor. From the point of view of the processor, no cycles are lost. During cycle three, instructions are fetched from the previous highest priority ready task, but during cycle 4, the task switch is complete; instructions are fetched to execute the new task.

5 Conclusions

The Real Time Project aims to create a predictable embedded architecture. Multiple register sets in hardware allow context switches to occur seamlessly in one clock cycle. The hardware scheduler can schedule each processor cycle with the highest priority ready task. RTOS primitives, controlled by RTP-specific operating instructions, move inter-task communication to hardware. The RTP provides predictable behavior by utilizing fast-context switching, RTOS primitives implemented in hardware, a hardware scheduler, and a new feature, the Task-Resource Matrix. Using this architecture, an embedded designer can predict with 100% accuracy the total number of clock cycles that will occur between when an interrupt occurs, to when that interrupt will be fully serviced. The embedded designer can know how many clock cycles from the point that the mutex release function is called, to when a higher priority task can start running, including all of the RTOS overhead. With the entire software portion of the RTOS contained within only around 405 instructions, most of the overhead of the RTOS is removed to hardware. This enables more processing cycles to be dedicated to useful computation, allowing the processor frequency to be slower but still being capable of meeting all of the real-time deadlines.

5.1 Limitations / Future Improvements

The current version of the RTP hardware has no debug capabilities built into it at this time. The primary method of software programmers to debug is to use “printf” or a similar method of printing out strings. While most embedded applications don’t have the luxury of a screen to print out debug statements, it is possible to use a debug UART to print out debug messages. This could be a UART or similar serial transfer protocol that could be included in DEBUG builds of the RTP system, but not in the production or release build. This would be an easy addition to the RTP architecture.

Initial developmental stages of debugging require better methods than using printf. After all, if your code or system doesn’t get to the point where a printf could be called, what good is it then? Systems design engineers, therefore, require better methods of debugging. There are many commercially available hardware debuggers and in-circuit emulators. Most of these, however, have specifically supported processors that they can interface to, even when implementing the JTAG protocols. There are some, however, that can work independently of the processor and simply pass messages from a controlling piece of software to the hardware. The Insight box is one of these JTAG “dumb” devices. In order to have efficient debugging capabilities into the RTP project, a significant overhead of software work for just the host PC would be required. This would provide an interface to the hardware that can be interpreted by the host PC and displayed in a user friendly format. The designer would not only want to view the currently active set of registers, but also all of the registers of any RTP processor to view the state of any task. Visibility into the TRM is of particular interest so that one could see any outstanding dependencies of any tasks. The hardware would need to be modified and

augmented to support a JTAG connection. The ability to "step" through code is a feature that every embedded design could benefit from. The ability to see how hardware reacts to every single line of code is crucial in debugging embedded systems. Except for the most sophisticated models, simulators lack the ability to accurately simulate an embedded system. While simulation can play an important role in determining if hardware will execute as designed, it falls short in being able to produce asynchronous hardware events, such as interrupts.

After the design has been implemented, it is often desirable to gather performance data on your system. To know whether tasks are being starved of processor cycles, or whether processors spend most of their time idle could help designers further customize their hardware.

An alternative to adding debug support to the RTP project may be to port the concept driving the RTP project to another commercially available processor. By using the new features incorporated into Xilinx's Microblaze® or Altera's NIOS II®, the hardware RTOS elements described earlier may be incorporated with specialized instructions. The NIOS II allows for up to 256 application-specific combinatorial, multi-cycle, or parameterized instructions. The drawback of this approach is that there will no longer be any fast-context switching and no support for multiple processors.

An extension to the RTP project may be to add an external memory controller. This will allow the RTP to address memory outside of the limitations of the FPGA where the RTP resides. This will also allow more than the normally addressable memory range of memory to be addressed (more than 16-bit addresses). This memory controller can be

made to handle address and data multiplexed (A/D mux) memory devices which are becoming more common in recent embedded designs.

Additionally, the program counter can easily be increased from 12-bits to 16-bits. This, however, makes it impossible to make a jump to any address in the memory range using a literal with a single instruction, but increases the memory space from 4k instructions to 64k. It is also possible to increase the program counter to more than 16 bits. Increasing the PC to more than 16 bits would require significant hardware modifications.

The scheduler can also be augmented to include different schemes of scheduling. It need not be dedicated to simply using a priority-based method, but can be increased to include a modified round-robin, earliest deadline first or even a rate-monotonic scheme.

Hardware and code acceleration pieces can be easily added to the RTP architecture. Queues and mailboxes should be implemented and controlled by the reader and writer resources. Some additional hardware can include a floating point arithmetic unit, to speed up floating point calculations. A "scratchpad" memory shared between processors could be used to extend the memory available to a single processor. Different networking cores could be implemented to increase the usability of the RTP architecture. Such protocols could include SPI and I²C for intra-PC board communications, or USB, Lontalk, or Ethernet for off-board communications.

5.2 Final Words

In conclusion, the RTP architecture is already a very powerful architecture. With some additions, this architecture could become a vehicle for further research.

6 References

[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] J. Adomat, J. Furunas, L. Lindh and J. Starner, "Real-Time Kernel in Hardware RTU: A Step Towards Deterministic and High-Performance Real-Time Systems," *Proceedings of EURWRTS '96*, pp. 164-168, June 1996.

[6] Realfast, http://www.realfast.se

[7] T. Nakano, Y. Komastudaira, A. Shiomi and M. Imai, "VLSI implementation of a Real Time Operating System," *Proc. of ASP-DAC '97*, pp.679-680, January 1997.

[8] 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.

[9] 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.

[10] T. Nacano, 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

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

[12] D. Sun, D. M. Blough, and V. J. Mooney III, *Atalanta: A New Multiprocessor RTOS Kernel for System-on-a-Chip Applications*, Technical Report, School of Electrical and Computer Engineering, Georgia Institute of Technology.

[13] 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

[14] H. Takada, S. Honda, R. Nishiyama, H. Yuyama, "Hardware/software co-configuration for multiprocessor SoPC**",** *IEEE Workshop on Software Technologies for Future Embedded Systems, (WSTFES'03)*, pp. 7-8, 15-16 May 2003.

[15] Altera, http://www.altera.com

[16] The Free Dictionary . com, http://encyclopedia.thefreedictionary.com/

[17] P. Kuacharoen, M. A. Shalan, Vincent J. Mooney III, *A Configurable Hardware Scheduler for Real-Time Systems,* Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms 2003

[18] S. W. Isaacson, D. K. Wilde, *The Task-Resource Matrix: Control for a Distributed Reconfigurable Multi-Processor Hardware RTOS,* (ERSA'04), pp.130-136

[19] R. S. Klinger, *Compilation and Generation of Multi-Processor On a ChipReal-Time Embedded Systems,* Master's Thesis, Brigham Young University*, 2007*

[20] ThreadX, http://www.rtos.com

[21] Microcross, http://www.microcross.com/html/ucos_benefits.html

[22] Wind River, http://www.windriver.com

[23] *ARM1176JZ-S r0p6 Technical Reference Manual*, ARM, http://www.arm.com, 2004-2007

[24] S. Saez, J. Via, A. Crespo, A. Garcia, *A Hardware Scheduler for Complex Real-Time Systems*, Proceedings of the International Symposium on Industrial Electronics, ISIE'99, 1999, vol. 1 pp. 43-48