

TABLE OF CONTENTS

1.0	Introduction	3
1.1	The RTX 2000, 2001A, And 2010 Microcontrollers	3
1.2	The RTX 2000 Family Programmer's Reference Manual	5
2.0	RTX Architecture	9
2.1	On-Chip Stacks	11
2.1.1	The Parameter Stack	11
2.1.2	The Return Stack	11
2.2	RTX 2000 Series Bus Architecture	12
2.2.1	Data Buses	12
2.2.2	Address Buses	13
2.3	Registers	14
2.3.1	Stack Related Registers	14
2.3.2	Status/Control Registers	14
2.3.3	Internal Processor Registers	14
2.4	Memory	15
3.0	Operations	19
3.1	Instruction Execution	19
3.2	Stack Operation	21
3.3	Subroutine Calls and Returns	22
3.4	Branching And Looping	23
3.5	Streamed Instructions	24
3.6	Math/Logic Operations	25
3.6.1	Registers And I/O Devices	26
3.6.2	Memory	26
3.6.3	Literals	26
3.7	Stack Operations	27
3.7.1	DUP	27
3.7.2	SWAP	28
3.7.3	DROP	28
3.7.4	OVER	29
3.7.5	>R	29
3.7.6	R>	30
3.7.7	R@	30
3.8	Interrupts	31
3.8.1	Maskable Interrupts	31
3.8.2	Non-Maskable Interrupts (NMI)	31
3.8.2.1	On the RTX 2000	31
3.8.2.2	On the RTX 2001A	32
3.8.2.3	On the RTX 2010	32

4.0	RTX Registers	35
4.1	Stack Related Registers	39
4.1.1	TOP Register (Parameter Stack)	39
4.1.2	NEXT Register (Parameter Stack)	39
4.1.3	I Register (Return Stack)	40
4.1.3.1	I At Address 00H	41
4.1.3.2	I At Address 01H	41
4.1.3.3	I At Address 02H (Stream Count/Loop Count)	42
4.1.4	IPR Register (Return Stack)	43
4.2	Internal Processor Registers	44
4.3	Control/Status Registers	45
4.3.1	The Configuration Register - Address 03H	46
4.3.2	The MD Register - Address 04H	47
4.3.2.1	MD On The RTX 2000 and RTX 2010	48
4.3.2.2	MD On The RTX 2001A	48
4.3.3	The SQ Register - Address 05H	49
4.3.4	The SR Register - Address 06H	49
4.3.5	The PC Register - Address 07H	50
4.3.6	The Interrupt Mask Register, IMR - Address 08H	51
4.3.7	The Stack Pointer Register, SPR - Address 09H	52
4.3.8	Address 0AH	52
4.3.8.1	On The RTX 2000	52
4.3.8.2	On The RTX 2001A and RTX 2010	52
4.3.9	Address 0BH: IVR, SVR, And SLR	53
4.3.9.1	Write-only On The RTX 2000: SLR	53
4.3.9.2	Write-only On The RTX 2001A: SVR	53
4.3.9.3	Write-only On The RTX 2010: SVR	54
4.3.10	Index Page Register - Address 0CH	55
4.3.11	Data Page Register, DPR - Address 0DH	56
4.3.12	User Page Register, UPR - Address 0EH	56
4.3.13	Code Page Register, CPR - Address 0FH	56
4.3.14	Interrupt Base/Control Register - Address 10H	57
4.3.15	User Base Register, UBR - Address 11H	57
4.3.16	Address 12H	58
4.3.16.1	On The RTX 2000 and RTX 2001A	58
4.3.16.2	On The RTX 2010: MXR	58
4.3.17	Timer/Counter 0 - Address 13H	59
4.3.18	Timer/Counter 1 - Address 14H	59
4.3.19	Timer/Counter 2 - Address 15H	59
4.3.20	Address 16H	60
4.3.20.1	RTX 2000 - MLR	60
4.3.20.2	RTX 2001A - RX, Scratchpad/Counting Register	60
4.3.20.3	RTX 2010 - MLR	60
4.3.21	Address 17H	61
4.3.21.1	RTX 2000 - MHR	61
4.3.21.2	RTX 2001A - RH	61
4.3.21.3	RTX 2010 - MHR	61

5.0	External Bus Interfaces	65
5.1	ASIC Bus Interface	66
5.1.1	RTX 2000 and RTX 2001A Extended Cycle Operation	67
5.1.2	RTX 2010 Extended Cycle Operation	68
5.2	Memory Interface	69
5.2.1	Code Memory Space	71
5.2.1.1	Subroutine Calls and Returns	71
5.2.1.2	Branching	73
5.2.2	Data Memory Space	74
5.2.2.1	Memory Page Selection	74
5.2.2.2	Memory Access Mode Selection	75
5.2.2.3	Memory Access Examples	76
5.2.3	User Memory Space	80
6.0	On-Chip Peripherals	85
6.1	Stack Controllers	85
6.1.1	Stack Pointer Operation	86
6.1.1.1	Stack Pointers For the RTX 2000	87
6.1.1.2	RTX 2001A and RTX 2010 Stack Pointers	89
6.1.2	Stack Limit Operation	91
6.1.2.1	Stack Limits For the RTX 2000	91
6.1.2.2	Stack Limits For the RTX 2001A	92
6.1.2.3	Stack Limits For the RTX 2010	93
6.1.3	Configuration Of Substacks	95
6.1.3.1	Substack Configuration On The RTX 2001A	95
6.1.3.2	Substack Configuration On The RTX 2010	98
6.1.4	Stack Error Conditions	102
6.1.4.1	RTX 2001A and RTX 2010 Fatal Stack Errors	102
6.2	Interrupt Controller	103
6.2.1	Interrupt Acknowledgement	107
6.2.2	Disabling Interrupts	108
6.2.3	Software Interrupt	109
6.3	On-Chip Hardware Math Support	110
6.3.1	RTX 2000 Multiplier Operation	110
6.3.2	RTX 2010 Hardware Math Support	113
6.3.2.1	RTX 2010 Multiplier/Accumulator Operation	113
6.3.2.2	RTX 2010 Barrel Shifter and LZD Operation	115
6.4	Counter/Timers	116
6.4.1	Counter/Timer Operation	116
6.4.2	Counter/Timer Interrupts	117
6.4.3	Clock Selection	118

7.0	Instruction Set	121
7.1	General Information	121
	7.1.1 Streamed Execution Mode	122
	7.1.2 The Auto-decrementing Loop Instruction	122
7.2	Format	123
7.3	Subroutine Call	127
7.4	Subroutine Return	128
7.5	Classes 8 and 9: Branches and Loops	129
7.6	Class 10: ALU Operations	136
	7.6.1 Carry Bit	137
	7.6.2 Shift Operations	138
7.7	Enhanced Processor-Specific Operations	153
	7.7.1 Streamed MAC Instructions On The RTX 2010	182
7.8	Class 11-a : ASIC Bus Access	184
	7.8.1 ASIC Bus Instructions	184
	7.8.2 Predefined ASIC Bus Instructions	191
7.9	Class 11b - Short Literals	196
7.10	Class 12: User Memory Access	201
7.11	Class 13: Long Literals	208
7.12	Classes 14 and 15: Data Memory Access	214
7.13	Undefined Opcodes	227
8.0	Step Math Functions	225
8.1	Introduction	225
	8.1.1 Step Math Using The RTX 2000	225
	8.1.2 Step Math Using The RTX 2001A	226
	8.1.3 Step Math Using The RTX 2010	226
8.2	Data Flow in Step Math	227
8.3	17-Bit Math	229
8.4	The Step Math Instruction Format	230
	8.4.1 ALU Micro Opcode Field (aaa)	231
	8.4.2 Register Selection Micro Opcode Field (r)	232
	8.4.3 Conditional Behavior Micro Opcode Field (yy)	233
	8.4.4 Subroutine Return Micro Opcode Field (R)	234
	8.4.5 Unconditional Shift Micro Opcode Field (sss)	235
	8.4.6 Signed/Unsigned Micro Opcode Field (S)	236
8.5	Operation of the 17th-Bit Adder	237
8.6	Interrupting Step Math Operations	238
8.7	Some Useful Opcodes	239
8.8	Step Multiplication	240
	8.8.1 Signed Step Multiplication	240
	8.8.2 Mixed Sign Multiplication Type A	242
	8.8.3 Unsigned Multiplication	243
	8.8.4 Mixed Sign Multiplication Type B	244
8.9	Step Division	245
	8.9.1 Standard Division Program	248
	8.9.2 Alternate Division Program	248

8.10	Step Square Root	249
8.11	Step Bit Reversal	251
8.12	Step Cyclic Redundancy Check (CRC)	255
8.13	Step Math Reference	258

CHAPTER 1

INTRODUCTION

1 Introduction

The Harris Real Time Express (RTX) 2000 Family of microcontrollers is a highly integrated family of 16-bit CMOS microcontrollers designed for real-time control systems requiring high performance with low power consumption.

1.1 The RTX 2000, 2001A, And 2010 Microcontrollers

The architecture of the RTX 2000 Series of products results in high instruction execution rates. The highly parallel architecture allows the RTX to perform several functions in one instruction cycle, and all instructions execute in either one or two clock cycles. Instructions are fetched from memory and executed immediately; there are no instruction "pipelines" or caches to flush when performing branches or calls.

The RTX 2000, 2001A and 2010 Microcontrollers have on-chip support hardware for performing many of the functions typically needed in a real-time system, including an interrupt controller, a memory page controller, two stack controllers, and three 16-bit counter/timers. In addition to these "on-chip peripherals", the RTX 2000 provides a 16-by-16 hardware multiplier, while the RTX 2010 provides a 16-by-16 hardware multiplier-accumulator along with a 32-bit Barrel Shifter and a 32-bit Leading Zero Detector for Floating Point support. Table 1.1 shows a break-out of the features of each of these products.

The RTX 2000 Class architecture was designed to execute the high-level language Forth as its "assembly language". The instruction set provides the features necessary for implementing much of the Forth language directly. Instructions are available for manipulating stacks, performing memory access, controlling program flow, and basic math and logic operations.

One RTX instruction may combine the functions of two or three high level Forth instructions, resulting in an effective processor throughput which is faster than the processor clock speed.

The stack oriented architecture of the RTX also makes it well suited for running such computer languages as C.

<u>RTX 2000</u>	<u>RTX 2001A</u>	<u>RTX 2010</u>
Interrupt Controller	Interrupt Controller	Interrupt Controller
Stack Controller	Stack Controller	Stack Controller
Two 256-Word Stacks	Two 64-Word Stacks	Two 256-Word Stacks
Three 16-Bit Timer/Counters	Three 16-Bit Timer/Counters	Three 16-Bit Timer/Counters
1-Cycle 16-Bit Multiplier		1-Cycle 16-Bit Mult./Accum.
		1-Cycle 32-Bit Barrel Shifter; Floating Point Support

TABLE 1.1: RTX On-Chip Hardware Peripherals

1.2 The RTX 2000 Family Programmer's Reference Manual

Figure 1.1 offers an overview of the interface between a user and an RTX Microcontroller. The documentation which supports each layer of this interface is also shown.

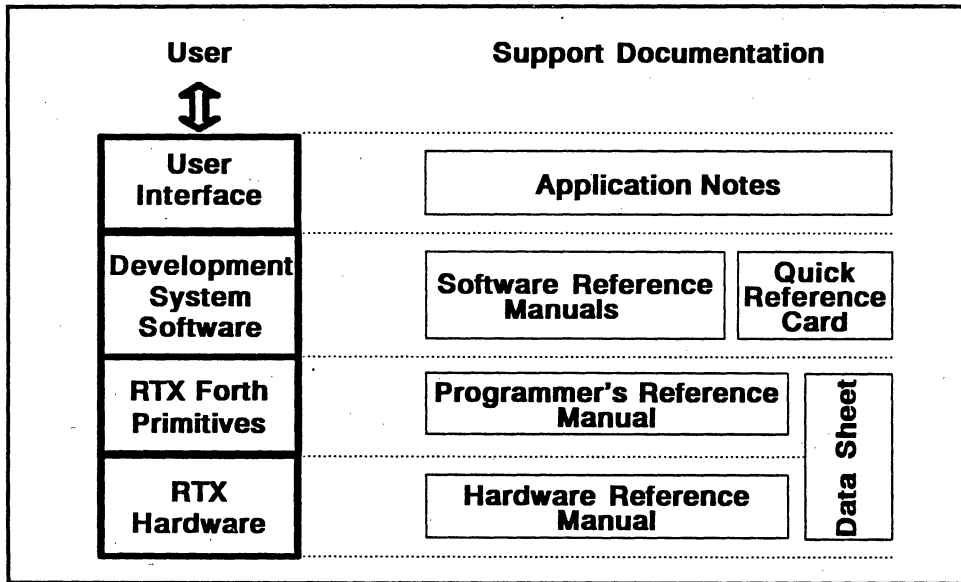


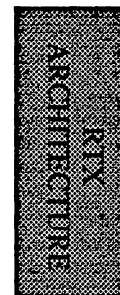
FIGURE 1.1: USER/RTX INTERFACE

The Programmer's Reference Manual describes the RTX 2000, RTX 2001A, and RTX 2010 Microcontrollers from a programmer's point of view, including architecture, registers, data paths, hardware interfaces, and primitive instructions. Topics described in various sections of this manual include:

- Chapter 2 Overall architecture of RTX microcontrollers
- Chapter 3 General operation of RTX microcontrollers
- Chapter 4 The RTX register set
- Chapter 5 Memory Interface
- Chapter 6 On-chip Peripheral Devices
- Chapter 7 RTX Instruction Set
- Chapter 8 Implementation of Multi-step Math Functions
- Chapter 9 Implementing Forth on the RTX
- Chapter 10 Code Optimization Techniques

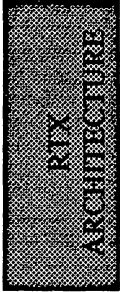
Some functional differences exist between the different members of this family of microcontrollers. When such differences exist, the applicable sections of this manual describe those differences. Where major differences exist, they are broken into separate paragraphs, and are offset with a side bar for clarification.

For additional information specific to your microcontroller, please refer to the appropriate data sheet.



CHAPTER 2

RTX ARCHITECTURE



2 RTX Architecture

This chapter provides an overview of the programmer's model of the RTX Microcontroller architecture. Figures 2.1, 2.2 and 2.3 show block diagrams for the RTX 2000, RTX 2001A, and RTX 2010 Microcontrollers respectively.

The RTX microcontroller is a stack based machine with two on-chip stacks. Most math, I/O and memory reference operations take their operands from the Parameter Stack, and leave their results on the Parameter Stack. Subroutine calls use the Return Stack for saving their return addresses.

There are twenty-three registers on the RTX 2000, twenty-four registers on the RTX 2001A, and twenty-five registers on the RTX 2010. These registers control processor configuration and status, hold intermediate results during computations, and provide an interface between the processor and its on-chip peripheral devices.

The RTX registers and stacks are interconnected through a series of 16-bit data buses which transfer data within the processor and with the outside world.

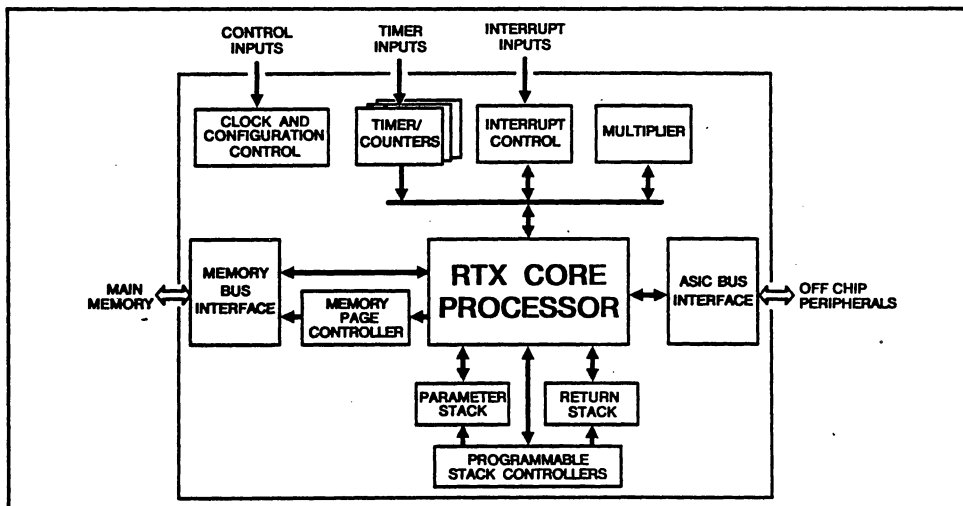
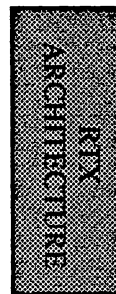


FIGURE 2.1: RTX 2000 BLOCK DIAGRAM



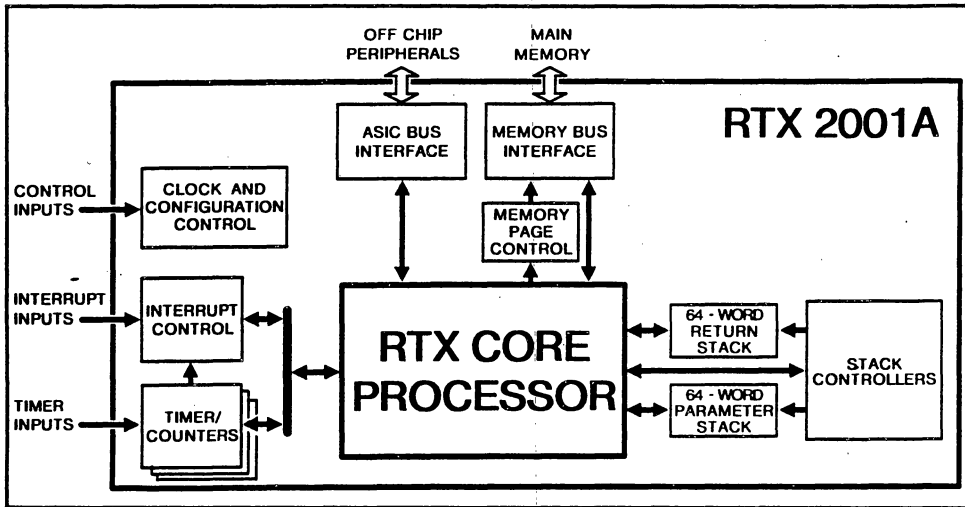


FIGURE 2.2: RTX 2001A BLOCK DIAGRAM

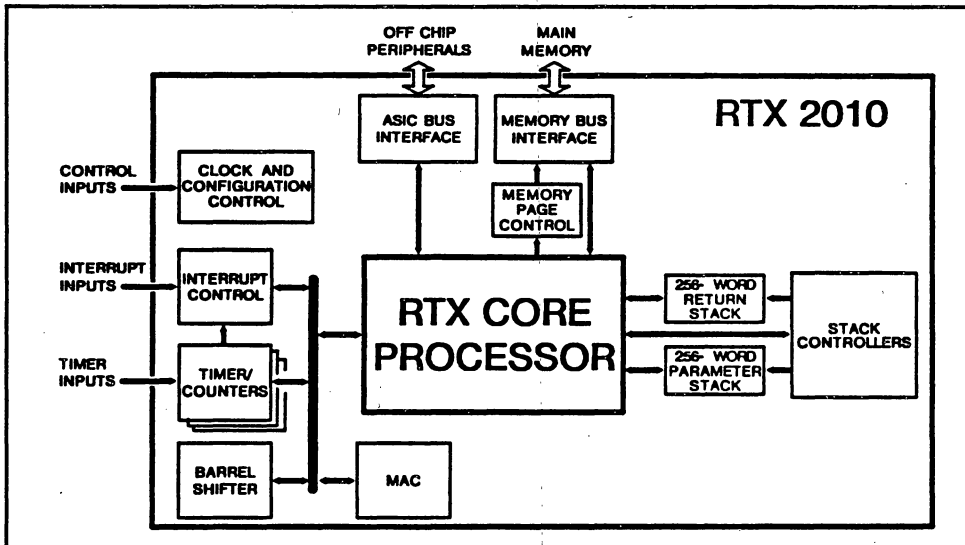


FIGURE 2.3: RTX 2010 BLOCK DIAGRAM

2.1 On-Chip Stacks

The RTX microcontroller contains two on-chip last-in-first-out (LIFO) stack memories. The top elements of each stack are immediately accessible through registers. The remainder of each stack is located in on-chip RAM arrays. The control logic associated with each stack determines which stack locations are to be read or written, and monitors the stacks for overflow and underflow conditions. See Section 3.1 for a description of stack operations.

Stacks on the RTX 2000 and RTX 2010 are each 256 elements deep; stacks on the RTX 2001A are 64 elements deep.

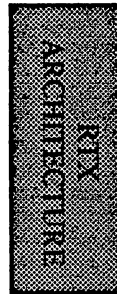
2.1.1 The Parameter Stack

The 16-bit wide Parameter Stack provides the operands for most math, logic, and memory reference instructions. It is used for passing parameters between subroutines, and as a scratchpad area for temporary storage of data.

The top two elements of the Parameter Stack are the TOP Register, which contains the top element, and the NEXT Register, which contains the second element. For certain instructions, TOP or NEXT are the implicit data source or destination, and the RTX can perform operations dealing with TOP and NEXT in one clock cycle. For more information about TOP and NEXT, see Chapter 4.

2.1.2 The Return Stack

The 21-bit wide Return Stack is used for storing subroutine return addresses and for holding index counts for loops and repeated instructions, and can also be used as a temporary storage area. The top element of the Return Stack is comprised of the 16-bit wide I register and the 5-bit wide IPR Register. The RTX can move data between the top elements of the Parameter and Return Stacks in a single clock cycle. For more information about I and IPR, see Chapter 4.



2.2 RTX 2000 Series Bus Architecture

The RTX 2000 Series bus architecture provides for unidirectional data paths and simultaneous operation of some data buses. This parallelism allows for maximum efficiency of data flow. External data is transferred via the ASIC Data Bus and the Memory Data Bus. Addresses for external access are output via the Memory Address Bus and the ASIC Address Bus.

2.2.1 Data Buses

The RTX QUAD Bus™ architecture consists of 4 independent 16-bit data buses, all of which may be active simultaneously.

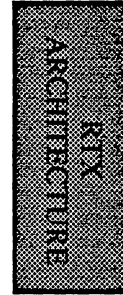
- The **Memory Data Bus** carries program instructions and program data to and from Main Memory. 16-bit data words (but not program instructions) are passed through byte-swapping hardware which allows the processor to control the order of storage in memory for the low and high bytes of the word.
- The **ASIC Bus™** is the I/O and register interface bus. This bus provides the interface between the Parameter Stack and the processor registers and external I/O devices. The ASIC Bus passes input data through the on-chip Arithmetic/Logic Unit (ALU) before pushing the data onto the Parameter Stack. This allows the RTX to perform math (adding, subtracting), logic (masking), and shifting operations on the data as it is being read.
- The **Parameter Stack Bus** carries data between the top-of-stack registers and the Parameter Stack RAM.
- The **Return Stack Bus** carries data between the top-of-stack registers and the Return Stack RAM.

2.2.2 Address Buses

For off-chip communications, the RTX microprocessor has two address buses: the 19-bit Memory Address Bus, and the 3-bit ASIC Address Bus.

- The **Memory Address Bus (MA19-MA01)** carries the address of the Main memory location to be accessed, either for instruction fetches or memory read/write operations. This is a 19-bit bus, along with Upper Data Strobe (UDS) and Lower Data Strobe (LDS), which allows the RTX to address 1 megabyte of memory.
- The **ASIC Address Bus (GA02-GA00)** carries address information for external ASIC devices.

See Chapter 5 for information about RTX External Bus Interfaces.



2.3 Registers

The RTX 2000 Series microcontrollers contain three types of registers. Stack related registers, Status/Control registers, and Internal Processor registers.

2.3.1 Stack Related Registers

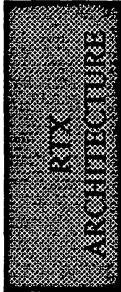
Stack related registers contain the top elements of the Parameter and Return Stacks. These registers are the implicit source and destination for many of the processor operations, and are described in detail in Chapter 4.

2.3.2 Status/Control Registers

Status/Control registers are accessed through the ASIC Bus, and determine the operating environment for the processor by controlling the processor configuration and on-chip peripheral devices. These registers are described in detail in Chapter 4.

2.3.3 Internal Processor Registers

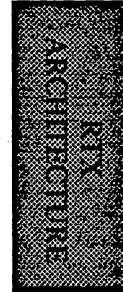
Internal Processor registers are not directly accessible to the programmer, and are described in Chapter 4.



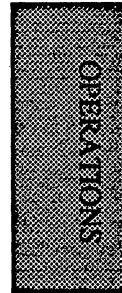
2.4 Memory

The RTX 2000, RTX 2001A, and RTX 2010 Microcontrollers directly address 1 Megabyte (512K 16-bit words) of memory. This memory is divided into 16 pages of 64K bytes (32K words) each, and may be made up of any combination of ROM, RAM, or memory mapped I/O devices.

The RTX memory interface is described in detail in Chapter 5.



CHAPTER 3
OPERATIONS



3 Operations

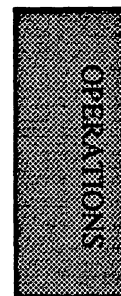
This chapter provides an overview of the internal processor operations. The operations are described in greater detail in Chapter 7, "Instruction Set".

3.1 Instruction Execution

The RTX Microcontrollers have an Instruction Decoder which provides control of all data paths and the Program Counter Register (PC). This hardware determines what function is to be performed by looking at the contents of the Instruction Register (IR), and subsequently determines the sequence of operations through data path control.

In one-cycle operations, the instruction which is to be executed is latched into IR at the beginning of a clock cycle, then is decoded. All necessary internal operations are performed simultaneously with fetching the next instruction. See Figure 3.1.

Instructions which perform memory access require two clock cycles to be executed. During the first cycle of a memory access instruction, the instruction is decoded, the address of the memory location to be accessed is placed on the Memory Address Bus (MA19-MA01), and the memory data (MD15-MD00) is read or written. During the second cycle, the address of the next instruction to be executed is placed on the Memory Address Bus, and the next instruction is fetched, as indicated in Figure 3.1.



OPERATIONS

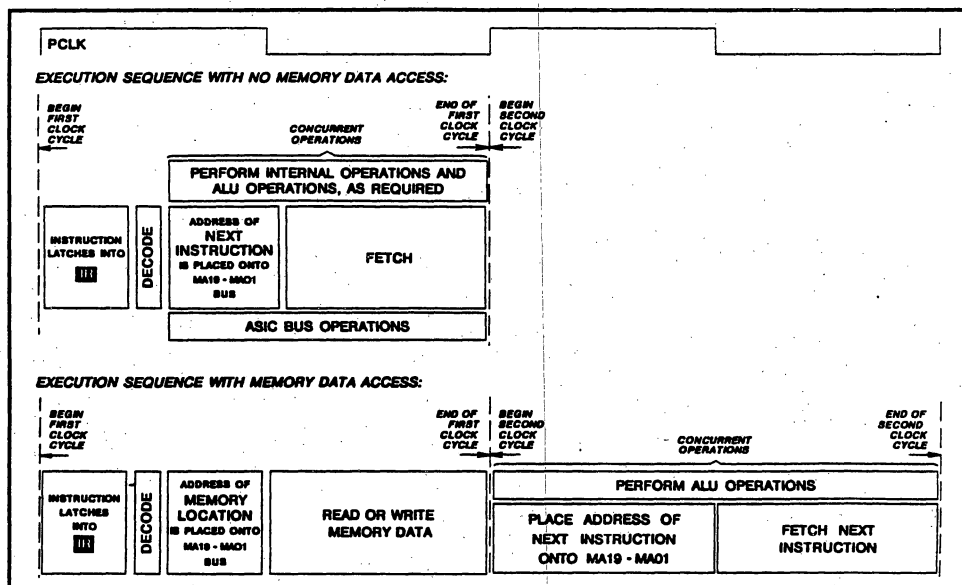


FIGURE 3.1: INSTRUCTION EXECUTION

3.2 Stack Operation

The RTX Microprocessors utilize a Last-in, First-out (LIFO) stack architecture. In this type of architecture, the last data element stored in the memory stack will be the first element retrieved from that region of memory. See Figure 3.2.

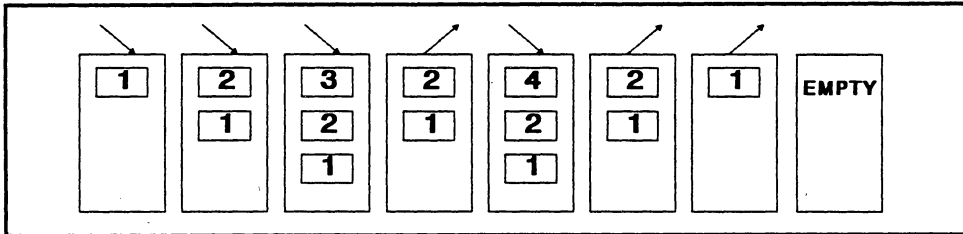


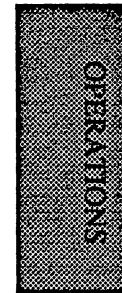
FIGURE 3.2: STACK OPERATION

This structure for information storage and retrieval provides the computer with one central location for temporary storage of information.

The RTX takes advantage of this architecture, utilizing two separate on-chip stacks. The first, the Parameter Stack, is used for temporary storage of data and for passing parameters between subroutines. The second, the Return Stack, is used to store return addresses during subroutine calls and returns. The Quad Bus™ architecture of the RTX Microcontrollers allows both stacks to be accessed in parallel by a single instruction, this dual stack arrangement allows overhead to be minimized during subroutine operations. The Return Stack can also be used for temporary storage of values when it is not being used during a subroutine call or return.

For faster access, both the Parameter Stack and the Return Stack utilize registers for the top elements and on-chip memory (Stack Memory) for the remaining elements.

For more detailed information about RTX stack operation, see Section 6.1.



3.3 Subroutine Calls and Returns

An RTX subroutine call instruction has the address of the routine to be called embedded in the instruction. When the subroutine call is executed, the address of the instruction following the call instruction is pushed onto the Return Stack. When the subroutine is completed, a Return-from-Subroutine instruction will pop the return address from the stack, and execution will resume with the instruction following the call.

The RTX architecture is optimized for performing subroutine calls and returns with minimum processor overhead. A subroutine call within the same memory page can be made in one clock cycle. A call to a location in a different memory page takes 3 clock cycles.

Subroutine returns take 0 clock cycles if performed as part of another instruction, and 1 cycle if executed as a separate instruction.

3.4 Branching And Looping

The RTX can perform unconditional branches or conditional branches, based on the contents of the top elements of the Parameter and Return Stacks. All branches take one clock cycle, regardless of whether or not the branch is performed.

3.5 Streamed Instructions

The RTX processor has a "streamed" instruction feature, in which an instruction is repeated a specified number of times without repeating the instruction fetch cycle. This feature is useful for doing fast data transfers, loops and some math functions.

See Chapter 7, "Instruction Set" for more details about the "streamed" instruction feature.

3.6 Math/Logic Operations

Math and logic operations are performed by the ALU circuitry of the RTX. The operations which may be performed include the simple math operators + and -, and the logic operators AND, OR, XOR, NOR, NAND, XNOR, and NOT.

See Section 6.3 for information about the on-chip hardware multiplier, multiplier/accumulator, barrel shifter, and Floating Point support features.

The TOP register is always one input to the ALU. The second, "Y", input may come from a variety of sources, as indicated in Figure 3.3.

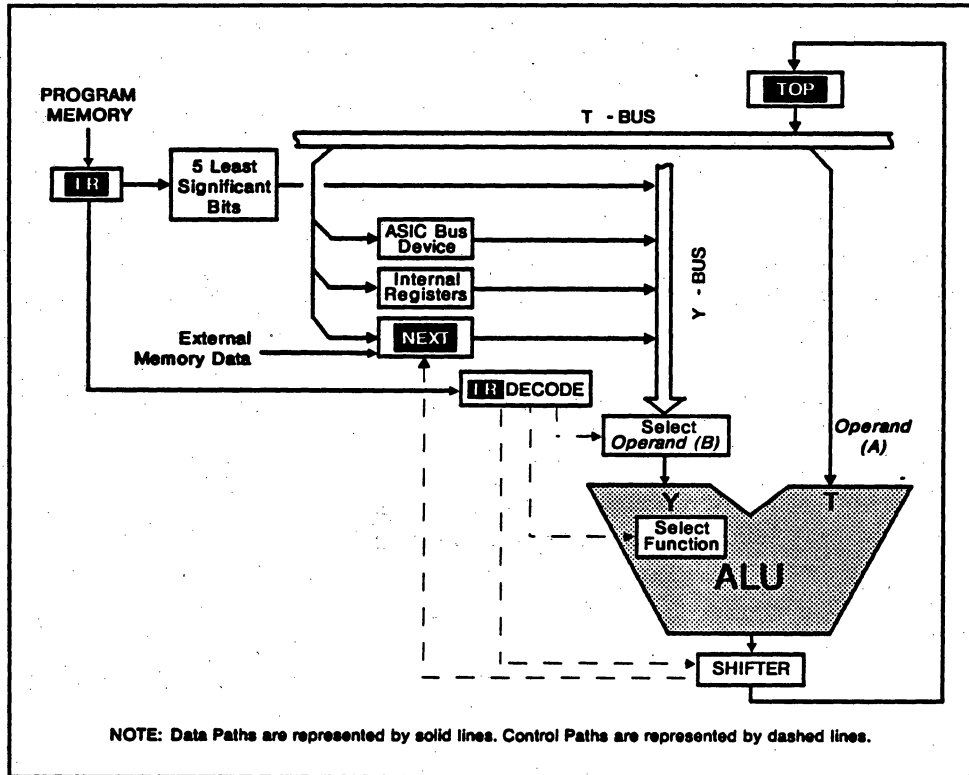


FIGURE 3.3: ALU DATA FLOW

3.6.1 Registers And I/O Devices

The contents of the TOP and NEXT registers are always available as operands to the ALU, and are the implicit operands for most of the RTX Math/Logic instructions.

The contents of the other registers and external I/O devices are addressable as devices on the ASIC Bus.

3.6.2 Memory

Data may be fetched from, and stored to, Main Memory using the Word and Byte access instructions (Classes 14 and 15 in Chapter 7, the "Instruction Set") and User memory access instructions (Class 12).

3.6.3 Literals

A literal is a constant value to be pushed onto the stack, or to be used as the second operand of an arithmetic or logic operation. The RTX processor recognizes two types of literals - short literals and long literals.

A short literal is a 5-bit value between 0 and 31 and is encoded as a field in a machine instruction.

A long literal may be any signed or unsigned 16-bit integer, and is stored in main memory immediately following the opcode that utilizes it.

3.7 Stack Operations

The top two locations of the Parameter Stack are TOP and NEXT, and the remainder of the stack memory is located in on-chip RAM. Because of this, the RTX Microcontrollers have the ability to manipulate stack elements to allow optimization of many instructions. Descriptions of these stack manipulation operations are given in the following sections. These primitives can be combined with other operations to allow one-cycle execution of multiple operations. See Chapter 7 for information about specific instructions.

3.7.1 DUP

DUP copies the top element of the Parameter Stack, and pushes the result onto the stack, leaving the stack with two identical elements in the top two stack locations.

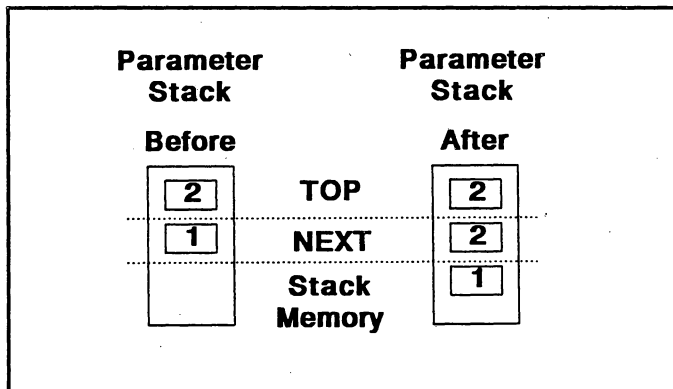
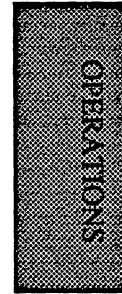


FIGURE 3.4: STACK EFFECTS OF DUP



3.7.2 SWAP

SWAP flips the top two elements of the Parameter Stack, causing the top element to move to the second location, and the second element to move to the top location.

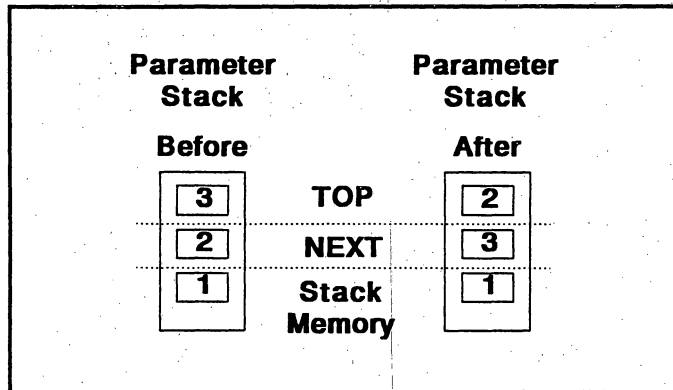


FIGURE 3.5: STACK EFFECTS OF SWAP

3.7.3 DROP

DROP pops the Parameter Stack, dropping the top element. That element is lost, and is not used in subsequent operations.

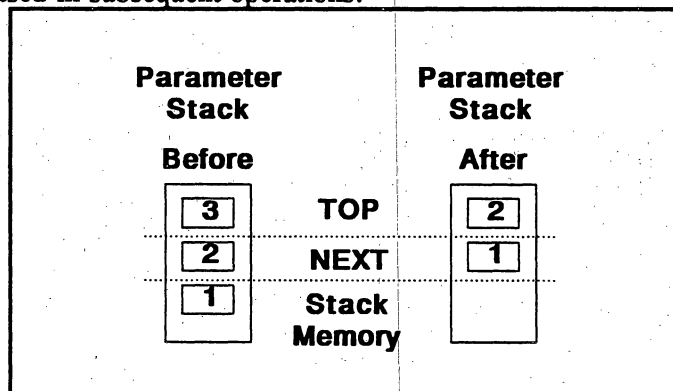


FIGURE 3.6: STACK EFFECTS OF DROP

3.7.4 OVER

OVER copies and pushes the third Parameter Stack element into the top location.

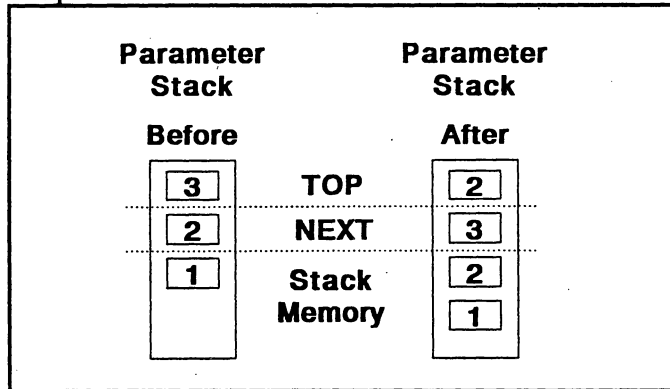


FIGURE 3.7: STACK EFFECTS OF OVER

3.7.5 >R

>R (called "to R") takes the information in TOP and stores it in the least significant 16 bits (I) of the top location of the Return Stack. This causes the current Code page value to be written to IPR, the most significant 5 bits of the top location of the Return Stack.

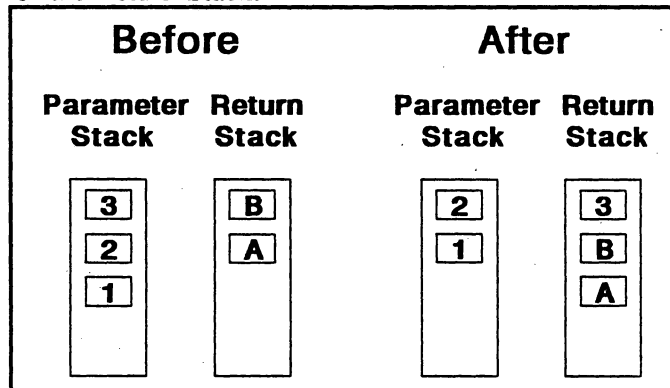


FIGURE 3.8: STACK EFFECTS OF >R

But, isn't CPR
only 4 bits?
What about the
5th IPR bit?

3.7.6 R>

r> (called "R from") retrieves the information in the least significant 16 bits of the top element of the Return Stack and pushes it into TOP.

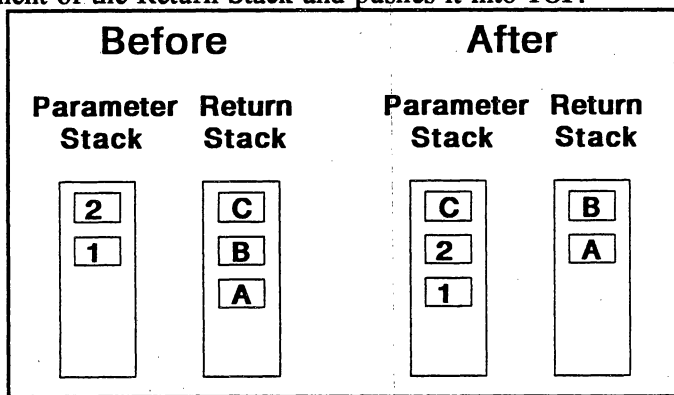


FIGURE 3.9: STACK EFFECTS OF R>

3.7.7 R@

r@ (called "R fetch") copies the top of the Return Stack to the top of the Parameter Stack.

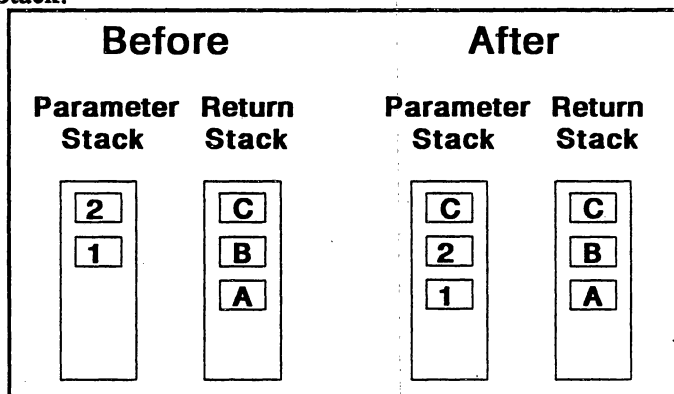


FIGURE 3.10: STACK EFFECTS OF R@

3.8 Interrupts

The RTX processor may be interrupted from several sources, both from internal devices and from external inputs.

The on-chip Interrupt Controller has fourteen interrupt request inputs. Thirteen of these interrupt request inputs are maskable interrupts, and one is a Non-Maskable Interrupt (NMI) request.

3.8.1 Maskable Interrupts

The Interrupt Controller samples the request inputs during each instruction, prioritizes any active interrupt requests, and signals the processor when an interrupt request is present.

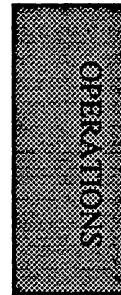
For more information about interrupt acknowledgement, disabling interrupts, and software interrupts, see Section 6.2.

3.8.2 Non-Maskable Interrupts (NMI)

The NMI is an external, edge-sensitive input which requires a rising edge to request an interrupt.

3.8.2.1 On the RTX 2000

The NMI can cause the processor to perform an Interrupt Acknowledge cycle in the middle of such operations as Step Math instructions, Streamed instructions, and other operations that could result in the loss of data or misoperation of the hardware if interrupted. For this reason, a "Return From Subroutine" should not be performed from the NMI service routine. Instead, the NMI handler should re-initialize the system.



3.8.2.2 On the RTX 2001A

On the RTX 2001A, the NMI input has a glitch filter circuit which requires that the signal that initiates the NMI must last at least two cycles of ICLK.

The NMI can cause the processor to perform an Interrupt Acknowledge cycle in the middle of such operations as Step Math instructions, Streamed instructions, and other operations that could result in the loss of data or misoperation of the hardware if interrupted. For this reason, a "Return From Subroutine" should not be performed from the NMI service routine. Instead, the NMI handler should re-initialize the system.

3.8.2.3 On the RTX 2010

On the RTX 2010, the NMI has two modes of operation which are controlled by the NMI_MODE Flag (bit 11 of the CR).

When CR bit 11 is cleared (=0), the NMI cannot be masked and can interrupt any cycle. This allows a fast response to the NMI, but does not guarantee that a Return From Interrupt will always provide correct operation. The NMI_MODE Flag is cleared at Reset.

When the NMI_MODE bit is set (=1), the NMI may be inhibited by the processor during certain critical operations, and further NMIs and maskable interrupts are disabled until the NMI Interrupt Service Routine has been completed and a return has been executed. In this mode, a return from the NMI Interrupt Service Routine will allow the processor to resume correct execution at the point where it was interrupted.

OPERATIONS

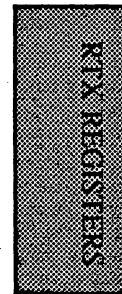
CHAPTER 4
RTX REGISTERS

4 RTX Registers

The three types of registers which the RTX microcontrollers use are: Stack Related Registers, Internal Processor Registers, and Status/Control Registers.

At power up or Reset, the RTX registers are initialized. The reset states for the RTX 2000 are shown in Table 4.1. The reset states for the RTX 2001A are shown in Table 4.2. The reset states for the RTX 2010 are shown in Table 4.3. In each of these tables, the read and write capabilities of each register are indicated in the R/W column, where:

- R-W Indicates that the register can be either read from or written to.
- R Indicates a read-only register.
- W Indicates a write-only register.
- R/W Indicates that the first register is read-only and the second register is write-only (as in the case of the Timer/Counter and Timer Preload Registers).
- * Indicates that individual bits in the register may be read-only or write-only and that the bit map for that register should be consulted.
- N Indicates that the register cannot be read from or written to.



Register addresses are given in hexadecimal, denoted by "H" here and elsewhere in this manual.

The sections which follow describe each of the registers in more detail.

TABLE 4.1: RTX 2000 REGISTER INITIALIZATION

REGISTER	ASIC ADDR	INITIALIZATION VALUES	R/W	COMMENTS
TOP		0000 0000 0000 0000	R-W	
NEXT		1111 1111 1111 1111	R-W	
IR		0000 0000 0000 0000	R	
I	00H 01H 02H	1111 1111 1111 1111	R-W	
CR	03H	0100 0000 0000 1000	*	Interrupts disabled, BOOT=1, Byte Order=0
MD	04H	1111 1111 1111 1111	R-W	
SR	06H	0000 0000 0000 0000	R-W	
PC	07H	0000 0000 0000 0000	R-W	
IMR	08H	0000 0000 0000 0000	R-W	All interrupts unmasked
SFR	09H	0000 0000 0000 0000	R-W	First stack location
SLR	08H	1111 1111 1111 1111	W	Limit for each stack set to 255
IVR	08H	0000 0010 0000 0000	R	Read only; initialized to "No Interrupt Value"
IPR	0CH	0000 0000 0000 0000	R-W	Initialize for Code Page 0
DPR	0DH	0000 0000 0000 0000	R-W	Initialize for Data Page 0
UPR	0EH	0000 0000 0000 0000	R-W	Initialize for User Page 0
CPR	0FH	0000 0000 0000 0000	R-W	Initialize for Code Page 0
IBC	10H	0000 0000 0000 0000	*	Interrupt Base=0, Counters on internal clocks, no rounding, use CPR for data accesses
UBR	11H	0000 0000 0000 0000	R-W	User Base Address = 0
TC0/TP0 TC1/TP1 TC2/TP2	13H 14H 15H	0000 0000 0000 0000	R/W R/W R/W	All Timer/Counters set to time-out after 65536 counts
MLR	16H	1111 1111 0000 0000	R	Read only; Mult. Low Product
MHR	17H	1111 1111 1111 1111	R	Read only; Mult. High Product

RTX REGISTERS

TABLE 4.2: RTX 2001A REGISTER INITIALIZATION

REGISTER	ASIC ADDR	INITIALIZATION VALUES	R/W	COMMENTS
TOP		0000 0000 0000 0000	R-W	
NEXT		1111 1111 1111 1111	R-W	
IR		0000 0000 0000 0000	R	
I	00H 01H 02H	1111 1111 1111 1111	R-W	
CR	03H	0100 0000 0000 1000	*	Interrupts disabled, BOOT=1, Byte Order=0
MD	04H	1111 1111 1111 1111	R-W	
SR	06H	0000 0000 0000 0000	R-W	
PC	07H	0000 0000 0000 0000	R-W	
IMR	08H	0000 0000 0000 0000	R-W	All interrupts unmasked
SPR	09H	0000 0000 0000 0000	R-W	Stack start addresses set to 0
SUR	0AH	0000 0011 0000 0011	R-W	Stack underflow limits set
SVR	0BH	1111 1111 1111 1111	W	Write only; each stack overflow limit set for max. stack size
IVR	0BH	0000 0010 0000 0000	R	Read only; Interrupt Vector initialized to "No Interrupt" value
IPR	0CH	0000 0000 0000 0000	R-W	Initialized for Code Page 0
DPR	0DH	0000 0000 0000 0000	R-W	Initialized for Data Page 0
UPR	0EH	0000 0000 0000 0000	R-W	Initialized for User Page 0
CPR	0FH	0000 0000 0000 0000	R-W	Initialized for Code Page 0
IBC	10H	0000 0000 0000 0000	*	Interrupt Base=0, Counters on internal clocks, no rounding, use CPR for data accesses
UBR	11H	0000 0000 0000 0000	R-W	User Base address set to 0
TC0/TP0 TC1/TP1 TC2/TP2	13H 14H 15H	0000 0000 0000 0000	R/W R/W R/W	All Timer/Counters set to time-out after 65536 counts
RX	16H	0000 0000 0000 0000	R-W	Scratchpad/Counting Register
RH	17H	0000 0000 0000 0000	R-W	Scratchpad Register

RTX REGISTERS

TABLE 4.3: RTX 2010 REGISTER INITIALIZATION

REGISTER	ASIC ADDR	INITIALIZATION VALUES	R/W	COMMENTS
TOP		0000 0000 0000 0000	R-W	
NEXT		1111 1111 1111 1111	R-W	
IR		0000 0000 0000 0000	N	
I	00H 01H 02H	1111 1111 1111 1111	R-W	
CR	03H	0100 0000 0000 1000	*	Interrupts disabled, BOOT=1, Byte Order=0
MD	04H	1111 1111 1111 1111	R-W	
SR	06H	0000 0010 0000 0000	R-W	
PC	07H	0000 0000 0000 0000	R-W	
IMR	08H	0000 0000 0000 0000	R-W	All interrupts unmasked
SPR	09H	0000 0000 0000 0000	R-W	Stack start addresses set to 0
SUR	0AH	0000 0111 0000 0111	R-W	Stack underflow limits set
SVR	0BH	1111 1111 1111 1111	W	Write only; each stack overflow limit set for max. stack size
IVR	0BH	0000 0010 0000 0000	R	Read only; Interrupt Vector initialized to "No Interrupt" value
IPR	0CH	0000 0000 0000 0000	R-W	Initialized for Code Page 0
DPR	0DH	0000 0000 0000 0000	R-W	Initialized for Data Page 0
UPR	0EH	0000 0000 0000 0000	R-W	Initialized for User Page 0
CPR	0FH	0000 0000 0000 0000	R-W	Initialized for Code Page 0
IBC	10H	0000 0000 0000 0000	*	Interrupt Base=0, Counters on internal clocks, no rounding, use CPR for data accesses
UBR	11H	0000 0000 0000 0000	R-W	User Base address set to 0
MXR	12H	0000 0000 0000 0000	R-W	MAC Extension Register; LZD 0 Count; Barrel Shifter Count
TC0/TP0 TC1/TP1 TC2/TP2	13H 14H 15H	0000 0000 0000 0000	R/W R/W R/W	All Timer/Counters set to time-out after 65536 counts
MLR	16H	0000 0000 0000 0000	R-W	Multiplier and MAC Low Register
MHR	17H	0000 0000 0000 0000	R-W	Multiplier, Barrel Shifter, and LZD High Register; MAC Middle Register

RTX REGISTERS

4.1 Stack Related Registers

These registers contain the top elements of the Parameter and Return Stacks, and are the implicit source and destination for many of the processor operations.

4.1.1 TOP Register (Parameter Stack)

The TOP Register contains the top element of the Parameter Stack, and has no ASIC address assignment.

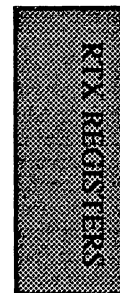
This is the primary working register for the processor, and is the implicit data source or destination for certain instructions.

All ALU results are loaded into TOP. The output from TOP may be written to any ASIC Bus register and to external I/O devices.

4.1.2 NEXT Register (Parameter Stack)

The NEXT Register contains the second element of the Parameter Stack, and has no ASIC address assignment.

During arithmetic operations, this register holds the lower 16 bits of a 32-bit operand. NEXT is also the source of data for all memory writes.



4.1.3 I Register

(Return Stack)

The Index Register, I, can be accessed at three different ASIC addresses, and the choice of ASIC address determines the type of operation to be performed.

As a Stack Related Register at ASIC addresses 00H (Hex) and 01H, I contains the lower 16 bits of the top element of the 21-bit wide Return Stack. IPR contains the other 5 bits. See Section 4.1.4 for more details about IPR.

The contents of I may be accessed in either push/pop mode, in which values are moved to/from Return Stack memory as required, or in read/write mode in which the Return Stack is not affected.

In addition to its use in holding return address bits, at ASIC address 02H, this register is also used to hold the count for streamed (repeated) instructions and loop instructions. Operation of I at this ASIC address is described in more detail in Section 4.1.3.3. I access operations and the associated addresses are shown in Table 4.4.

TABLE 4.4: I ACCESS OPERATIONS

OPERATION (g-read, g-write)	RETURN BIT VALUE	ASIC ADDRESS ggggg	REGISTER	FUNCTION
R	0	00000	I	Pushes the contents of I into IOP ₂ (with no pop of the Return Stack)
R	1	00000	I	Pushes the contents of I into IOP ₂ , then performs a Subroutine Return
W	0	00000	I	Pops the contents of IOP ₂ into I (with no push of the Return Stack)
W	1	00000	I	Performs a Subroutine Return, then pushes the contents of IOP ₂ into I
R	0	00001	I	Pushes the contents of I into IOP ₂ , popping the Return Stack
R	1	00001	I	Pushes the contents of I into IOP ₂ without popping the Return Stack, then executes the Subroutine Return
W	0	00001	I	Pushes the contents of IOP ₂ into I popping the Parameter Stack
W	1	00001	I	Performs a Subroutine Return, then pushes the contents of IOP ₂ into I
R	0	00010	I	Pushes the contents of I shifted left by one bit, into IOP ₂ (the Return Stack is not popped)
R	1	00010	I	Pushes the contents of I shifted left by one bit, into IOP ₂ (the Return Stack is not popped), then performs a Subroutine Return
W	0	00010	I	Pushes the contents of IOP ₂ into I as a "stream" count, indicating that the next instruction is to be performed a specified number of times; the Parameter Stack is popped
W	1	00010	I	Performs a Subroutine Return, then pushes the stream count into I

RTX REGISTERS

Too small; difficult to read

does this pop TOP?

(also for 0011)

4.1.3.1 I

At Address 00H

Location 00H is used to access I without causing any net pushes or pops of the Return Stack.

Reading from this location pushes the contents of I onto the Parameter Stack.

Reading from this location as part of a subroutine return pushes the contents of I onto the Parameter Stack, then performs a Return-From-Subroutine.

Writing to this location during normal operation pops the top item on the Parameter Stack into I; the original contents of I are lost.

Writing to this location as part of a subroutine return operation first executes the return, then pushes the top item of the Parameter Stack onto the Return Stack.

4.1.3.2 I

At Address 01H

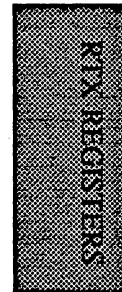
I at address 01H is used to push and pop the Return Stack.

Reading this location during normal operation pushes the contents of I onto the Parameter Stack and pops the Return Stack.

Reading this location as part of a subroutine return operation pushes the contents of I onto the Parameter Stack without popping the Return Stack, then executes the subroutine return. See Section 5.1.1 for more information about operation during subroutine returns.

Writing to this location during normal operation pushes the top item from the Parameter Stack onto the Return Stack, popping the Parameter Stack.

Writing to this location as part of a subroutine return operation first executes the subroutine return, then pushes the top Parameter Stack item onto the Return Stack. See Section 5.1.1 for more information about subroutine return operation.



huh? section 5.1.1? did you mean 5.2.1?

4.1.3.3 I

At Address 02H (Stream Count/Loop Count)

Reading this location pushes the contents of I shifted left by one bit onto the Parameter Stack. The Return Stack is not popped.

Reading this location as part of a subroutine return pushes the contents of I shifted left by one bit into TOP (the Return Stack is not popped), and then performs a Return-From-Subroutine.

Writing to this location during normal operation pushes the top Parameter Stack item into I as a "stream" count, indicating that the next instruction is to be performed a specified number of times, the Parameter Stack is popped.

Writing to this location as part of a subroutine return operation executes the subroutine return first, then pushes the stream count onto the Return Stack.

RTX REGISTERS

See Section 7.1.1 for more information on streamed instructions.

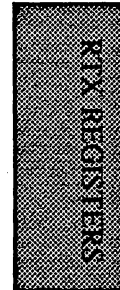
4.1.4 IPR Register

(Return Stack)

The IPR Register, at ASIC Address 0CH (Hex), can be described as both a Stack Related Register and as a Control/Status Register. See Section 4.3.10 for more information. This register contains the 5 most significant bits of the top element of the Return Stack (the I Register contains the other 16 bits).

Reading from or writing directly to IPR does not push or pop the Return Stack, but pushes or pops of the Return Stack (when reading or writing to I) do cause the contents of IPR to be overwritten. Writing to I during non-subroutine operations causes the current Code Page value to be written to IPR.

↳ what about bit 4?
Is this loaded with DPRSEL?



4.2 Internal Processor Registers

Internal Processor Registers are not directly accessible to the programmer.

The Instruction Register, **IR**, is actually a latch which contains the instruction currently being executed. This register is loaded directly from main memory via an instruction fetch, and is not accessible under program control.

The bits of the instruction in **IR** are decoded to determine which operations to perform, to determine the location of the next instruction to be executed, and to provide data for immediate operations.

RTX REGISTERS

~~In a future individual release
it would be nice to list the
"hidden state" of the processor
that can be corrupted
(e.g. stream read bit can be
corrupted by interrupt)~~ stet

4.3 Control/Status Registers

The contents of the RTX microcontroller's Control/Status Registers determine the operating environment for the processor, and allow the processor to monitor and control the various I/O devices on the chip.

All internal registers are accessed through the ASIC Bus. ASIC addresses 0 through 23 (17 hexadecimal) are assigned to on-chip registers and devices, and are described in this section. Section 7.7 describes the RTX instructions which access the ASIC Bus.

addresses 0, 1, 2
are in previous
sections
Yes, this is a nit.

RTX REGISTERS

4.3.1 The Configuration Register - Address 03H

The Configuration Register, **CR**, controls the setup/status of the RTX processor.

Reading this location pushes the current contents of the register onto the Parameter Stack.

Writing to this location pops the top Parameter Stack item into **CR**, updating the control bits. The Interrupt Base/Control Register contains additional processor control bits.

The bits in **CR** are assigned as shown in Table 4.5.

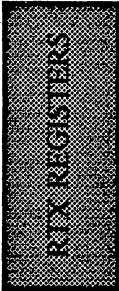


TABLE 4.5: CONFIGURATION REGISTER BIT ASSIGNMENTS

MSB	IL	RID	†RES ARCE	RES	†RES NMIM	RES	RES	RES	RES	RES	RES	SID	BOOT	BYTE	CCY	CY	LSB
-----	----	-----	--------------	-----	--------------	-----	-----	-----	-----	-----	-----	-----	------	------	-----	----	-----

IL	Bit 15 (MSB)	Read-only; Interrupt Latch: When set to 1, indicates that an interrupt request is pending. See Section 6.2.
RID	Bit 14	Read-only; Read Interrupt Disable: Status of Interrupt Disable bit. When set to 1, indicates that interrupts are disabled. Resets to 1. Use SID bit to set value. See Section 6.2.
†RES ARCE	Bit 13	Reserved on the RTX 2000 and RTX 2001A. On the RTX 2010: When this bit is set, the PCLK cycle for every ASIC bus read is extended. See Section 5.1 for more details.
RES	Bit 12	Reserved
†RES NMIM	Bit 11	Reserved on the RTX 2000 and RTX 2001A. On the RTX 2010: When this bit =1, return from a Non-Maskable Interrupt can be made. See Section 3.5.2.3 for more information.
RES	Bits 5-10	Reserved
SID	Bit 4	Write-only, <u>always reads as zero</u> ; Set Interrupt Disable: When set to 1, the processor will not respond to interrupts. RID bit contains true value of Interrupt Disable bit. See Section 6.2.
BOOT	Bit 3	R/W; BOOT: Controls BOOT output pin. May be used to select boot memory on power up.
BYTE	Bit 2	R/W; Byte Order: Controls order in which bytes of data will be read from or written to memory. See Section 5.2.2.1
CCY	Bit 1	R/W; Complex Carry: Carry bit from ALU extension. See Section 8.3.
CY	Bit 0	R/W; Carry: ALU Carry output. See Sect. 8.3.

8000

6000

2000

1000

800

10

8

4

2

1

3.8.2.3

RTX REGISTERS

read value result is implementation-dependent

2020 may allow read of RID on this bit

4.3.2 The MD Register - Address 04H

The MD Register is used to hold intermediate values during step math operations (see Chapter 8). It may also be used as a general purpose scratchpad register.

Reading this location pushes the contents of the MD Register onto the Parameter Stack.

Writing to this location pops the top Parameter Stack item into the MD Register, replacing its previous contents.

4.3.2.1 MD On The RTX 2000 and RTX 2010

On the RTX 2000, MD is the Multistep Divide Register. During multistep divide operations, this register holds the divisor, while TOP and NEXT hold the 32-bit dividend.

4.3.2.2 MD On The RTX 2001A

On the RTX 2001A, MD is the Multiply/Divide Register. This register holds the divisor during step divide operations (the 32-bit dividend is in TOP and NEXT). During step multiply operations, this register holds the multiplier, while NEXT holds the multiplicand.

RTX REGISTERS

This implies step multiply doesn't work on 2000, 2010. This is untrue

4.3.3 The SQ Register - Address 05H

This address is a "pseudo-register" for step math operations (see Chapter 8).

Reading this location reads the contents of the MD Register, shifts the result left by one bit, then logically OR's this value with the contents of the SR Register. The result is pushed onto the Parameter Stack.

Writing to this location shifts the top Parameter Stack item left by 8 bits, then pops this value into the MD Register.

4.3.4 The SR Register - Address 06H

The Square Root Register is used to hold intermediate values during the calculation of square roots. It may also be used as a general purpose scratchpad register.

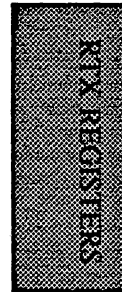
Reading this location pushes the contents of the SR Register onto the Parameter Stack.

Writing to this location pops the top Parameter Stack item into the SR Register, replacing its previous contents.

Not documented
on data
sheets!
Table 9.

? zero
shifted
into
low
bit?

? zeros shifted into
8 bits ? low



4.3.5 The PC Register - Address 07H

The Program Counter Register, PC, contains the lower 16 bits of the address of the instruction following the one currently executing.

Reading this location pushes the contents of the PC (the address of the instruction following the one which reads the PC) onto the Parameter Stack. OK

Writing to this location during normal operation causes a subroutine call to the address contained in the top Parameter Stack item; the Parameter Stack is popped. Writing to this location as part of a subroutine return operation pushes the top Parameter Stack item onto the Return Stack, then executes the subroutine return; the Parameter Stack is popped.

See Table 4.6 for PC Register access operations.

TABLE 4.6: PC REGISTER ACCESS OPERATIONS

OPERATION (g-read, g-write)	RETURN BIT VALUE	ASIC ADDRESS 88888	REGISTER	FUNCTION
R	0	00111	PC	Pushes the contents of PC into IOP
R	1	00111	PC	Pushes the contents of PC into IOP, then performs a Subroutine Return
W	0	00111	PC	Performs a Subroutine Call to the address contained in IOP, popping the Parameter Stack
W	1	00111	PC	Pushes the contents of IOP onto the Return Stack before executing the Subroutine Return

OK
JUMP instead of CALL?

RTX REGISTERS

4.3.6 The Interrupt Mask Register, IMR - Address 08H

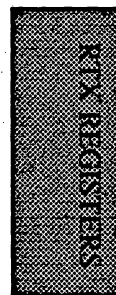
The bits in the Interrupt Mask Register, IMR, cause individual interrupt request inputs to the Interrupt Controller to be enabled or disabled. When a bit is set to 1, the corresponding input is masked (disabled). The IMR resets to all 0's - all interrupts unmasked. Only NMI, the Non-Maskable Interrupt cannot be masked.

Reading this location pushes the current contents of the IMR onto the Parameter Stack.

Writing to this location pops the top Parameter Stack item into the IMR, updating the mask values. See Table 4.7 for bit assignments.

TABLE 4.7: INTERRUPT MASK REGISTER BIT ASSIGNMENTS

MSB	RES	RES	SWI	E15	E14	E13	T2	T1	T0	E12	RSV	PSV	RSU	PSU	E11	RES	LSB
	RES			Bits 14-15													
			SWI	Bit 13													
			E15	Bit 12													
			E14	Bit 11													
			E13	Bit 10													
			T2	Bit 9													
			T1	Bit 8													
			T0	Bit 7													
			E12	Bit 6													
			RSV	Bit 5													
			PSV	Bit 4													
			RSU	Bit 3													
			PSU	Bit 2													
			E11	Bit 1													
			RES	Bit 0													



4.3.7 The Stack Pointer Register, SPR - Address 09H

This location contains the combined registers for the Parameter Stack Pointer and Return Stack Pointer, which are accessed together. Bits 0-7 contain the pointer for the Parameter Stack, bits 8-15 contain the pointer for the Return Stack.

Reading this location pushes the contents of the register onto the Parameter Stack. The value read for the Parameter Stack pointer will reflect the Parameter Stack contents after the register value is pushed.

Writing to this location pops the top Parameter Stack item into the Stack Pointer Register.

4.3.8 Address 0AH

The assignment and utilization of this address is different for the RTX 2000, RTX 2001A, and RTX 2010 Microcontrollers.

4.3.8.1 On The RTX 2000

This location is reserved on the RTX 2000.

4.3.8.2 On The RTX 2001A and RTX 2010

On the RTX 2001A and RTX 2010, this address is used for the Stack Underflow Limit Register, SUR. This register holds the underflow limit values for the Parameter Stack and the Return Stack, which must be accessed together.

This register can be utilized to define the use of substacks for both stacks. See Section 6.1.3 for more stack/substack configuration information.

4.3.9 Address 0BH: IVR, SVR, And SLR

This address serves as two registers, and may be utilized by either the Interrupt Controller or the Stack Controllers, depending on whether a read operation or a write operation is being performed.

In the read-only mode, this is the **Interrupt Vector Register** on all RTX 2000 Family Microcontrollers, and is used to hold the current Interrupt Vector value. This register is initialized to the "No Interrupt" value. Reading this location pushes the value of the current vector being generated by the Interrupt Controller onto the Parameter Stack and clears any pending Timer/Counter interrupts.

In the write-only mode, this address is utilized for stack limit operations by the Stack Controller. The specific function of this address differs depending on which processor is being used.

4.3.9.1 Write-only On The RTX 2000: SLR

In the write-only mode, this address is used as the Stack Limit Register. At Reset, this register is set to its maximum value of 255.

Writing to this location loads new values into the Parameter Stack and Return Stack Limit Registers. Bit 0-7 are assigned to the Parameter Stack, bits 8-15 to the Return Stack, and both are accessed together.

4.3.9.2 Write-only On The RTX 2001A: SVR

In the write-only mode, this address is used for the Stack Overflow Limit Register and holds the overflow limits for the Parameter Stack and the Return Stack. These limits must be accessed together. The maximum overflow limit value for each stack on the RTX 2001A is 64.

4.3.9.3 Write-only On The RTX 2010: SVR

In the write-only mode, this address is used for the Stack Overflow Limit Register and holds the overflow limits for the Parameter Stack and the Return Stack. These limits must be accessed together. The maximum overflow limit value for each stack on the RTX 2010 is 256.

4.3.10 Index Page Register - Address 0CH

This 5-bit register contains bits 16-20 of the top item of the Return Stack. Bits 0-3 of the Index Page Register, (IPR), contain the contents of the Code Page Register at the time the current subroutine was called (i.e., the memory page number to which the processor will return when execution of the current subroutine has been completed. Bit 4 contains the value of the Data Page Register Select Bit (DPRSEL) at the time the current subroutine was called. See Figure 4.1 and Section 5.2.2.

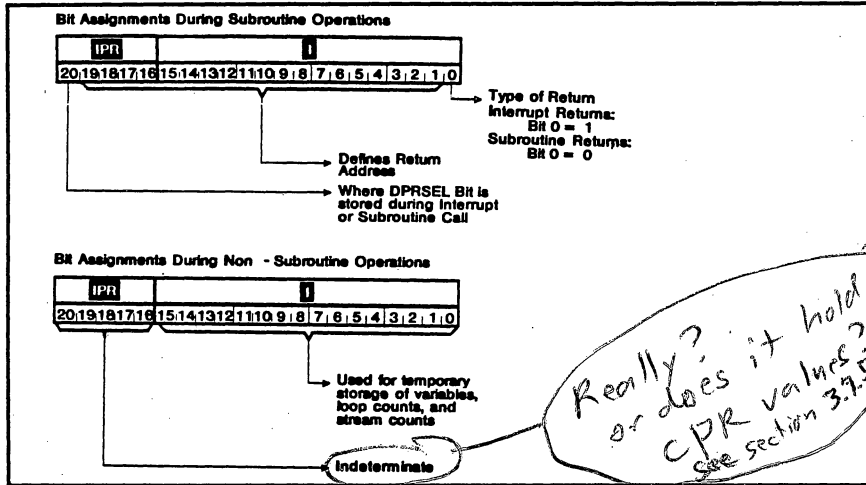


FIGURE 4.1: RETURN STACK BIT ASSIGNMENTS

The Index Page Register provides a mechanism to access the upper bits of the subroutine return address. Reads and writes to the IPR do not pop or push the Return Stack. However, operations which push and pop the Return Stack do overwrite the contents of IPR. These operations include subroutine calls, subroutine returns, and reads and writes to the Index Register at ASIC Bus addresses 01H and 02H. → what about 00H?

Reading this location pushes the contents of the IPR onto the Parameter Stack.

Writing to this location loads a new 5-bit value into the IPR. This operation should be used with caution, because it will change the subroutine return address.

4.3.11 Data Page Register, DPR - Address 0DH

When the DPRSEL bit (bit 5 of the IBC Register) is set =1, this 4-bit register ~~contains~~ the number of the memory page which will be accessed by memory reference instructions. See Sections 4.3.14 and 5.2.1.

provides

4.3.12 User Page Register, UPR - Address 0EH

This 4-bit register contains the number of the memory page which will be accessed by User Memory Space instructions. See User Memory Access Instructions in Chapter 7.

RTX REGISTERS

4.3.13 Code Page Register, CPR - Address 0FH

This register contains the number of the memory page which will be accessed by all instruction fetch cycles. Additionally, if the DPRSEL bit is set =0, this register contains the number for the memory page to be accessed by memory reference instructions.

provides

4.3.14 Interrupt Base/Control Register - Address 10H

The bits in this register control special processor setup and configuration values. See Table 4.8 for the IBC Register bit assignments. See Section 4.3.1 for information about additional control/status bits in CR.

TABLE 4.8: IBC REGISTER BIT ASSIGNMENTS

MSB	IB5	IB4	IB3	IB2	IB1	IB0	TB1	TB0	CYCEXT	ROUND	DPRSEL	RES	*	*	*	*	LSB
-----	-----	-----	-----	-----	-----	-----	-----	-----	--------	-------	--------	-----	---	---	---	---	-----

IB0-IB5	Bits 10-15	Interrupt Vector Base Address: Provides bits 10-15 of Interrupt vector generated by the Interrupt Controller during an INTA cycle. See Section 6.2.
TB0 TB1	Bit 8 Bit 9	Timer Clock Select: Determine the source for the input clock signals for the 3 Counter/Timers. See Sec. 6.4.
CYCEXT	Bit 7	CYCEXT on the RTX 2000 and 2001A: When =1, extends bus cycle by 1 PCLK period for every INTA cycle or User Memory Instruction cycle. See Sec. 5.1.1 and 5.1.2. CYCEXT on the RTX 2010: Allows extended cycle length for User Memory Instruction cycles. See Sec. 5.1.
ROUND	Bit 6	On the RTX 2000 and RTX 2010: ROUND option; when set to 1, the least significant 16 bits of the multiplier output are rounded into the most significant 16 bits. See Section 6.3. On the RTX 2001A: Reserved; should be set to 0 during write operations.
DPRSEL	Bit 5	Data Page Register Select: Determines whether source of bits 16-19 of Memory Address Bus are from CPR or DPR for memory access instructions. See Sec. 5.2.2.
*	Bits 0-4 Bit 0 Bit 1 Bit 2 Bit 3 Bit 4	On the RTX 2000, these bits are reserved and should be set to 0. On the RTX 2010 and RTX 2001A, these are used as read-only stack controller flags where: Fatal Stack Error Flag; Parameter Stack Underflow Flag; Return Stack Underflow Flag; Parameter Stack Overflow Flag; Return Stack Overflow Flag.

mask: 0xFC00

mask: 0x0300

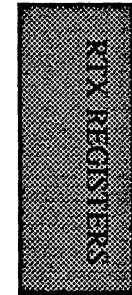
0x0080

0040

0020

mask:
0x001F

6.4.3



4.3.15 User Base Register, UBR - Address 11H

The contents of this register point to the beginning of a 32 word memory block which will be used for all User Memory Access instructions. See Section 5.2.3 for information about User Memory Space.

4.3.16 Address 12H

The function of this address is determined by the RTX processor being used.

4.3.16.1 On The RTX 2000 and RTX 2001A

This location is reserved on the RTX 2000 and RTX 2001A.

4.3.16.2 On The RTX 2010: MXR

The MAC Extension Register, MXR, is a 16-bit read/write register which holds the most significant 16 bits of the MAC Accumulator. For the Barrel Shifter instructions, this register holds the shift count. For the Leading Zero Detector instructions, the leading zero count is stored in this register.

4.3.17 Timer/Counter 0 - Address 13H

Reading this location pushes the current contents of Timer/Counter 0 onto the Parameter Stack. See Section 6.4 for more information about Timer/Counters.

Writing to this location loads the pre-load register for Timer/Counter 0.

4.3.18 Timer/Counter 1 - Address 14H

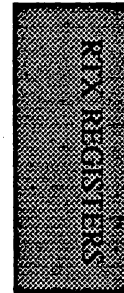
Reading this location pushes the current contents of Timer/Counter 1 onto the Parameter Stack. See Section 6.4 for more information about Timer/Counters.

Writing to this location loads the pre-load register for Timer/Counter 1.

4.3.19 Timer/Counter 2 - Address 15H

Reading this location pushes the current contents of Timer/Counter 2 onto the Parameter Stack. See Section 6.4 for more information about Timer/Counters.

Writing to this location loads the pre-load register for Timer/Counter 2.



4.3.20 Address 16H

Operations using this address depend upon whether the RTX 2000, RTX 2010, or the RTX 2001A Microcontroller is being used.

4.3.20.1 RTX 2000 - MLR

On the RTX 2000, this address is the Multiplier Low Register, MLR, and is used with the RTX 2000 on-chip hardware multiplier.

Reading this location pushes the lower 16 bits of the multiplier output onto the Parameter Stack. The contents of TOP are pushed into NEXT, but NEXT is not pushed onto the stack.

The MLR Register is a read-only register on the RTX 2000.

4.3.20.2 RTX 2001A - RX, Scratchpad/Counting Register

On the RTX 2001A, this address is the RX Register. The RX Register is a general purpose Read/Write scratch pad register. Special instructions are available to increment or decrement RX in one cycle. This allows the RX register to be easily utilized as a 16-bit program controlled counting register.

Incrementing the register contents beyond the "all ones" state results in a wrap to the "all zeros" state. Decrementing the register below the "all zeros" state results in a wrap to the "all ones" state.

4.3.20.3 RTX 2010 - MLR

On the RTX 2010, this address is for the Multiplier Low Register, MLR, and holds the least significant 16 bits of the 32-bit product generated by the on-chip hardware multiplier. This register is also used to hold the least significant 16 bits of the MAC Accumulator, and Barrel Shifter. See Section 6.3.2 for information about the Multiplier/Accumulator. The MLR can be read or written on the RTX 2010.

What happens for wrap? over ALU? swap ALU?

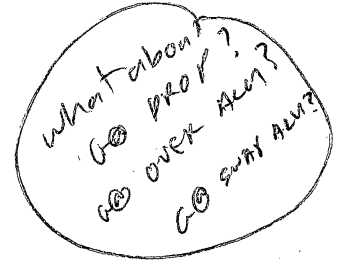
4.3.21 Address 17H

Operations at this address depend upon whether the RTX 2000 Microcontroller or the RTX 2001A Microcontroller is being used.

4.3.21.1 RTX 2000 - MHR

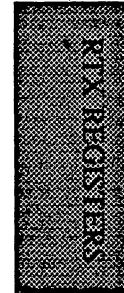
On the RTX 2000, the Multiplier High Register, MHR, is used with the on-chip hardware multiplier.

Reading this location pushes the upper 16 bits of the multiplier output onto the Parameter Stack. The contents of TOP are pushed into NEXT, but NEXT is not pushed onto the stack. The MHR Register is a read-only register on the RTX 2000.



4.3.21.2 RTX 2001A - RH

On the RTX 2001A, this address is for the RH Register. This is a 16-bit scratchpad register for data storage, which provides faster access than access to memory or a location buried in the stack.



4.3.21.3 RTX 2010 - MHR

On the RTX 2010, the Multiplier High Register, MHR, holds the most significant 16 bits of the 32-bit product generated by the on-chip hardware multiplier. If the IBC Register's ROUND bit is set, this register contains the rounded 16-bit output of the multiplier. In the Accumulator context, this register holds the middle 16 bits of the MAC, or the most significant 16 bits of the Barrel Shifter. See Section 6.3.2 for information about the Multiplier/Accumulator.

CHAPTER 5

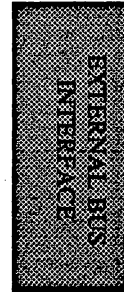
EXTERNAL BUS INTERFACE

EXTERNAL BUS
INTERFACE

5 External Bus Interfaces

Addresses for access to external memory or ASIC devices are output via either the Memory Address Bus (MA19-MA01) or the ASIC Address Bus (GA02-GA00).

External data is transferred by the ASIC Data Bus (GD15-GD00) and the Memory Data Bus (MD15-MD00), which are both bidirectional buses.



5.1 ASIC Bus Interface

The ASIC Bus services both internal processor core registers and the on-chip peripheral registers, and eight external off-chip ASIC Bus locations.

All ASIC Bus operations require a single cycle to execute and transfer a full 16-bit word of data. The external ASIC Bus maps into the last eight locations of the 32 location ASIC Address Space. The three least significant bits of the address are available as the ASIC Address Bus. See Table 5.1 for the address map.

TABLE 5.1: ASIC BUS MAP

ASIC BUS SIGNAL			ASIC ADDRESS
GA02	GA00	GA00	
0	0	0	18H
0	0	1	19H
0	1	0	1AH
0	1	1	1BH
1	0	0	1CH
1	0	1	1DH

5.1.1 RTX 2000 and RTX 2001A Extended Cycle Operation

On the RTX 2000 and RTX 2001A, bus cycle timing can be extended by 1 PCLK period to allow the use of some slow memory devices without requiring the addition of external Wait states. When the CYCEXT bit (IBC bit 7) is set equal to 1, extended cycles are used for all User Memory and Interrupt Acknowledge cycles.

EXTERNAL BUS
INTERFACE

5.1.2 RTX 2010 Extended Cycle Operation

On the RTX 2010, the user has the option of independently extending bus cycle operations by 1 PCLK period for either User Memory Cycles or for ASIC Bus Read operations. This provides the ability to interface to some peripherals and slow memory devices without using externally generated Wait states.

Setting the Cycle Extend bit (CYCEXT), bit 7 of the IBC Register, will cause extended cycles to be used for all accesses to User memory.

Setting the ASIC Read Cycle Extend bit (ARCE), bit 13 of the CR Register, will cause extended cycles to be used for all Read accesses on the external ASIC Bus.

Both the CYCEXT bit and the ARCE bit are cleared on Reset.

5.2 Memory Interface

The RTX processors directly address 512K words of memory, divided into 16 pages of 32K words each.

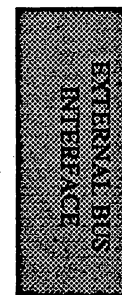
The memory page currently being addressed is selected by one of three 4-bit "address page" registers, depending on what type of memory access is being performed.

The RTX addresses 3 types of memory space, each with an associated address page register. These are Code space, Data space, and User space.

- **Code Memory Space** is accessed by all instruction fetch operations. See Section 5.2.1.
- **Data Memory Space** refers to all memory locations accessed by memory reference instructions. See Section 5.2.2.
- **User Memory Space** provides efficient access to a block of 32 words which may reside anywhere in the processor's memory space. See Section 5.2.3.

The RTX instruction set includes classes of instructions for referencing each type of memory space. With the exception of instruction fetches and streamed MAC operations, RTX memory accesses involve the TOP and NEXT registers.

The TOP register contains the address of the memory location to be read or written. The NEXT Register interfaces to the Memory Data Bus. For memory writes, the value contained in NEXT is written to the location addressed by the contents of TOP. For memory reads, the contents of the memory location addressed by the contents of TOP are loaded into NEXT, then the stack is popped, dropping the address and leaving the memory data in TOP.



The RTX's memory reference instructions have various forms which determine the net stack effect of the memory read or write. Depending on the instruction format, the contents of TOP and NEXT may be overwritten by memory data, preserved on the stack, or modified through ALU operations.

The RTX's 20-bit Memory Address Bus is composed of the 16-bit address from the TOP register, and 4 bits from the appropriate address page register.

The Code Page Register is used for all references to Code memory space, and the Data Page or Code Page Register for all references to Data Space. The Code and Data Page Registers may point to the same memory page, as in a system containing all RAM memory, or to different pages, as in a system with mixed ROM and RAM. Additionally, the CPR and DPR may point to the same page for small RAM/ROM systems. User Space addresses are a special case, and are discussed in Section 5.2.3.

The page address registers may be read or written by using ASIC Bus access instructions (see Chapter 7). The registers may be read at any time to determine the current active memory pages.

5.2.1 Code Memory Space

Code memory space contains machine instructions to be executed by the RTX processor.

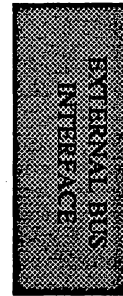
5.2.1.1 Subroutine Calls and Returns

RTX subroutine calls take place within the memory page specified by the Code Page Register. Any instruction with Bit 15 (the most significant bit) set to 0 will cause a subroutine call to the address contained in the lower 15 bits of the instruction. The address to be called is calculated by shifting the value contained in the instruction left by one bit and inserting a zero in the least significant bit. For example, the machine instruction 3211H (Hex) will cause a subroutine call to location 6422H. See Table 5.2.

Long Calls may be made to a memory page other than the current Code page by first loading the appropriate page number into the Code Page Register, then executing the subroutine call.

Loading a value into the Code Page Register performs two special functions. First, the effect of loading the Code Page Register is delayed by one instruction, so that the instruction following the load instruction is fetched from the current code page. Second, interrupts are disabled for one clock cycle following the load instruction. This guarantees that the instruction following the load (typically the Call instruction) will be executed without an intervening Interrupt Service Routine which might corrupt the contents of the registers.

Subroutine calls save their 21-bit return address on the Return Stack (the top element is composed of the Index Register and the Index Page Register). As subsequent calls occur, the storing of subroutine return addresses in IPR and I causes the previous contents of IPR and I to be pushed into the Return Stack. The 21-bit return address is contained in the Index Register and the Index Page Register and consists of the following:



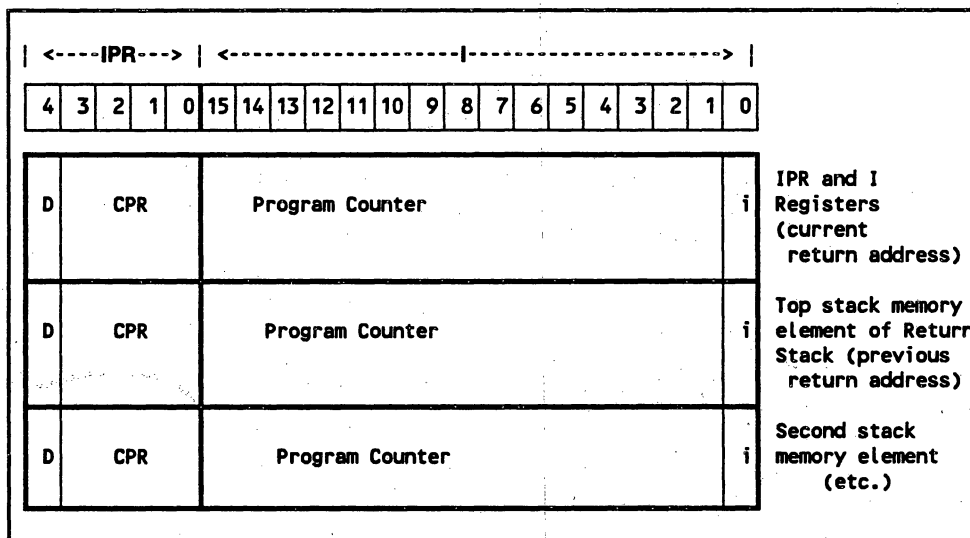
and the current value of DPRSEL

Index Register (I), at address 01H

- Bit 0 Set to 1 if call results from an interrupt acknowledge, 0 otherwise. As indicated by "i" in Figure 5.2.
- Bits 1-15 Word address to which to return (bits 1-15 of Program Counter). The least significant bit of the return address is implicitly 0 since instructions are always fetched on word boundaries.

Index Page Register (IPR)

- Bits 0-3 Code page to which to return
- Bit 4 Value of DPRSEL bit (see description of Data Memory). As indicated by "D" in Figure 5.2.

**FIGURE 5.2: RETURN STACK STRUCTURE**

Example: Code executing at location 1220H in Code Page 3 calls a subroutine located at address 3322H in Code Page 4. See Table 5.2.

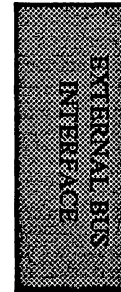


TABLE 5.2: SAMPLE SUBROUTINE CALL AND RETURN

Cycle	Code Page	Address (PC)	Actual Address (MA19-MA01)	Instruction
1	3	1220	31220	Set CPR = 4
2	3	1222	31222	Call location 3322H
3	4	3322	43322	1st subroutine instruction
4	4	3324	43324	2nd instruction
5	4	3326	43326	Return From Subroutine
6	3	1224	31124	1st instruction after Call

5.2.1.2 Branching

Branching instructions work similarly to subroutine calls. Branches may be performed across page boundaries by first loading the Code Page Register with the new page number (the current page plus 1 for forward branches, the current page minus one for backward branches). See Chapter 7 for specific details on branch instructions.

If the instruction following a "Load Code Page Register" instruction is not a Call or Branch, it will be executed, then the next instruction will be fetched from the memory page specified by the new contents of the Code Page Register.

*DPR SEC
is restored
when
executing
return*

5.2.2 Data Memory Space

Data Memory refers to memory locations accessed by the RTX's "Data Memory Access" class of instructions. These would typically be RAM locations used for variables and data storage.

5.2.2.1 Memory Page Selection

The memory page referenced by data memory instructions may be selected by either the Data Page Register or the Code Page Register. The DPRSEL bit (Data Page Register Select bit, IBC bit 5) determines which register will be used.

When $DPRSEL = 0$, all main memory accesses will occur in the memory page addressed by the Code Page Register. This is the default mode. In this mode, code and data memory spaces reside in the same memory page, and would typically be used in systems with 64K bytes or less of memory. In such a system, the Memory Address Bus bits generated by the page select logic (bits MA19-MA16) would not be required.

When $DPRSEL = 1$, all main memory accesses will occur in the memory page addressed by the Data Page Register. The Data Page Register may point to the same page as the Code Page Register, or to a separate page.

The state of the DPRSEL bit may be controlled through three methods. First, DPRSEL is directly readable and can be set as a bit in the Interrupt Base/Control register. Second, it may be set or reset in one clock cycle by special forms of the Register read/write instructions pertaining to the Data Page Register (see "Predefined ASIC Bus Instructions" in Chapter 7). Third, DPRSEL is saved as bit 4 of the Index Page Register during subroutine calls. The value in IPR may be modified by a subroutine; the new value will be written into the DPRSEL bit and take effect as soon as a Subroutine Return instruction is executed.

5.2.2.2 Memory Access Mode Selection

To support the use of shared memory interfaces with other processors, the RTX can be configured to access Data Memory in either of two modes which determine whether the byte order in memory will be High-Low (Mode 0) which is the default mode, or Low-High (Mode 1). Bit 2 of the Configuration Register is used to select the Data Memory Access Mode.

The default, Mode 0, is selected when CR bit 2 = 0. This means that the most significant byte of data in the processor register (NEXT) will be read from or written to the even byte address in memory, and the least significant byte of data in NEXT will be read from or written to the odd byte address in memory.

Mode 1 is selected when CR bit 2 = 1. This means that the most significant byte of data in NEXT will be read from or written to the odd byte address in memory, and the least significant byte of data in NEXT will be read from or written to the even byte address in memory. See Figure 5.3.

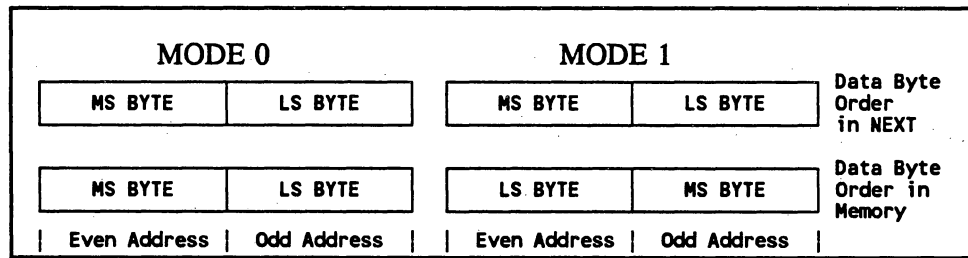


FIGURE 5.3: MEMORY ACCESS MODES

In addition to allowing selection of byte order, the RTX allows the user to choose between accessing Data Memory in either 16-bit words or 8-bit bytes.

→ what does this mean?

5.2.2.3 Memory Access Examples

Byte reads from locations 1000H and 1001H will both read a byte from word address 1000H. CR bit 2 and bit 0 of TOP determine which byte of the memory location will be accessed. For 8-bit writes, only bits 0-7 of NEXT are transferred to memory. For 8-bit reads, data from memory is transferred into bits 0-7 of NEXT; bits 8-15 of NEXT are set to 0.

Example: Reading and Writing a 16-bit value (1234H) to memory location 1000H (all values are in hexadecimal) yields the results shown in Figure 5.6 at the end of the first cycle.

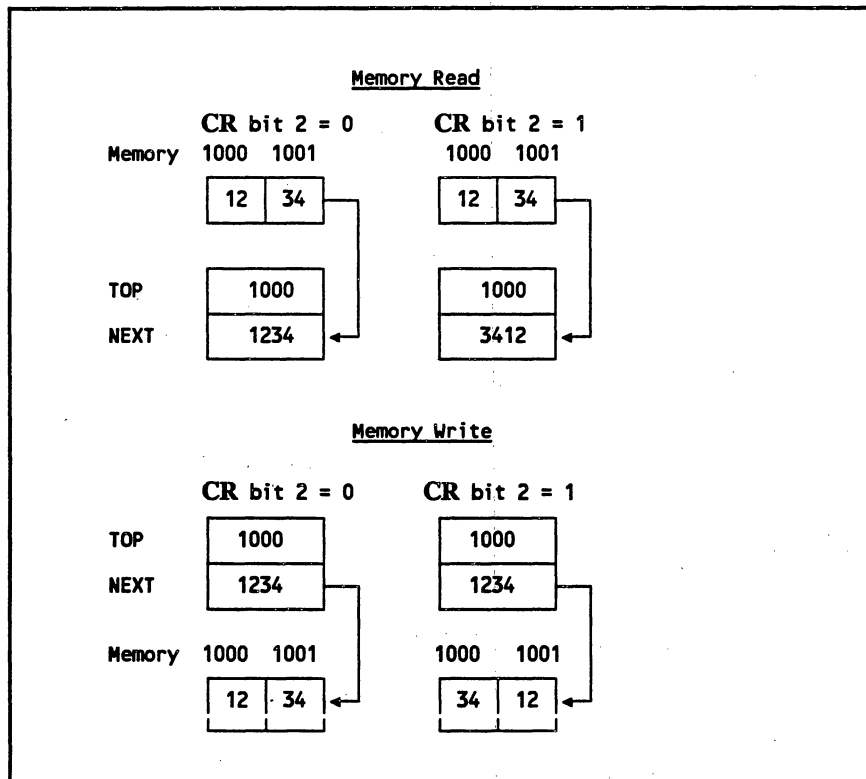


FIGURE 5.6: 16-BIT READ/WRITE TO EVEN MEMORY ADDRESS

The least significant bit of the memory address contained in TOP may also be used to control the Byte-swapping feature. If the LSB of TOP is 1 when performing a 16-bit memory access, then an odd byte address is being accessed. This means that the same word address will be read or written, but the bytes of the word read or written to memory will be swapped.

Accessing a word with the LSB of the address set to 1 effectively inverts the Byte Order bit.

Example: Reading and Writing a 16-bit value (1234H) to memory location 1001H (all values are in hexadecimal) yields the results shown in Figure 5.7 at the end of the first cycle.

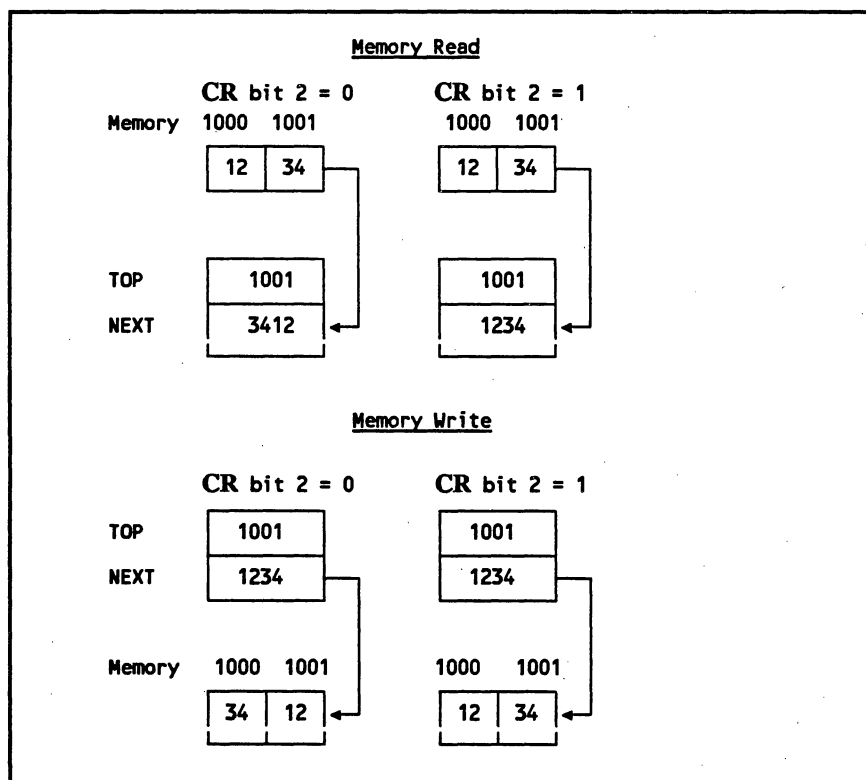


FIGURE 5.7: 16-BIT READ/WRITE TO ODD MEMORY ADDRESS

The Byte Order bit also affects 8-bit memory accesses. If the Byte Order bit is set to 1, the LSB of the address contained in TOP is inverted before performing the memory read or write. Following are two examples.

Example: Reading and Writing an 8-bit value to memory location 1000H yields the the results shown in Figure 5.8 at the end of the first cycle:

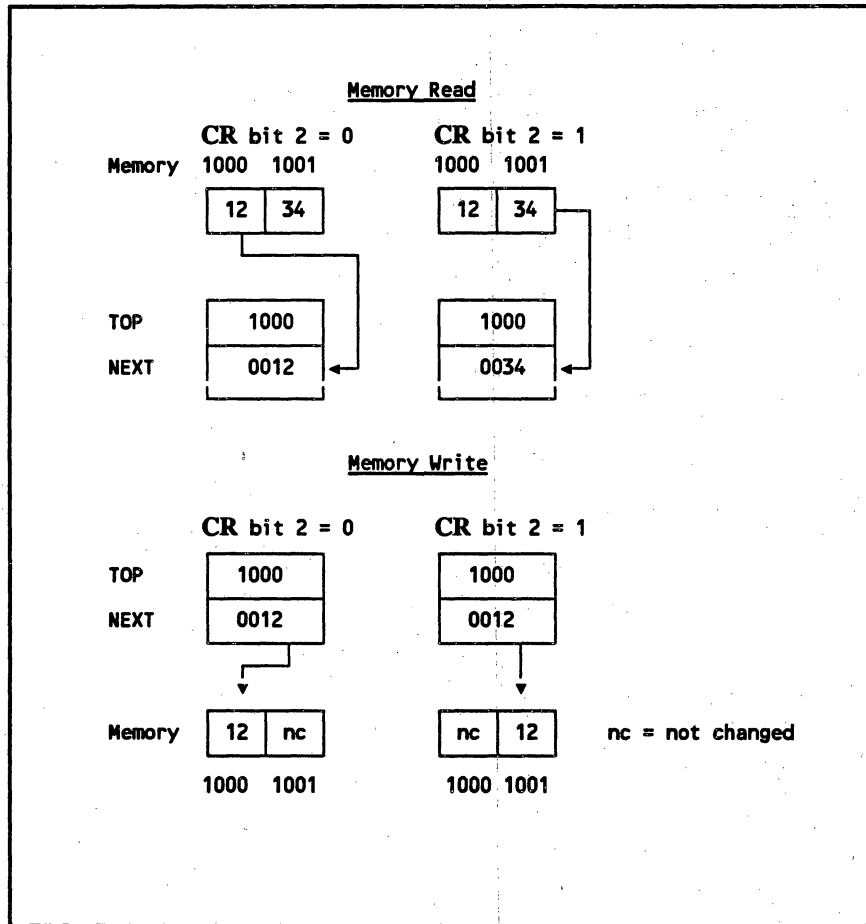


FIGURE 5.8: 8-BIT READ/WRITE TO EVEN MEMORY ADDRESS

Example: Reading and Writing an 8-bit value to memory location 1001H yields the results shown in Figure 5.9 at the end of the first cycle.

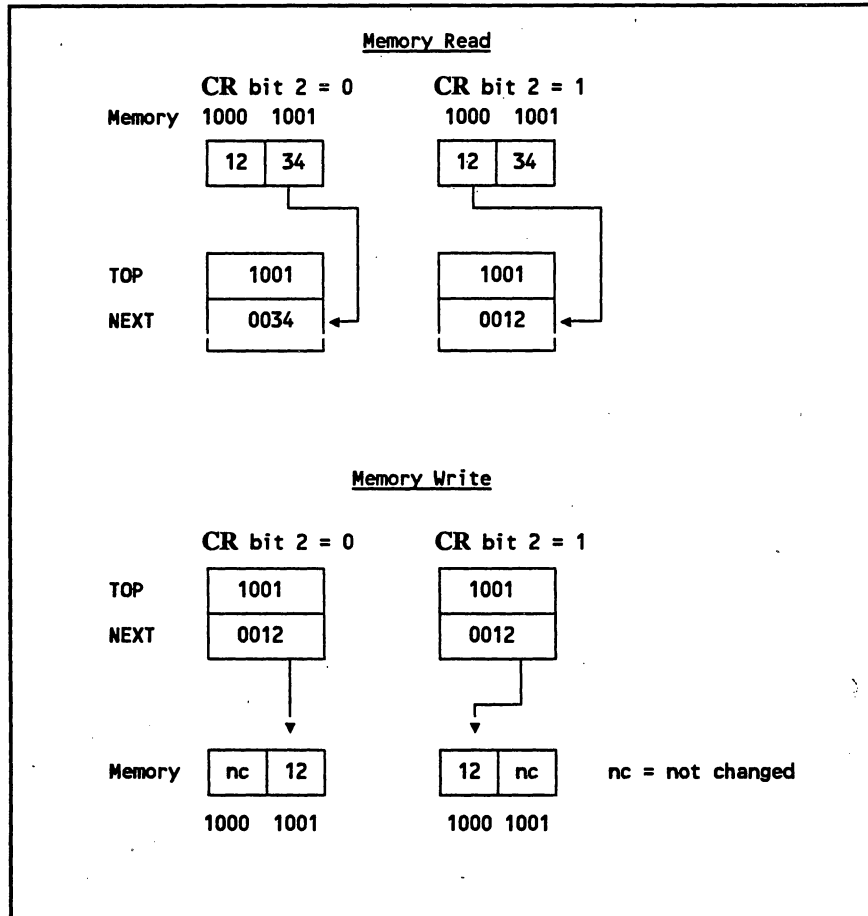


FIGURE 5.9: 8-BIT READ/WRITE TO ODD MEMORY ADDRESS

5.2.3 User Memory Space

User Memory space is a block of 32 words which the RTX processor can access without having to first calculate an address and load it into TOP. The logical address to be referenced within the 32-word block is embedded in the machine instruction which accesses the memory location.

User Memory space would typically be used to hold data which must be accessed frequently, such as system parameters or context save areas in a multi-tasking system. See Chapter 7 for descriptions of the User Memory Reference Instructions.

The physical address to be accessed when addressing User Memory space is derived from several components, shown in Figure 5.10.

The User Page Register (UPR bits 03-00) points to the memory page to be used for User Memory Access. The User Base Address Register (UBR bits 15-06) contains the offset for the particular 32-word block to be accessed by User Memory Instructions. The exact word in the 32-word User block to be accessed is specified by the address contained in the lower 5 bits of the User Access Instruction.

As indicated in Figure 5.10, bits 05-01 of the UBR Register are logically OR'ed with the 5 address bits embedded in the User Access Instruction (IR bits 04-00), and the results yield the next five memory address bits (MA05-MA01). Because of the logical OR which takes place, only the ten most significant bits of UBR should be used to specify the User Base Address, since setting the lower bits will have the effect of reducing the user block size by duplicating addresses.

Finally, since User accesses are always on word boundaries, bit 0 of the UBR should always be zero.

Table 5.3 provides some samples of addresses, as determined by the contents of UPR, UBR, and IR.

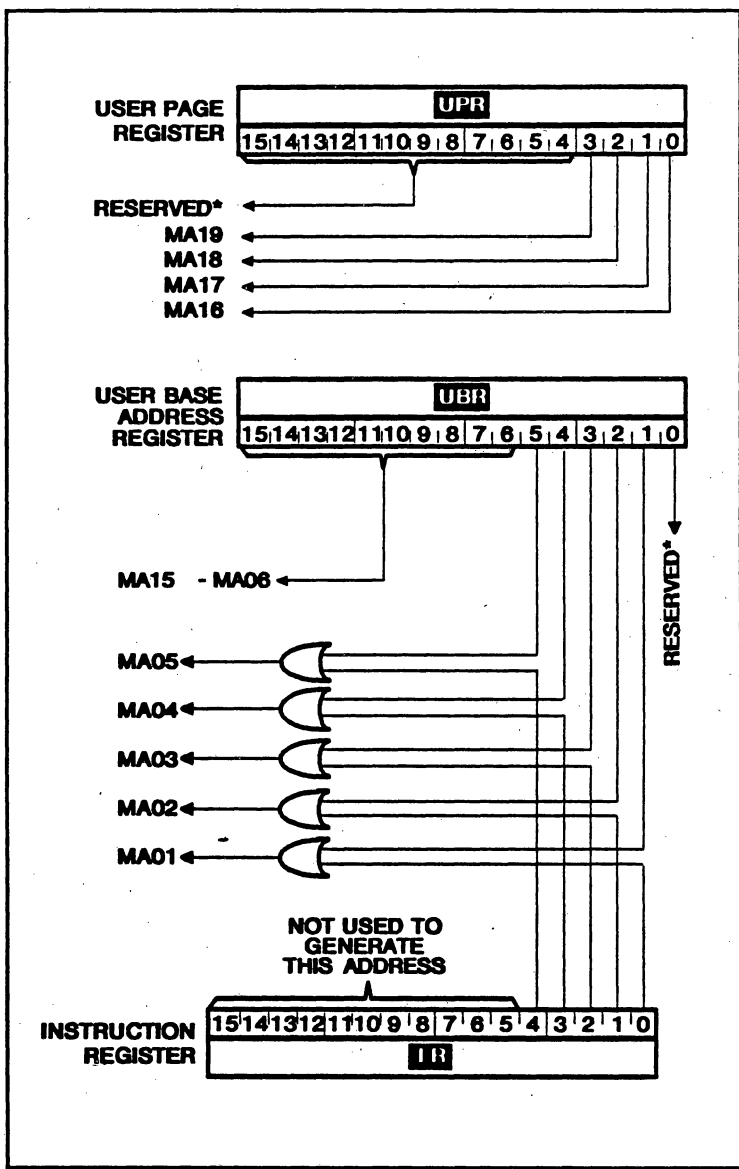


FIGURE 5.10: USER MEMORY ADDRESS COMPONENTS

TABLE 5.3: USER MEMORY ADDRESS EXAMPLES

UPR (4 bits)	UBR (10 bits)	Address field (5 bits) from IR	Actual Address (20 bits)
0H	1240H	03H	01246H
2H	3140H	0FH	2315EH
2H	3310H	1FH	2333EH

Note that in the third example some locations within the 32-word block will not be accessible because bit 4 in the User Base Register is set to 1 and will cause the corresponding bit of the address to always be set due to the OR operation.

By adjusting the contents of the User Page Register and User Base Register, an application may have any number of 32-word User spaces (up to 1 megabyte).

The byte-swapping operations described for the Data Memory accesses do not affect User Memory accesses.

CHAPTER 6

ON-CHIP PERIPHERALS

6 On-Chip Peripherals

The RTX 2000 Series microcontrollers contain hardware to support many of the functions typically needed in real-time control systems. These include two Stack Controllers, an Interrupt Controller, and three 16-bit Counter/Timers.

In addition, the RTX 2000 offers an on-chip 16-by-16 Hardware Multiplier, while the RTX 2010 offers an on-chip Multiplier/Accumulator, Leading Zero Detector, 32-bit Barrel Shifter, hardware floating point support, and multi-tasking stack support.

All on-chip peripheral devices are accessible through the ASIC Bus by the use of ASIC Bus Read and Write instructions. The contents of the TOP register may be written to the devices, and the outputs of the devices may be read through the ALU into the TOP register.

This section contains the information necessary for programming the On-Chip Peripheral devices. Refer to Chapter 4 for more information about the ASIC Bus addresses for each device.

6.1 Stack Controllers

Each RTX Microcontroller contains two identical stack controller circuits, one for the Parameter Stack, and one for the Return Stack. The RTX Stack Controllers utilize stack pointers and stack limits for control of stack operations. Specific details of how the stack controllers work are determined by the type of processor being used.

On the RTX 2000, operation of the Programmable Stack Controllers depends on the contents of two registers, the Stack Pointer Register (SPR), and the Stack Limit Register (SLR).

On the RTX 2001A, operation of the Programmable Stack Controllers depends on the contents of three registers. These registers are the Stack Pointer Register (SPR), the Stack Overflow Limit Register (SVR), and the Stack Underflow Limit Register (SUR).

On the RTX 2010, operation of the Programmable Stack Controllers depends on the contents of three registers. These registers are the Stack Pointer Register (SPR), the Stack Overflow Limit Register (SVR), and the Stack Underflow Limit Register (SUR). To use these registers to perform Multitasking operations, see Section 6.1.3.2.

6.1.1 Stack Pointer Operation

The Stack Pointers for both stacks are combined into one 16-bit register for access through the ASIC Bus. This register may be used to read and write both stack pointers in parallel. The stack pointers are used to determine the "top" location in stack memory for each stack.

6.1.1.1 Stack Pointers For the RTX 2000

On the RTX 2000, the value for each stack pointer is initialized to a value of 0 at reset, and can range from 0 to 255. Each stack pointer indicates the position of the "top" item in stack memory, which contains the data that was most recently pushed into the stack. See Figure 6.1.

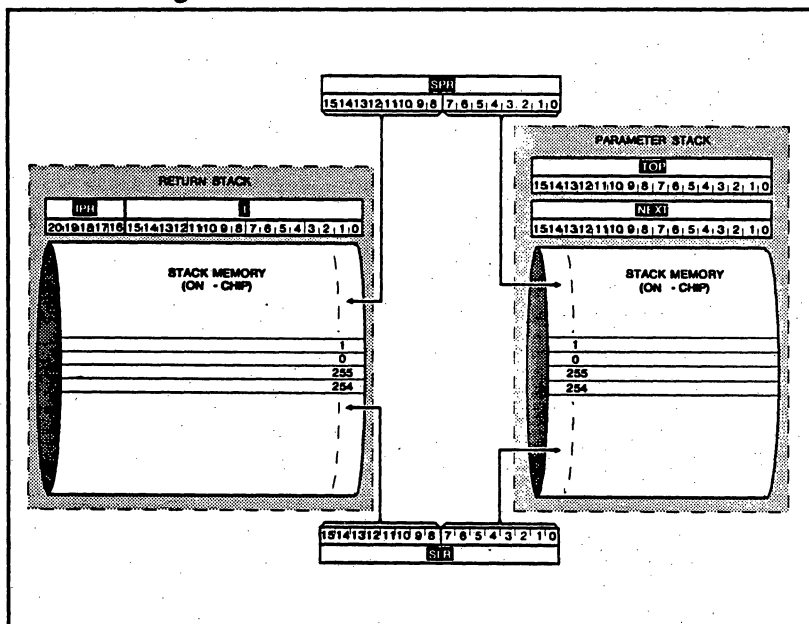


FIGURE 6.1: RTX 2000 STACK CONTROL

The Stack Pointer Register is at ASIC address 09H, and may be used to read and write both stack pointers. Bits 0-7 contain the stack pointer value for the Parameter Stack, while bits 8-15 contain the pointer value for the Return Stack (see Figure 6.2).

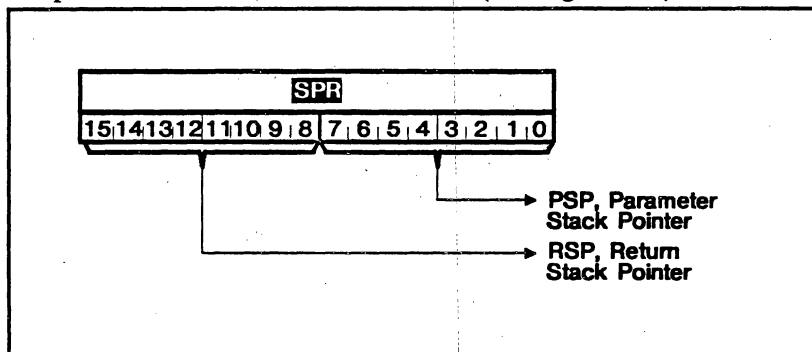


FIGURE 6.2: RTX 2000 STACK POINTER REGISTER

During a stack push operation, the SPR is incremented by 1 before the new item is pushed onto the stack (i.e., when the operation begins, the register contains the address of the next stack location to be written for each stack). The Stack Pointer may be set to a new value by writing to SPR; the value written to the register should be one less than the address of the first location to be written.

During a stack Read operation, the pointer indicates the next item which can be popped from the stack memory. After that item has been popped, the stack pointer is decremented by 1. Since reading the stack pointer pushes a value onto the Parameter Stack, the value read will be 2 more than the number of items on the Parameter Stack prior to reading the Stack Pointer Register.

Stack Underflow on the RTX 2000 - The SPR monitors the total number of items on the stacks, and will generate a "stack underflow" interrupt request if more items are popped from the stack than were pushed onto it. The underflow signals are fed to the Interrupt Controller (see Section 6.2) and may be masked through the Interrupt Mask Register (IMR).

6.1.1.2 RTX 2001A and RTX 2010 Stack Pointers

On the RTX 2001A and RTX 2010, the value for each stack pointer is initialized to a value of 0 at reset. On the RTX 2001A the stack pointer values can range from 0 to 63; on the RTX 2010 they can range from 0 to 256.

Each stack pointer indicates the position of the "top" item in stack memory, which contains the next stack element to be accessed in a stack write operation. After a stack write ("push") operation, the stack pointer is incremented.

In a stack read operation, the stack memory location with an address one less than the pointer location will be accessed. After a stack read ("pop") the pointer is decremented. See Figure 6.3.

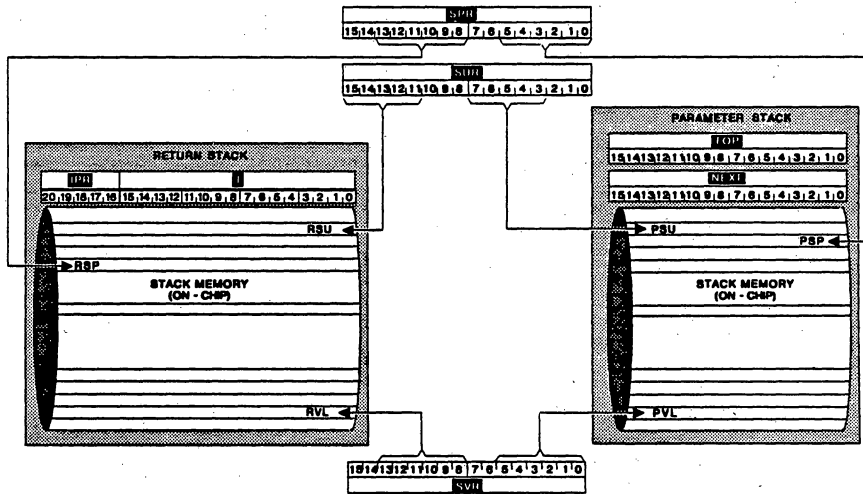


FIGURE 6.3: RTX 2001A/2010 STACK CONTROL

On the RTX 2001A, bits 0-5 contain the stack pointer value for the Parameter Stack, while bits 8-13 contain the pointer value for the Return Stack. See Figure 6.4.

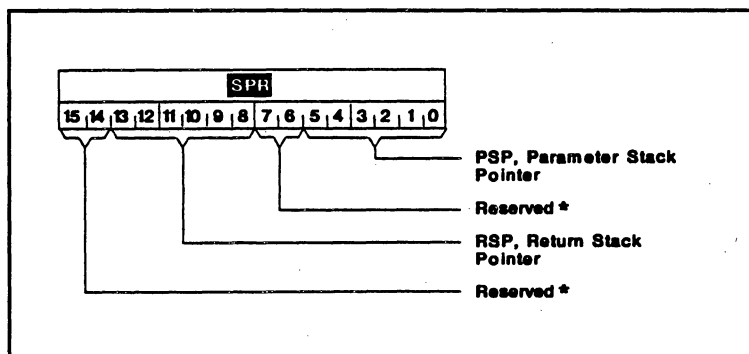


FIGURE 6.4: RTX 2001A STACK POINTER REGISTER

On the RTX 2010, bits 0-7 contain the stack pointer value for the Parameter Stack, while bits 8-15 contain the pointer value for the Return Stack. See Figure 6.4.

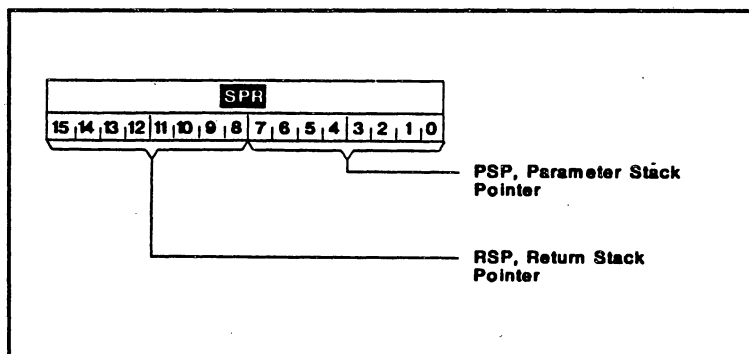


FIGURE 6.5: RTX 2010 STACK POINTER REGISTER

6.1.2 Stack Limit Operation

Stack limits are used to prevent data stored in the stack from being overwritten. Since the stacks wrap around, existing data on the stack will be overwritten by the new data when an overflow occurs. Underflows occur when an attempt is made to pop data off an empty stack, causing invalid data to be read from the stack. Since the processor can take up to four clock cycles to respond to an interrupt, the values set into the stack limit registers should include a safety margin which allows valid stack operation until the processor executes the interrupt service routine.

On the RTX 2000, RTX 2010, and RTX 2001A, a buffer zone may be set up so that stack error interrupts are generated prior to an actual overflow. In addition, the RTX 2001A and RTX 2010 Underflow Limit Registers provide the capability to define an underflow buffer.

The RTX 2000 Family processors utilize ASIC Address 0BH for the 16-bit, write-only register which contains the maximum stack size limits for the Parameter and the Return Stacks. On the RTX 2000, this register is called the Stack Limit Register, (SLR). On the RTX 2001A and RTX 2010, it is called the Stack Overflow Limit Register, (SVR).

6.1.2.1 Stack Limits For the RTX 2000

On the RTX 2000, the maximum limit for the Parameter Stack is in bits 0-7 of the Stack Limit Register; bits 8-15 contain the maximum limit for the Return Stack (see Figure 6.6). These limit values determine the number of items which may be pushed onto each stack before the Interrupt Controller will generate a "Stack Overflow" interrupt signal. The Limit Register for both stacks must be initialized on powerup or reset, if stack error interrupts are to be used.

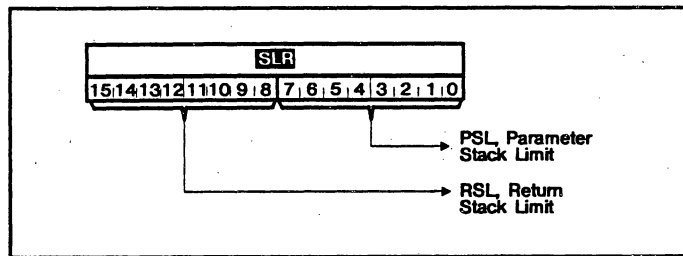


FIGURE 6.6: RTX 2000 STACK LIMIT REGISTER

6.1.2.2 Stack Limits For the RTX 2001A

The RTX 2001A and RTX 2010 Microcontrollers utilize two registers to provide stack limit control. They are the Stack Overflow Limit Register, SVR at ASIC address 0BH, and the Stack Underflow Limit Register, SUR at ASIC address 0AH; SVR is write-only register.

Overflow limits: The overflow limit is the number of items which may be pushed onto the stack before an interrupt will be detected. Bits 0-5 of the Stack Overflow Limit Register contain the maximum limit for the Parameter Stack, and bits 8-13 contain the maximum limit for the Return Stack (see Figure 6.7).

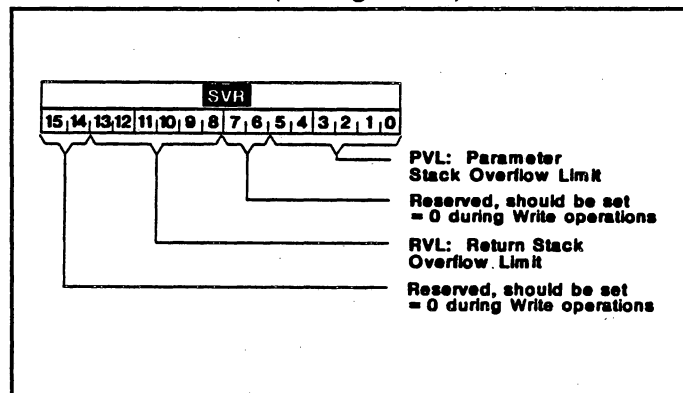


FIGURE 6.7: RTX 2001A STACK OVERFLOW LIMITS

Underflow limits: Bits 3-7 of the Stack Underflow Limit Register contain the underflow limit for the Parameter Stack, and bits 11-15 contain the underflow limit for the Return Stack. See Figure 6.8.

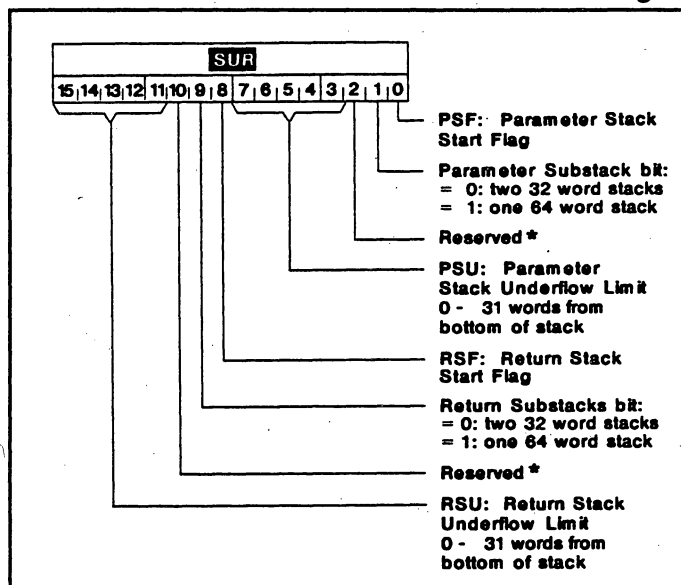


FIGURE 6.8: RTX 2001A STACK UNDERFLOW LIMITS

6.1.2.3 Stack Limits For the RTX 2010

The RTX 2010 Microcontroller utilizes two registers to provide stack limit control. They are the Stack Overflow Limit Register, SVR at ASIC address OBH, and the Stack Underflow Limit Register, SUR at ASIC address OAH.

Overflow limits: The overflow limit is the number of items which may be pushed onto the stack before an interrupt will be detected. Bits 0-7 of the Stack Overflow Limit Register contain the maximum limit for the Parameter Stack, and bits 8-15 contain the maximum limit for the Return Stack (see Figure 6.9).

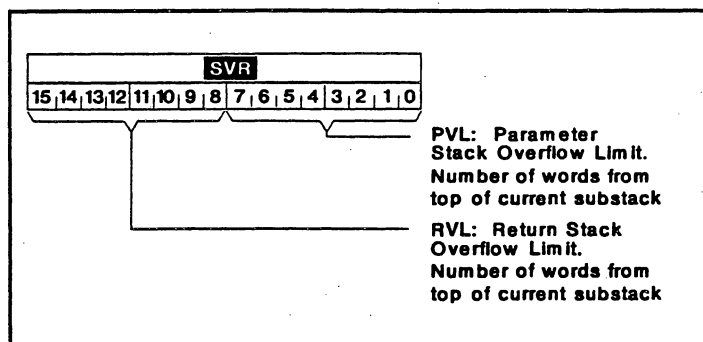


FIGURE 6.9: RTX 2010 STACK OVERFLOW LIMITS

Underflow limits: Bits 3-7 of the Stack Underflow Limit Register contain the underflow limit for the Parameter Stack, PSU, and bits 11-15 contain the underflow limit for the Return Stack, RSU. See Figure 6.10. In addition, this register is utilized to define the use of substacks for both stacks (see Section 6.1.3). All Stack Underflow Limit Register values must be accessed together.

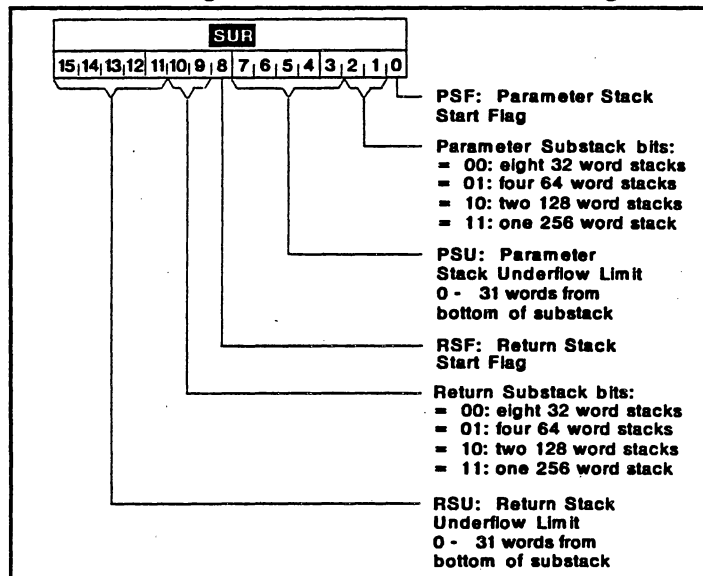


FIGURE 6.10: RTX 2010 STACK UNDERFLOW LIMITS

6.1.3 Configuration Of Substacks

The enhanced Stack Controller logic on the RTX 2001A and RTX 2010 allows the stack related registers to be used for configuring substacks.

6.1.3.1 Substack Configuration On The RTX 2001A

Each 64-word stack may be subdivided into two substacks under hardware control for simplified management of multiple tasks. Each substack is 32 words deep. Stack size is selected by writing to bit 1 of the Stack Underflow Limit Register for the Parameter Stack, and bit 9 for the Return Stack. See Figure 6.7.

Substacks are implemented by making bits 5 or 13 of the Stack Pointer Register control bits (i.e. they are not incremented when the stack size is 32 words). Using this, a particular substack is selected by writing a value which contains both the stack pointer value and the substack number to the Stack Pointer Register.

Each stack has a Stack Start Flag which may be used for virtual stacks. This is bit 0 of the SUR for the Parameter Stack, and bit 8 of the SUR for the Return Stack. If the Stack Start Flag is one, the stack starts at the bottom of the stack or substack (location 0). If the Stack Start Flag is 0, the substack starts in the middle of the stack. In a stack 64 elements deep, this is location 32; In a stack 32 elements deep, this is location 16. An exception to this occurs if the overflow limit in the Stack Overflow Limit Register is set for a location below the middle of the stack. In this case, the stacks always start at the bottom locations.

Manipulating the Stack Start Flag provides a mechanism for creating a virtual stack in memory which is maintained by interrupt driven handlers.

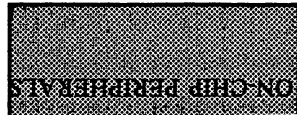
Possible applications for substacks include use as a recirculating buffer (to allow quick access for a series of repeated values such as coefficients for polynomial evaluation or a digital filter), or to log a continuous stream of data until a triggering event (for analysis of data before and after the trigger without having to store all of the incoming data), as in the case of a digital oscilloscope or logic analyzer.

See Table 6.1 for control bit settings for possible stack/substack configurations. In Table 6.1, note the following:

1. **SPR** is the Stack Pointer Register; **SVR** is the Stack Overflow Limit Register; **SUR** is the Stack Underflow Limit Register.
2. **P0** through **P15** are the **SPR** bits; **V0** through **V15** are the **SVR** bits; **U0** through **U15** are the **SUR** bits.
3. The Overflow Limit is the stack memory address at which an overflow condition will occur during a stack write operation.
4. The Underflow limit is the stack memory address below which an underflow condition will occur during a stack read operation.
5. The Fatal limit is the stack memory address at which a fatal error condition will occur during a stack read or write operation.
6. Stack error conditions remain in effect until a new value is written to the **SPR**.
7. Stacks and substacks are circular. After writing to the highest location in the stack, the next location to be written to will be the lowest location; after reading the lowest location, the highest location will be read next.

TABLE 6.1: RTX 2001A SUBSTACK CONFIGURATION

CONTROL BIT SETTINGS:							PARAMETER STACK CONFIGURATION:														
SPR	SVR		SUR			STACK SIZE (WORDS)	STACK RANGE		FATAL LIMIT	UNDERFLOW LIMIT						OVERFLOW LIMIT					
	P5	V5	V4	U2	U1		U0	LOWEST ADDRESS		HIGHEST ADDRESS	5	4	3	2	1	0	5	4	3	2	1
0	X	0	0	0	X	32	0	31	31	0	0	U6	U5	U4	U3	0	0	V3	V2	V1	V0
0	X	1	0	0	0	32	0	31	15	0	1	U6	U5	U4	U3	0	0	V3	V2	V1	V0
0	X	1	0	0	1	32	0	31	31	0	0	U6	U5	U4	U3	0	1	V3	V2	V1	V0
1	X	0	0	0	X	32	32	63	63	1	0	U6	U5	U4	U3	1	0	V3	V2	V1	V0
1	X	1	0	0	0	32	32	63	47	1	1	U6	U5	U4	U3	1	0	V3	V2	V1	V0
1	X	1	0	0	1	32	32	63	63	1	0	U6	U5	U4	U3	1	1	V3	V2	V1	V0
X	0	X	0	1	X	64	0	63	63	0	U7	U6	U5	U4	U3	0	V4	V3	V2	V1	V0
X	1	X	0	1	0	64	0	63	31	1	U7	U6	U5	U4	U3	0	V4	V3	V2	V1	V0
X	1	X	0	1	1	64	0	63	63	0	U7	U6	U5	U4	U3	1	V4	V3	V2	V1	V0
CONTROL BIT SETTINGS:							RETURN STACK CONFIGURATION:														
SPR	SVR		SUR			STACK SIZE (WORDS)	STACK RANGE		FATAL LIMIT	UNDERFLOW LIMIT						OVERFLOW LIMIT					
	P13	V13	V12	U10	U9		U8	LOWEST ADDRESS		HIGHEST ADDRESS	5	4	3	2	1	0	5	4	3	2	1
0	X	0	0	0	X	32	0	31	31	0	0	U14	U13	U12	U11	0	0	V11	V10	V9	V8
0	X	1	0	0	0	32	0	31	15	0	1	U14	U13	U12	U11	0	0	V11	V10	V9	V8
0	X	1	0	0	1	32	0	31	31	0	0	U14	U13	U12	U11	0	1	V11	V10	V9	V8
1	X	0	0	0	X	32	32	63	63	1	0	U14	U13	U12	U11	1	0	V11	V10	V9	V8
1	X	1	0	0	0	32	32	63	47	1	1	U14	U13	U12	U11	1	0	V11	V10	V9	V8
1	X	1	0	0	1	32	32	63	63	1	0	U14	U13	U12	U11	1	1	V11	V10	V9	V8
X	0	X	0	1	X	64	0	63	63	0	U15	U14	U13	U12	U11	0	V12	V11	V10	V9	V8
X	1	X	0	1	0	64	0	63	31	1	U15	U14	U13	U12	U11	0	V12	V11	V10	V9	V8
X	1	X	0	1	1	64	0	63	63	0	U15	U14	U13	U12	U11	1	V12	V11	V10	V9	V8



6.1.3.2 Substack Configuration On The RTX 2010


Each 256-word stack may be subdivided into up to eight 32-word substacks, four 64-word substacks, or two 128-word substacks. This is accomplished under hardware control for simplified management of multiple tasks. Stack size is selected by writing to bits 1 and 2 of the SUR for the Parameter Stack, and bits 9 and 10 for the Return Stack.

Substacks are implemented by making bits 5-7 of the SPR (for the Parameter Stack) and bits 13-15 of the SPR (for the Return Stack) control bits. For example, if there are eight 32-word substacks implemented in the Parameter Stack, bits 5-7 of the SPR are not incremented, but instead are used as an offset pointer into the Parameter Stack to indicate the beginning point (i.e. substack number) of each 32-word substack implemented. Because of this, a particular substack is selected by writing a value which contains both the stack pointer value and the substack number to the SPR.

Each stack has a Stack Start Flag which may be used for virtual stacks. This is bit 0 of the SUR for the Parameter Stack, and bit 8 of the SUR for the Return Stack.

If the Stack Start Flag is one, the stack starts at the bottom of the stack or substack (location 0). If the Stack Start Flag is 0, the substack starts in the middle of the stack. In a stack 256 elements deep, this is location 128; In a stack 128 elements deep, this is location 64; In a stack 64 elements deep, this is location 32; In a stack 32 elements deep, this is location 16.

An exception to this occurs if the overflow limit in the Stack Overflow Limit Register is set for a location below the middle of the stack. In this case, the stacks always start at the bottom locations. See Tables 6.2 and 6.3 for the possible stack configurations. Manipulating the Stack Start Flag provides a mechanism for creating a virtual stack in memory which is maintained by interrupt driven handlers.



Possible applications for substacks include use as a recirculating buffer (to allow quick access for a series of repeated values such as coefficients for polynomial evaluation or a digital filter), or to log a continuous stream of data until a triggering event (for analysis of data before and after the trigger without having to store all of the incoming data). The latter application could be used in a digital oscilloscope or logic analyzer.

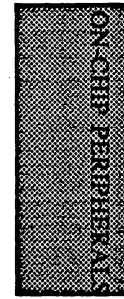


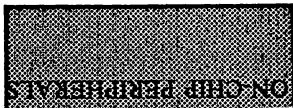
TABLE 6.2: RTX 2010 SUBSTACK CONFIGURATION

CONTROL BIT SETTINGS							PARAMETER STACK CONFIGURATION																								
SVR				SUR			FATAL LIMIT							UNDERFLOW LIMIT							OVERFLOW LIMIT										
V7	V6	V5	V4	U2	U1	U0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	
X	X	X	0	0	0	X	P7	P6	P5	1	1	1	1	1	P7	P6	P5	0	U6	U5	U4	U3	P7	P6	P5	0	V3	V2	V1	V0	
X	X	X	1	0	0	0	P7	P6	P5	0	1	1	1	1	P7	P6	P5	1	U6	U5	U4	U3	P7	P6	P5	0	V3	V2	V1	V0	
X	X	X	1	0	0	1	P7	P6	P5	1	1	1	1	1	P7	P6	P5	0	U6	U5	U4	U3	P7	P6	P5	1	V3	V2	V1	V0	
X	X	0	X	0	1	X	P7	P6	1	1	1	1	1	1	P7	P6	0	U7	U6	U5	U4	U3	P7	P6	0	V4	V3	V2	V1	V0	
X	X	1	X	0	1	0	P7	P6	0	1	1	1	1	1	P7	P6	1	U7	U6	U5	U4	U3	P7	P6	0	V4	V3	V2	V1	V0	
X	X	1	X	0	1	1	P7	P6	1	1	1	1	1	1	P7	P6	1	U7	U6	U5	U4	U3	P7	P6	1	V4	V3	V2	V1	V0	
X	0	X	X	1	0	X	P7	1	1	1	1	1	1	1	P7	0	0	U7	U6	U5	U4	U3	P7	0	V5	V4	V3	V2	V1	V0	
X	1	X	X	1	0	0	P7	0	1	1	1	1	1	1	P7	1	0	U7	U6	U5	U4	U3	P7	0	V5	V4	V3	V2	V1	V0	
X	1	X	X	1	0	1	P7	1	1	1	1	1	1	1	P7	0	0	U7	U6	U5	U4	U3	P7	1	V5	V4	V3	V2	V1	V0	
0	X	X	X	1	1	X	1	1	1	1	1	1	1	1	0	0	0	U7	U6	U5	U4	U3	0	V6	V5	V4	V3	V2	V1	V0	
1	X	X	X	1	1	0	0	1	1	1	1	1	1	1	1	0	0	0	U7	U6	U5	U4	U3	0	V6	V5	V4	V3	V2	V1	V0
1	X	X	X	1	1	1	1	1	1	1	1	1	1	1	0	0	0	U7	U6	U5	U4	U3	1	V6	V5	V4	V3	V2	V1	V0	

CONTROL BIT SETTING							PARAMETER STACK CONFIGURATION																								
SVR				SUR			FATAL LIMIT							UNDERFLOW LIMIT							OVERFLOW LIMIT										
V15	V14	V13	V12	U10	U9	U8	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	
X	X	X	0	0	0	X	P15	P14	P13	1	1	1	1	1	P15	P14	P13	0	U14	U13	U12	U11	P15	P14	P13	0	V11	V10	V9	V8	
X	X	X	1	0	0	0	P15	P14	P13	0	1	1	1	1	P15	P14	P13	1	U14	U13	U12	U11	P15	P14	P13	0	V11	V10	V9	V8	
X	X	X	1	0	0	1	P15	P14	P13	1	1	1	1	1	P15	P14	P13	0	U14	U13	U12	U11	P15	P14	P13	1	V11	V10	V9	V8	
X	X	0	X	0	1	X	P15	P14	1	1	1	1	1	1	P15	P14	0	U15	U14	U13	U12	U11	P15	P14	0	V12	V11	V10	V9	V8	
X	X	1	X	0	1	0	P15	P14	0	1	1	1	1	1	P15	P14	1	U15	U14	U13	U12	U11	P15	P14	0	V12	V11	V10	V9	V8	
X	X	1	X	0	1	1	P15	P14	1	1	1	1	1	1	P15	P14	0	U15	U14	U13	U12	U11	P15	P14	1	V12	V11	V10	V9	V8	
X	0	X	X	1	0	X	P15	1	1	1	1	1	1	1	P15	0	0	U15	U14	U13	U12	U11	P15	0	V13	V12	V11	V10	V9	V8	
X	1	X	X	1	0	0	P15	0	1	1	1	1	1	1	P15	1	0	U15	U14	U13	U12	U11	P15	0	V13	V12	V11	V10	V9	V8	
X	1	X	X	1	0	1	P15	1	1	1	1	1	1	1	P15	0	0	U15	U14	U13	U12	U11	P15	1	V13	V12	V11	V10	V9	V8	
0	X	X	X	1	1	X	1	1	1	1	1	1	1	1	0	0	0	U15	U14	U13	U12	U11	0	V14	V13	V12	V11	V10	V9	V8	
1	X	X	X	1	1	0	0	1	1	1	1	1	1	1	1	0	0	0	U15	U14	U13	U12	U11	0	V14	V13	V12	V11	V10	V9	V8
1	X	X	X	1	1	1	1	1	1	1	1	1	1	1	0	0	0	U15	U14	U13	U12	U11	1	V14	V13	V12	V11	V10	V9	V8	

TABLE 6.3: RTX 2010 MULTITASKING SUBSTACKS

CONTROL BIT SETTINGS							PARAMETER STACK CONFIGURATION																
SVR				SUR			STACK SIZE WORDS	STACK RANGE															
								LOWEST ADDRESS							HIGHEST ADDRESS								
V7	V6	V5	V4	U2	U1	U0		7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
X	X	X	0	0	0	X	32	P7	P6	P5	0	0	0	0	0	P7	P6	P5	1	1	1	1	1
X	X	X	1	0	0	0	32	P7	P6	P5	0	0	0	0	0	P7	P6	P5	1	1	1	1	1
X	X	X	1	0	0	1	32	P7	P6	P5	0	0	0	0	0	P7	P6	P5	1	1	1	1	1
X	X	0	X	0	1	X	64	P7	P6	0	0	0	0	0	0	P7	P6	1	1	1	1	1	1
X	X	1	X	0	1	0	64	P7	P6	0	0	0	0	0	0	P7	P6	1	1	1	1	1	1
X	X	1	X	0	1	1	64	P7	P6	0	0	0	0	0	0	P7	P6	1	1	1	1	1	1
X	0	X	X	1	0	X	128	P7	0	0	0	0	0	0	0	P7	1	1	1	1	1	1	1
X	1	X	X	1	0	0	128	P7	0	0	0	0	0	0	0	P7	1	1	1	1	1	1	1
X	1	X	X	1	0	1	128	P7	0	0	0	0	0	0	0	P7	1	1	1	1	1	1	1
0	X	X	X	1	1	X	256	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
1	X	X	X	1	1	0	256	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
1	X	X	X	1	1	1	256	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
CONTROL BIT SETTINGS							RETURN STACK CONFIGURATION																
SVR				SUR			STACK SIZE WORDS	STACK RANGE															
								LOWEST ADDRESS							HIGHEST ADDRESS								
V15	V14	V13	V12	U10	U9	U8		7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
X	X	X	0	0	0	X	32	P15	P14	P13	0	0	0	0	0	P15	P14	P13	1	1	1	1	1
X	X	X	1	0	0	0	32	P15	P14	P13	0	0	0	0	0	P15	P14	P13	1	1	1	1	1
X	X	X	1	0	0	1	32	P15	P14	P13	0	0	0	0	0	P15	P14	P13	1	1	1	1	1
X	X	0	X	0	1	X	64	P15	P14	0	0	0	0	0	0	P15	P14	1	1	1	1	1	1
X	X	1	X	0	1	0	64	P15	P14	0	0	0	0	0	0	P15	P14	1	1	1	1	1	1
X	X	1	X	0	1	1	64	P15	P14	0	0	0	0	0	0	P15	P14	1	1	1	1	1	1
X	0	X	X	1	0	X	128	P15	0	0	0	0	0	0	0	P15	1	1	1	1	1	1	1
X	1	X	X	1	0	0	128	P15	0	0	0	0	0	0	0	P15	1	1	1	1	1	1	1
X	1	X	X	1	0	1	128	P15	0	0	0	0	0	0	0	P15	1	1	1	1	1	1	1
0	X	X	X	1	1	X	256	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
1	X	X	X	1	1	0	256	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
1	X	X	X	1	1	1	256	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1



6.1.4 Stack Error Conditions

Stack errors which may occur on the RTX 2000, RTX 2001A, and RTX 2010 Microcontrollers are overflow and underflow.

An **overflow** occurs when an attempt is made to push data onto a full stack. Since the stacks wrap around, the result is that existing data on the stack will be overwritten by the new data when an overflow occurs.

An **underflow** occurs when an attempt is made to pop data off an empty stack, causing invalid data to be read from the stack.

A buffer zone may be set up using the stack limits to cause a stack error interrupt to be generated prior to an actual overflow or underflow occurs.

6.1.4.1 RTX 2001A and RTX 2010 Fatal Stack Errors

In addition to the overflow and underflow stack errors, the RTX 2010 and RTX 2001A provide a fatal error flag.

A **Fatal Stack Error** occurs when an attempt is made to push data onto or to pop data off of the highest location of the substack. It does not generate an interrupt (since the normal stack limits can be used to generate the interrupt). The fatal errors for the stacks are logically OR'ed together to produce bit 0 of the **Interrupt Base Control Register**, and they are cleared whenever **SPR** is written to. The implication of a fatal error is that data on the stack may have been corrupted or that invalid data may have been read from the stack.

6.2 Interrupt Controller

The RTX 2000 Series Interrupt Controller prioritizes 13 interrupt requests, masks undesired interrupts, signals the processor core when a valid interrupt has occurred, and provides a vector to an interrupt handler to service the interrupt.

Inputs to the Interrupt Controller come from both internal and external sources. Internal sources are the stack overflow and underflow signals, the three counter/timers, and the Software Interrupt signal. External sources are the Non-Maskable Interrupt (NMI) input, and the External Interrupt pins EI1-EI5. EI pins 3, 4, and 5 may be shared with the Counter/Timers for external event counting. See Section 6.4 for details.

Except for NMI, the interrupt inputs may be individually enabled or disabled through the Interrupt Mask Register (IMR) at ASIC address 08H. Each bit of the IMR corresponds to one interrupt level; Table 6.2 shows the bit associated with each level. Setting a bit to 1 disables the corresponding level. Note that the Interrupt Disable bit in the Configuration Register must be 0 for any maskable interrupts to be recognized by the core. The NMI input may not be disabled through the IMR.

When the RTX receives an interrupt request, it saves the current contents of the Program Counter and Code Page registers in the IPR and I registers, which form the logical top element of the Return Stack, then initiates an Interrupt Acknowledge (INTA) cycle. During the INTA cycle, the Interrupt Controller generates a vector to the appropriate interrupt service routine. The RTX sets the Code Page Register to 0, then reads the vector from the Interrupt Controller to determine the address of the first instruction to execute for the interrupt service routine.

The vector provided by the Interrupt Controller consists of three parts:

- Bits 10-15 Come from bits 10-15 of the Interrupt Base/Control Register (IBC).
- Bits 5-9 Come from the interrupt vector and depend on the interrupt level; see Table 6.2.
- Bits 0-4 Are always 0.

The interrupt vector points into a 544 byte table located in Code Page 0 of the RTX memory. Each interrupt service routine is allocated 32 bytes in this table. If the service routine will not fit in 32 bytes, it may make calls to any address in the RTX's memory space. The interrupt service routine must include a Return-From-Subroutine instruction.

The interrupt service Table must be located on a 1024-byte address boundary; that is, address bits 0-9 must be 0. The IBC register should be initialized with the upper 6 bits of the address of the table. For example, if the table is located at location 1000H, IBC bits 15-10 should be set to 00010 binary. Table 6.2 shows the interrupt service routine address associated with each interrupt level.

The Interrupt Controller samples the interrupt request inputs during each instruction at the rising edge of PCLK (except when executing in streamed mode). If one or more inputs are active, the Interrupt Controller generates the vector corresponding to the input with the highest priority, and signals the core processor that an interrupt request is present. The core then initiates an INTA cycle. For the timer interrupts, which are edge triggered interrupts, the INTA cycle from the processor clears the highest priority timer interrupt and allows the Interrupt Controller to process lower priority interrupts.

The Interrupt Vector Register, IVR, which is a read-only register at ASIC address 0BH, contains the current vector being generated by the Interrupt Controller. If no interrupt request is present, bits 5-9 of the register will contain 10000 binary.

The IVR vector may be polled for interrupt request information. Note that a particular request level must be unmasked in order for the interrupt controller to generate a vector for it.

Conditions may occur in which an interrupt request goes active and then inactive prior to the INTA cycle. An example would be a stack operation that overflows the stack and a subsequent stack operation that corrects the condition. If the interrupt is active long enough, an INTA cycle will be initiated. This results in the generation of a "No Interrupt" vector as a valid address and program execution will transfer to the location indicated. Programmers should install a service routine for "No Interrupt" to account for this situation.

Example:

The interrupt vector table is located at 2000H. The IBC register should be loaded with the binary value 00100xxxxxxxxx where xxxxxxxx depends on system configuration. The Interrupt Controller would generate the following vectors:

No interrupt	-	2200H
NMI	-	21E0H
EI1 pin	-	21C0H
Timer 0	-	2100H
SWI	-	2040H

TABLE 6.2: INTERRUPT CONTROLLER

Priority	Source	IMR Bit	Type	Vector Address (binary)
0	Non-Maskable Interrupt NMI	none	Edge	vvv vv01 1110 0000
1	EI1 pin	1	Level	vvv vv01 1100 0000
2	Parameter Stack underflow	2	Level	vvv vv01 1010 0000
3	Return Stack underflow	3	Level	vvv vv01 1000 0000
4	Parameter Stack overflow	4	Level	vvv vv01 0110 0000
5	Return Stack overflow	5	Level	vvv vv01 0100 0000
6	EI2 pin	6	Level	vvv vv01 0010 0000
7	Timer 0	7	Edge	vvv vv01 0000 0000
8	Timer 1	8	Edge	vvv vv00 1110 0000
9	Timer 2	9	Edge	vvv vv00 1100 0000
10	EI3 pin	10	Level	vvv vv00 1010 0000
11	EI4 pin	11	Level	vvv vv00 1000 0000
12	EI5 pin	12	Level	vvv vv00 0110 0000
13	Software Interrupt	13	Level	vvv vv00 0100 0000
N/A	No Interrupt	N/A	N/A	vvv vv10 0000 0000

Where:

vvvvvv = bits 10-15 from IBC register

6.2.1 Interrupt Acknowledgement

If interrupts are enabled when the processor receives the Interrupt Controller's signal, it enters an Interrupt Acknowledge (INTA) cycle. During this cycle, the processor saves the current execution address on the Return Stack, disables interrupts as described in Section 3.5.2, then reads a vector from the Interrupt Controller which points to the address of a service routine to handle the particular interrupt. Section 6.2 describes the Interrupt Controller interface in more detail.

The INTA cycle sets the least significant bit of the return address saved on the Return Stack to a 1, to indicate that the subroutine (Interrupt Service Routine) was called as a result of an interrupt.

When the service routine executes a Return-From-Subroutine instruction to resume execution from the point where the processor was interrupted, the set LSB of the Return Stack causes interrupts to be enabled automatically. The Interrupt Service Routine can also re-enable interrupts, but is then subject to being interrupted by another interrupt.

6.2.2 Disabling Interrupts

The processor can enable or disable all maskable interrupts at any time by controlling the state of the Set Interrupt Disable bit in the Configuration Register (CR bit 4). Setting this bit to 1 disables interrupts; this is the state of the bit when the RTX is reset. The processor will not recognize interrupts until the bit is reset to 0 by writing to the CR register.

The CR register contains two bit positions associated with the Interrupt Disable bit.

The **Set Interrupt Disable (SID)** bit is a write-only bit which is used to set or reset the bit under program control; this bit will always read as 0 no matter what the bit is set to. This provides a convenient mechanism for quickly enabling interrupts, whereby the CR register is read onto the Parameter Stack (reading the SID bit as a 0), then immediately rewritten, effectively clearing the SID bit to 0 (enabled). This process requires only two clock cycles, eliminating the extra time it would take to read the register, mask the bit, then rewrite the register.

The **Interrupt Disable Status bit (CR bit 14)** is a read-only bit which contains the true state of the Interrupt Disable bit.

CR bit 15 indicates the status of the core interrupt request input from the interrupt controller. This bit may be polled to determine interrupt status when core interrupts are disabled.

6.2.3 Software Interrupt

The RTX has a single level Software Interrupt capability. A special form of one of the RTX I/O write instructions sets a flip-flop attached to one input of the Interrupt Controller. If the interrupt level associated with the Software Interrupt is unmasked (see Section 6.2), this input causes the Interrupt Controller to generate a vector pointing to the service routine corresponding to the Software Interrupt.

A separate I/O instruction clears the Software Interrupt Request flip-flop. The service routine for the Software Interrupt must execute this instruction before re-enabling interrupts or returning to normal program execution to prevent another SWI cycle from being executed. This interrupt request input is level-sensitive and will continue to generate interrupts until the flip-flop is reset. See "Predefined ASIC Instructions" in Chapter 7 for the machine instructions which set and clear the flip-flop.

Due to internal delays in generating the interrupt request, and the fact that the Software Interrupt is assigned to the lowest priority level, the interrupt will not be serviced for two instructions following execution of the Software Interrupt instruction. This means that the instructions immediately following the Software Interrupt should not assume that the interrupt has been serviced.

Inserting two nops between the Software Interrupt Instruction and the instruction which follows it will guarantee that the software interrupt will be serviced before the following instruction is executed.

6.3 On-Chip Hardware Math Support

For math intensive applications, the RTX 2000 Microcontroller is provided with a 16-bit on-chip hardware multiplier.

The RTX 2010 is provided with a 16-bit on-chip hardware Multiplier/Accumulator, 32-bit Barrel Shifter, Leading Zero Detector, and hardware Floating Point support.

The RTX 2001A does not have these features.

6.3.1 RTX 2000 Multiplier Operation

The hardware multiplier on the RTX 2000 multiplies two 16-bit numbers, yielding a 32-bit product, in one clock cycle. The multiplier can treat the input operands as either signed (two's complement) or unsigned integers, and can optionally round the result to 16 bits.

The multiplier's input operands come from the TOP and NEXT registers. The multiplication function is activated by a special form of the ASIC Bus write instructions to the Multiplier High (MHR) or Multiplier Low Register (MLR) address.

The form of the instruction used determines whether the operands will be treated as signed or unsigned values. See Section 7.7.1 for the exact instruction coding. Note that the multiply instructions do not pop the Parameter Stack; the contents of TOP and NEXT remain intact.

The product is stored in the Multiplier High and Multiplier Low Registers. The Multiplier High Register contains the upper 16 bits of the product, while the Multiplier Low Register contains the lower 16 bits.

The registers may be read in either order, and there is no requirement that both registers be read. Reading either register moves its value into the TOP register, and pushes the original value in TOP into NEXT. The original value of NEXT is lost; it is not

pushed onto the Parameter Stack. This permits overwriting the original operands left in TOP and NEXT, which were not popped by the multiply operation. See Figure 6.8.

If 32-bit precision is not required, the multiplier output may be rounded to 16 bits. This is accomplished by setting the ROUND bit in the Interrupt Base/Control Register to 1. The ROUND bit functions independently of signed or unsigned mode.

If the ROUND bit is set to one, all operations that use the multiplier automatically round the lower 16 bits of the result into the upper 16 bits. The rounding is achieved by adding 8000H to the least significant 16 bits (during the same cycle as the multiply). Thus, if the ROUND bit is set, after a multiply the result will be as follows:

- If the most significant bit of the MLR is set (=1), the MHR is incremented and the MSB of MLR will be 0.
- If the most significant bit of the MLR is not set (=0), the MLR is left unchanged, and the MSB of the MLR will be 1.

The multiply instructions disable interrupts during the multiplication cycle, and for the next two clock cycles. Reading either result register also disables interrupts during the read, and for the next clock cycle. This allows a multiplication operation to be performed, and both the upper and lower registers to be read sequentially, with no danger of an interrupt service routine corrupting the contents of the registers between reads.

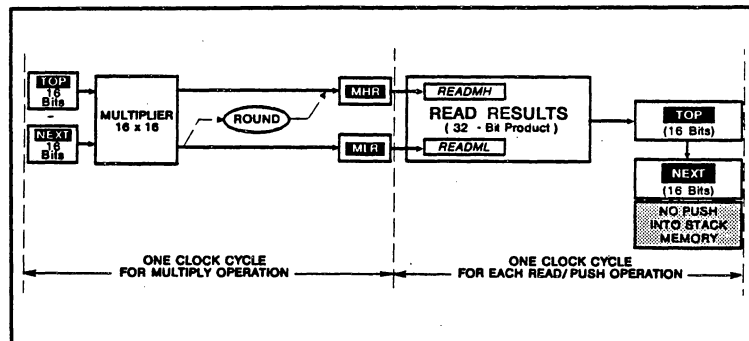


FIGURE 6.8: RTX 2000 MULTIPLIER OPERATION

Example 1: A typical multiplication sequence without rounding

- 1. Set ROUND bit to 0 (if not already set)
- 2. Load multiplier and multiplicand into TOP and NEXT
- 3. Execute appropriate signed or unsigned "multiply" instruction (interrupts are disabled)
- 4. Read lower result register (interrupts are disabled)
- 5. Read upper result register (interrupts are disabled)

The 32-bit product is now on the Parameter Stack, the most significant 16 bits are in TOP, the least significant 16 bits are in NEXT, and interrupts are enabled.

Example 2: A typical multiplication sequence with rounding

- 1. Set ROUND bit to 1 (if not already set)
- 2. Load multiplier and multiplicand into TOP and NEXT
- 3. Execute appropriate signed or unsigned "multiply" instruction (interrupts are disabled)
- 4. Read upper result register, MHR (interrupts are disabled)
- 5. Exchange TOP and NEXT registers (interrupts are disabled)
- 6. Discard top stack item (interrupts are enabled)

The 16-bit product is now in the TOP Register and interrupts are enabled.

6.3.2 RTX 2010 Hardware Math Support

In addition to an on-chip multiplier, the RTX 2010 provides additional hardware on-chip to support Multiply-Accumulate operations, 32-bit shift operations, and Leading Zero Detection.

6.3.2.1 RTX 2010 Multiplier/Accumulator Operation

The Hardware Multiplier/Accumulator (MAC) on the RTX 2010 functions as both a Multiplier, and as a Multiplier-Accumulator.

When used as a Multiplier alone, it multiplies two 16-bit numbers, yielding a 32-bit product in one clock cycle.

When used as a Multiplier-Accumulator, it multiplies two 16-bit numbers, yielding an intermediate 32-bit product, which is then added to the 48-bit Accumulator. This entire process takes place in a single clock cycle.

The MAC's input operands come from three possible sources (see Figure 17):

- The TOP and NEXT Registers
- The Parameter Stack and memory
- The ASIC Bus and memory

These inputs can be treated as either signed (two's complement) or unsigned integers, depending on the form of the instructions used. In addition, if the ROUND option is selected, the Multiplier can round the result to 16 bits. Note that the MAC instructions do not pop the Parameter Stack; the contents of TOP and NEXT remain intact.

For the Multiplier, the product is read from the Multiplier High Product Register, MHR, which contains the upper 16 bits of the product, and the Multiplier Low Product Register, MLR, which contains the lower 16 bits.

For the Multiplier-Accumulator, the accumulated product is read from the **Multiplier Extension Register, MXR**, which contains the upper 16 bits, the **MHR**, which contains the middle 16 bits, and the **MLR**, which contains the low 16 bits.

The registers may be read in any order, and there is no requirement that all registers be read. Reading from any of the three registers moves its value into **TOP**, and pushes the original value in **TOP** into **NEXT**.

If the read is from **MHR** or **MLR**, the original value of **NEXT** is lost, i.e. it is not pushed onto stack memory. This permits overwriting the original operands left in **TOP** and **NEXT**, which are not popped by the **MAC** operations.

If the read is from **MXR**, the original value of **NEXT** is pushed onto the stack.

In addition to this, any of the three **MAC** registers can be directly loaded from **TOP**. This pops **NEXT** into **TOP** and the Parameter Stack into **NEXT**.

If 32-bit precision is not required, the multiplier output may be rounded to 16 bits. The **RTX 2010 ROUND** mode functions exactly like the **RTX 2000 ROUND** mode. See Section 6.3.2.1 for details.

The multiply instructions suppress interrupts during the multiplication cycle. Reading **MHR** or **MLR** also suppresses interrupts during the read. This allows a multiplication operation to be performed, and both the upper and lower registers to be read sequentially, with no danger of a non-NMI interrupt service routine corrupting the contents of the registers between reads (for compatibility with the **RTX 2000**). The **Multiply-Accumulate** instructions do not suppress interrupts during instruction execution.

6.4 Counter/Timers

The RTX 2000 Family of microcontrollers contains three identical 16-bit Counter/Timers. Each counter may be configured as either an external event counter, in which case its clock input comes from an RTX input pin, or as a timer, in which its clock input comes from the processor's internal TCLK signal. Each Counter/Timer circuit consists of a pre-load register, a 16-bit down-counter, clock selection circuitry, and an interrupt output. See Figure 6.10.

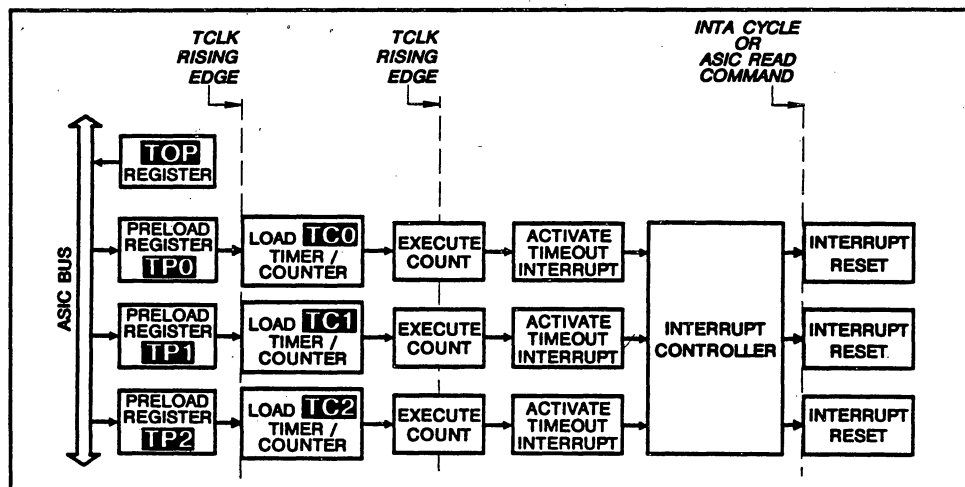


FIGURE 6.10: TIMER/COUNTER OPERATION

6.4.1 Counter/Timer Operation

Writing to a counter's ASIC Bus address loads a 16-bit value into its pre-load register. This value is loaded into the counter on the counter's next input clock cycle. Each subsequent input clock cycle decrements the counter by 1. The counters are free-running in that they do not stop when they reach 0, but rather reload from the pre-load register and continue counting. Loading a counter with 0 is equivalent to loading it with 65536.

The counters clock synchronously with the processor's internal TCLK signal. This prevents the clocking from occurring during an I/O read, and means that the contents of each counter may be read at any time without disturbing the count or interfering with the counting process. This also means that the processor clock must be running for counting (from either an internal or external clock) to take place, and that the maximum counting rate with an external clock source cannot exceed one-half the processor's clock rate.

When a counter is written, the value is not loaded until one TCLK or EI pulse later, depending on which is the source to the counter.

6.4.2 Counter/Timer Interrupts

Each counter generates an interrupt signal when it reaches 0. These signals are routed through the Interrupt Controller, and may be masked by setting the appropriate bits in the Interrupt Mask Register.

The counter interrupts are reset during the corresponding Interrupt Acknowledge cycle. This means that it is possible that there will be an interrupt request present when the interrupt levels associated with each counter are unmasked, especially if the counters have been running for some time before being loaded with a count value.

6.4.3 Clock Selection

The clock selection circuit determines the source for each counter input clock. Each counter may be clocked from either the processor's internal TCLK signal, or from one of the processor's External Input (EI) pins. The 3 EI pins EI3, EI4, and EI5 are shared with the interrupt controller. Each pin may either be an input to the Interrupt Controller, or a clock input to a counter. Bits 8 and 9 in the Interrupt Base/Control Register determine the usage of each pin. See Table 6.3.

TABLE 6.3: TIMER/COUNTER EI PIN ASSIGNMENTS

IBC bit 9	IBC bit 8	EI3	EI4	EI5
0	0	INT10	INT11	INT12
0	1	CLK0	INT11	INT12
1	0	CLK0	CLK1	INT12
1	1	CLK0	CLK1	CLK2

Notes:

INTn - input to Interrupt Controller, level n

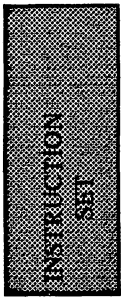
CLKn - clock input to Counter/Timer n

If a counter input is not assigned to an EI pin, it is decremented by the processor's TCLK signal.

If an Interrupt Controller input is not assigned to an EI pin, it is held inactive.

CHAPTER 7

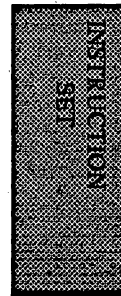
INSTRUCTION SET



7 Instruction Set

This section describes each of the instruction operation codes ("opcodes") available on the RTX processor. Since Forth is the "assembly language" for the processor, the instruction set is best described in terms of Forth primitives; Appendix A presents the opcodes in Forth format. This chapter presents each of the instructions in terms of their stack and register effects.

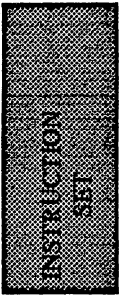
one possible



7.1 General Information

Instructions are always aligned on word boundaries, with the most significant byte of the instruction at the even address, and the least significant byte at the next higher odd address. All instructions are 16 bits long, with the exception of long literals which require 16 bits for the instruction and 16 bits for the literal value.

All RTX instructions execute in either one or two clock cycles. All instructions which do not perform memory accesses execute in a single clock cycle. Instructions which perform memory accesses or load long literal data require two clock cycles. This consistency of execution time makes it possible to write code with very predictable behavior.



7.1.1 Streamed Execution Mode

The RTX processor has a "streamed" instruction feature, in which an instruction is made to repeat a specified number of times by writing a count value into the Index Register. This feature is useful for doing fast data transfers, loops and some math functions.

The count is written into the Index Register using an ASIC Bus write instruction to the Index Register at ASIC Address 02H. See Section 4.3.1.3 for details. The value written must be 1 less than the desired number of repetitions.

Only the first cycle of a two cycle instruction is repeated. The second cycle is performed only once, after the first cycle has been repeated the desired number of times.

Interrupts are disabled during streamed instruction execution. Only a Non-maskable interrupt (NMI) will interrupt streamed execution.

The RTX 2010 provides the ability to set the NMI_MODE Flag (bit 11 of the CR Register). If this bit is set, (MODE1), then the NMI is suppressed until the streamed instruction has been completed.

7.1.2 The Auto-decrementing Loop Instruction

The RTX provides a fast auto-decrementing loop instruction called NEXT. The NEXT instruction branches based on a count previously pushed onto the Return Stack (in I).

The NEXT branch instruction tests the contents of the I Register at the end of each loop. If the contents are not 0, the I Register is decremented, and a branch (typically to the beginning of the loop) is executed; if the I Register contains 0, the Return Stack is popped, and execution continues with the instruction following the conditional branch instruction.

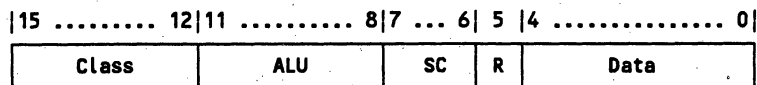
How does it work?
→ INDEX reg counts down...
what if you write 0?
more details!

is count popped at end of stream?

Why is this here?
It seems out of place!

7.2 Format

All processor instructions are 16 bits, with the following general fields:



Class	General type of instruction:
8,9	Branches and Loops
10	Math/Logic Functions
11	Register and Short Literal Operations
12	User Memory Access
13	Long Literals
14	Memory Access By Word
15	Memory Access By Byte

Each class is discussed separately.

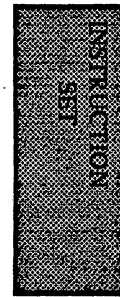
- ALU - ALU function to be performed.
- SC - Subclass. Function depends on Class field.
- R - Return bit. When set, causes a Return-From-Subroutine.
- Data - Depending on Class, indicates shift operation, short literal data, ASIC Bus address, or memory address.

Opcode descriptions use the format which follows.

The right-hand item in each list is the top stack element. "T" and "N" are used to represent the contents of TOP and NEXT before the instruction is executed. For example,

N T -- *T

shows that the instruction starts with values in TOP and NEXT, and ends with the contents of NEXT being discarded and the contents of TOP being inverted.



conditionally

When reading the "Processor operations" descriptions, it is important to keep in mind that the RTX performs the indicated operations in parallel when executing an instruction. Thus, the original contents of a register may be used as an operand for an instruction even though the register is loaded with a new value during execution of the instruction.

For example, the contents of TOP and NEXT may be used as operands for a math operation which replaces the contents of TOP with the results of the operation and pops the Parameter Stack into NEXT.

In the descriptions of Processor operations for two cycle instructions, the values shown for "T" and "N" during the second cycle of the instruction represent the values loaded into TOP and NEXT during the first cycle of the instruction, not the contents of TOP and NEXT before the instruction was executed.

For example,

Processor operations:

1st cycle:	N => T	m => N
2nd cycle:	N => T	T => N

Parameter Stack effect:

N T -- N m

should be interpreted as described on the following page.

During the first cycle, the contents of NEXT are written to TOP, overwriting the contents of TOP. At the same time, the contents (m) of the memory location addressed by the original contents of TOP are loaded into NEXT.

During the second cycle, the new contents of NEXT (the memory data) are written into TOP, while the new contents of TOP (the original contents of NEXT) are written back into NEXT. The net effect of this operation is to replace the contents of TOP with the contents of the memory location addressed by the contents of TOP.

7.3 Subroutine Call

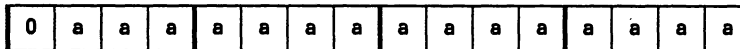
Any instruction which has bit 15 set to 0 will perform a Subroutine Call. The contents of the Code Page Register and the address of the instruction following the Call are pushed onto the Return Stack. The Program Counter Register is then loaded with the address contained in the instruction.

The address bits in the instruction represent the word address to be executed. The actual address may be calculated by shifting the value left by 1 bit, and inserting a 0 in the least significant bit. For example, an instruction code of 2A45H would cause a call to location 548AH:

```

                2A45: 0010 1010 0100 0101
shift and insert 0: 0101 0100 1000 1010 = 548AH
    
```

If a Subroutine Call is to be made to a Code page other than the one containing the Call instruction, the instruction immediately preceding the Call must load the correct page number into the Code Page Register.



- Description:** Subroutine-Call.
- Number of cycles:** 1
- Processor operations:**
- IPR, I => Rstack Save return address on Return Stack
 - PC => I
 - CPR => IPR
 - aaaaaaaaaaaaa0 => PC Load Call address into Program Counter

Parameter Stack effect:

no change

7.4 Subroutine Return

Any non-call/branch instruction which has the Subroutine Return bit (bit 5) set will cause a Return-From-Subroutine operation. The Return Stack is popped into the Program Counter Register and Code Page Register, causing execution to resume with the instruction following the call to the current subroutine. The Subroutine Return bit is shown in the opcode formats as "R".



Description: Return-From-Subroutine.

Number of cycles: 1 if coded as a separate instruction; 0 if coded as part of the last instruction in a subroutine

Processor operations:

I => PC IPR => CPR Rstack => I, IPR

Parameter Stack effect:

no change

The Subroutine Return bit may not be used in the following circumstances:

- A Branch or Call instruction. All bits of the instruction are significant.
- Any instruction which pops the Return Stack. "Return Stack pop" instructions which have the Return bit set behave as non-popping "Index Register Read" instructions.

In these situations, a stand-alone return instruction must be added as the last instruction of the subroutine. This would typically be a No Operation (NOP) instruction with the Return bit set.

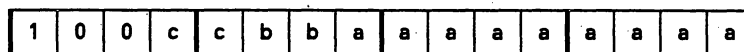
7.5 Classes 8 and 9: Branches and Loops

These instructions cause either a conditional or unconditional branch. The RTX Branch Instruction treats each Code Memory page as 64 "blocks" of 512 words each. Bits 15-10 of the Program Counter determine the block number; bits 9-0 determine the word offset within the block.

In order to perform branches in a single clock cycle, RTX branch instructions encode the branch address within the instruction.

The limited number of bits available for encoding the address requires that all branch destinations must be within the same, next, previous, or first memory block. Except for the "Branch to block 0" instruction, the longest branch which the processor can perform is $\pm 1K$ words.

RTX branch instructions have the following general form:



- "cc" - Determine conditions for branching. See Table 7.1.
- "bbaaaaaaaaa" - Branch address.
- "bb" - Block Select. Determines new value of bits 15-10 of Program Counter. See Table 7.2.
- "aaaaaaaa" - Replaces bits 9-1 of Program Counter (word offset from address 0 in the new block).

Bit 0 of the Program Counter Register is set to 0 (word aligned instructions). The resulting branch address is designated "ADR" in the instruction descriptions.

TABLE 7.1: BRANCH CONDITIONS

cc	Branch conditions
00	Branch if contents of TOP = 0. Don't pop stack.
01	Branch if contents of TOP = 0. Pop stack.
10	Unconditional branch
11	If contents of Index Register \neq 0, branch and decrement I

TABLE 7.2: BLOCK BRANCHING ASSIGNMENTS

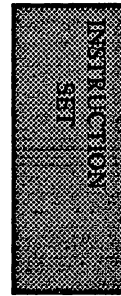
bb	result
00	Branch within same memory block (no change to bits 15-10)
01	Branch to next memory block (add 1 to value represented by bits 15-10)
10	Branch to Block 0 (set bits 15-10 to 0)
11	Branch to previous block (add -1 to value represented by bits 15-10)

The most important thing to note when calculating the address field for a branch instruction is that, when the branch instruction is executed, the Program Counter will already be pointing to the instruction following the branch instruction. The "bb" field will be applied to this address, not the address of the branch instruction.

This is only important when the branch instruction is the last instruction in a 512 word block. In this case, the Program Counter is already pointing to the first word in the next block, and the "bb" field must be calculated based on that block number, not the block containing the branch instruction.

Example: A branch instruction is located at address 07FEH, the last instruction in block #1 (bits 15-10 = 000001). When this instruction executes, the Program Counter is pointing to the instruction at 0800H, the first instruction in block #2. To perform a branch to a location in block #2, the "bb" field must be set to 00 (branch to same block) rather than 01 (branch to next block).

Branch Address Examples: (bbaaaaaaaaa = bits 0 - 9 of opcode)



<p>Example 1: Branch to same block</p> <p>address of branch instruction: 0001 0100 1010 0100 address to branch to: 0001 0100 1111 0000 bbaaaaaaaaa : 1001 000 0011110000 resulting address:</p> <p>PC-Register bits 15-10 => 0001 01 PC-Register bits 9-1 => 00 1111 000 PC-Register bit 0 => 0</p> <p>----- Final branch address => 0001 0100 1111 0000</p>	
<p>Example 2: Branch to next block</p> <p>address of branch instruction: 0001 0100 1010 0100 address to branch to: 0001 1000 0101 1110 bbaaaaaaaaa : 1001 001 0001011111 resulting address:</p> <p>PC-Register bits 15-10 => 0001 01 + 1 ----- => 0001 10</p> <p>PC-Register bits 9-1 => 00 0101 111 PC-Register bit 0 => 0</p> <p>----- Final branch address => 0001 1000 0101 1110</p>	
<p>Example 3: Branch to block 0</p> <p>address of branch instruction: 0001 0100 1010 0100 address to branch to: 0000 0000 1100 1010 bbaaaaaaaaa : 1001 010 001100101 resulting address:</p> <p>PC-Register bits 15-10 => 0000 00 PC-Register bits 9-1 => 00 1100 101 PC-Register bit 0 => 0</p> <p>----- Final branch address => 0000 0000 1100 1010</p>	
<p>Example 4: Branch to previous block</p> <p>address of branch instruction: 0001 0100 1010 0100 address to branch to: 0001 0000 1101 1110 bbaaaaaaaaa : 1001 011 0011011111 resulting address:</p> <p>PC-Register bits 15-10 => 0001 01 - 1 ----- => 0001 00</p> <p>PC-Register bits 9-1 => 00 1101 111 PC-Register bit 0 => 0</p> <p>----- Final branch address => 0001 0000 1101 1110</p>	

BRANCH**Unconditional Branch****Unconditional Branch**

1	0	0	1	0	b	b	a	a	a	a	a	a	a	a	a
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Description: Branch to address indicated by bbaaaaaaaaa.

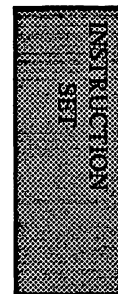
Number of cycles: 1

Processor operations:

ADR => PC

Parameter Stack effect:

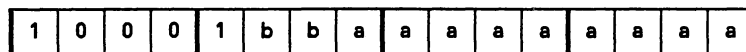
no change



BRANCH

Branch if T=0, Pop stack

Branch if T=0, Pop stack



Description:

If T = 0 Performs branch. Pops the Parameter Stack.

If T ≠ 0 Pops the Parameter Stack.

Number of cycles: 1

Processor operations:

If T = 0 N => T Pstack => N ADR => PC

If T ≠ 0 N => T Pstack => N

Parameter Stack effect:

If T = 0 N T -- N

If T ≠ 0 N T -- N

BRANCH**Branch if T=0, don't pop stack****Branch if T=0, don't pop stack****Description:**

If T = 0 Performs branch. Pops the Parameter Stack.

If T ≠ 0 No effect.

Number of cycles: 1

Processor operations:

If T = 0 N ⇒ T Pstack ⇒ N ADR ⇒ PC

If T ≠ 0 no operation

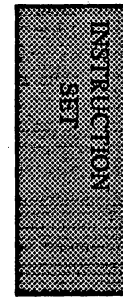
Parameter Stack effect:

If T = 0 N T -- N

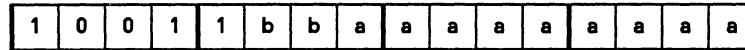
If T ≠ 0 N T -- N T

BRANCH

NEXT



Branch if I \neq 0



Description: This branch instruction is referred to as the "NEXT" instruction, and is useful for implementing a fast auto-decrementing loop.

If I \neq 0 Branch and decrement the Index Register (I), if I is not equal to 0.

If I = 0 If I contains 0, the Return Stack is popped. Execution continues with the next sequential instruction.

Number of cycles: 1

Processor operations:

If I \neq 0 I - 1 \Rightarrow I ADR \Rightarrow PC

If I = 0 Rstack \Rightarrow I

Parameter Stack effect:

no change

7.6 Class 10: ALU Operations

This class of instructions allows the processor to perform arithmetic and logic operations between the contents of the TOP and NEXT registers. These operations fall into two general categories: Single step and Multi-step. Multi-step Math operations are discussed in Chapter 8. The single step operations category covers those functions which may be completed in one clock cycle:

Addition	Stack manipulations
Subtraction	Boolean logic operations
1-bit shifting (*2 and /2)	

All ALU operations are performed between the contents of the TOP Register and another operand which is determined by the instruction. The results of the operation are loaded into TOP. The ALU function to be performed is encoded as a field in the instruction and is shown in the opcode formats as either "cccc" or "aaa".

Table 7.3 lists the ALU functions the RTX can perform. "T" indicates the contents of the TOP Register. "Y" indicates the source for the second ALU input. For single step math functions, Y is always the NEXT Register. For other classes of instructions, the source for Y will vary, depending on the instruction.

The "Resulting Carry" column indicates the new value which will be latched into the processor's Carry bit as a result of the operation.

TABLE 7.3: RTX ALU FUNCTIONS

cccc	aaa	function	Resulting Carry
0010	001	T AND Y	no change
0011		T NOR Y	no change
0100	010	T - Y	ALU carry
0101		T - Y with borrow	ALU carry
0110	011	T OR Y	no change
0111		T NAND Y	no change
1000	100	T + Y	ALU carry
1001		T + Y with carry	ALU carry
1010	101	T XOR Y	no change
1011		T XNOR Y	no change
1100	110	Y - T	ALU carry
1101		Y - T with borrow	ALU carry

Y - T

T - Y

7.6.1 Carry Bit

The Carry-out signal from the ALU is bit 0 (CY) of the Configuration Register, CR, and may be used for performing multi-precision addition and subtraction operations. The Configuration Register bit may be directly set or read under program control.

All addition and subtraction operations set the carry bit, but only the "add with carry" (cccc = 1101, see Table 7.3), "subtract with borrow" (cccc = 1101) and "swapped subtract with borrow" (cccc = 0101) use the value of the carry bit during calculations. None of the Boolean logic functions use or affect the carry.

Addition operations add the two ALU inputs, then optionally add the Carry-in bit (CY) to the least significant bit (LSB) of the sum. The Carry-out bit of the ALU becomes the new value for CY; 1 indicates an overflow out of the most significant bit (MSB).

Subtraction operations add the minuend (A in the examples below) to the 1's complement of the subtrahend (B in the examples), then optionally add the Carry-in (borrow) bit to the LSB of the sum. The Carry-out of the ALU indicates the borrow status; CY = 0 means that the result of the subtraction was negative and that a borrow should be performed from the next most significant stage of the subtraction.

TABLE 7.4: Examples: Cout = ALU Carry-out

A	B	Carry-in	without carry/borrow				with carry/borrow			
			A+B Cout		A-B Cout		A+B Cout		A-B Cout	
0	0	0	0	0	0	1	0	0	-1	0
0	0	1	0	0	0	1	1	0	0	1
0	1	0	1	0	-1	0	1	0	-2	0
0	1	1	1	0	-1	0	2	0	-1	0
1	0	0	1	0	1	1	1	0	0	1
1	0	1	1	0	1	1	2	0	1	1
1	1	0	2	0	0	1	2	0	-1	0
1	1	1	2	0	0	1	3	0	0	1
-1	0	0	-1	0	-1	1	-1	0	-2	1
-1	0	1	-1	0	-1	1	0	1	-1	1
-1	1	0	0	1	-2	1	0	1	-3	1
-1	1	1	0	1	-2	1	1	1	-2	1

7.6.2 Shift Operations

The single step math/logic functions allow the output of the ALU to be shifted as a 16-bit quantity, or the output of the ALU and the contents of the NEXT Register to be shifted as a 32-bit quantity in either direction before being loaded into the TOP (and NEXT) registers.

The shift function is embedded in the instruction, and is shown in the opcode formats as "ssss". Each of the shift functions is described in Tables 7.5 and 7.6, which use the following notations:

Zn	Bit n of ALU output (15 - 0)
TNn	Bit n of NEXT Register before shift (15 - 0)
CY	Old value of Carry bit as a result of ALU operation
C	New value of Carry bit as a result of the shift operation
T15,Tn,T0	MSB, typical bit, and LSB of TOP Register after the shift operation
N15,Nn,N0	MSB, typical bit, and LSB of NEXT Register after the shift operation

The first 8 shift functions affect only the TOP Register. The remaining shift functions affect either just the NEXT Register, or the TOP and NEXT registers combined as a 32-bit quantity.

In the 32-bit form, the TOP Register represents the most significant word of the 32-bit quantity and the NEXT Register the least significant.

TABLE 7.5: 16-BIT SHIFT FUNCTIONS

Shift ssss	name	effect	Status of C	TOP Register			NEXT Register		
				T15	Tn	T0	N15	Nn	N0
0000		no shift operation is performed	CY	Z15	Zn	Z0	TN15	TNn	TN0
0001	0<	Sign extend: The sign bit (bit 15) of TOP is propagated to all bit positions in TOP.	CY	Z15	Z15	Z15	TN15	TNn	TN0
0010	2*	Left Shift: TOP is shifted left by 1 bit, with 0 shifted into the LSB. MSB is shifted into the carry bit.	Z15	Z14	Zn-1	0	TN15	TNn	TN0
0011	2*c	Rotate Left: TOP is shifted left by 1 bit, with the carry bit shifted into the LSB. MSB is shifted into the carry bit.	Z15	Z14	Zn-1	CY	TN15	TNn	TN0
0100	cU2/	Right Shift Out of Carry: TOP is shifted right by 1 bit, with the carry bit shifted into the MSB. The LSB is discarded and 0 is shifted into the carry bit.	0	CY	Zn+1	Z1	TN15	TNn	TN0
0101	c2/	Rotate Right Through Carry: TOP is shifted right by 1 bit, with the carry bit shifted into the MSB. The LSB is shifted into the carry bit.	Z0	CY	Zn+1	Z1	TN15	TNn	TN0
0110	U2/	Logical Right Shift: TOP is shifted right by 1 bit, with 0 shifted into the MSB and carry bits. The LSB is discarded.	0	0	Zn+1	Z1	TN15	TNn	TN0
0111	2/	Arithmetic Right Shift: Bits 14-1 of TOP are shifted right by 1 bit. Bit 15 remains unchanged and is shifted into the carry bit and bit 14. The LSB is discarded.	Z15	Z15	Zn+1	Z1	TN15	TNn	TN0
1000	N2*	Left Shift of NEXT: NEXT is shifted left by 1 bit, with 0 shifted into the LSB. TOP and the carry bit are unchanged.	CY	Z15	Zn	Z0	TN14	TNn-1	0
1001	N2*c	Rotate NEXT Left: NEXT shifts left 1 bit, with the carry bit shifted into the LSB. TOP and the carry bit are unchanged.	CY	Z15	Zn	Z0	TN14	TNn-1	CY

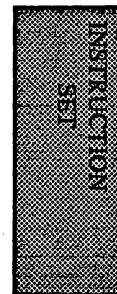
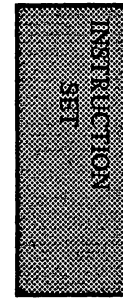
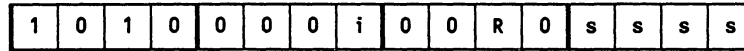


TABLE 7.6: 32-BIT SHIFT FUNCTIONS

Shift ssss	name	effect	Status of C	TOP Register			NEXT Register		
				T15	Tn	T0	N15	Nn	N0
1010	D2*	32-bit Left Shift: TOP and NEXT are shifted left 1 bit, with the MSB of NEXT shifted into the LSB of TOP, the MSB of TOP shifted into the carry bit, and 0 shifted into the LSB of NEXT.	Z15	Z14	Zn-1	TN15	TN14	TNn-1	0
1011	D2*c	32-bit Rotate Left: TOP and NEXT are shifted left 1 bit, the MSB of NEXT is shifted into the LSB of TOP, the carry bit is shifted into the LSB of NEXT, and the MSB of TOP is shifted into the carry bit.	Z15	Z14	Zn-1	TN15	TN14	TNn-1	CY
1100	cUD2/	32-bit Right Shift Out of Carry: TOP and NEXT are shifted right by 1 bit, the carry bit shifts into the MSB of TOP, the LSB of TOP is shifted into the MSB of NEXT, the LSB of NEXT is discarded, and 0 shifts into the carry bit.	0	CY	Zn+1	Z1	Z0	TNn+1	TN1
1101	cd2/	32-bit Rotate Right Through Carry: TOP and NEXT are shifted right by 1 bit, the carry bit shifts into the MSB of TOP, the LSB of TOP is shifted into the MSB of NEXT, and LSB of NEXT shifts into the carry.	TN0	CY	Zn+1	Z1	Z0	TNn+1	TN1
1110	UD2/	32-bit Logical Right Shift: TOP and NEXT are shifted right 1 bit with 0 shifted into MSB of TOP and the carry bit, the LSB of TOP is shifted into the MSB of NEXT, and LSB of NEXT is discarded.	0	0	Zn+1	Z1	Z0	TNn+1	TN1
1111	D2/	32-bit Arithmetic Right Shift: Bits 14-0 of TOP and all of NEXT are shifted right 1 bit; Bit 15 of TOP remains unchanged and is shifted into the carry bit and bit 14. The LSB of TOP is shifted into the MSB of NEXT; the LSB of NEXT is discarded.	Z15	Z15	Zn+1	Z1	Z0	TNn+1	TN1



Invert/Shift T



Description:

If i = 0 Performs shift operation ssss. Original contents of NEXT are left intact unless affected by shift operation.

If i = 1 Inverts TOP, performs shift operation ssss. Original contents of NEXT are left intact unless affected by shift operation.

Note that if both i and ssss are 0, this is a 1-cycle No Operation (NOP) instruction.

Number of cycles: 1

Processor operations:

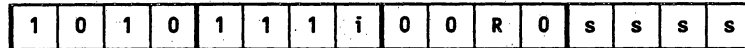
If i = 0 shift[T] => T

If i = 1 shift[*T] => T

Parameter Stack effect:

If i = 0 T -- shift[T]

If i = 1 T -- shift[*T]

ALU/SHIFT OPERATIONS**N => T, Invert/shift****N => T, Invert/shift****Description:**

If i = 0 Loads contents of TOP with contents of NEXT, then performs shift operation ssss. Original contents of NEXT are left intact unless affected by a shift operation.

If i = 1 Loads contents of TOP with contents of NEXT, inverting the value, then performs shift operation ssss. Original contents of NEXT are left intact unless affected by a shift operation.

Number of cycles: 1

Processor operations:

If i = 0 shift[N] => T

If i = 1 shift[*N] => T

Parameter Stack effect:

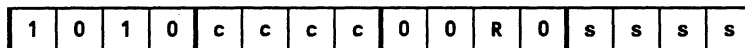
If i = 0 N T -- N shift[N]

If i = 1 N T -- N shift[*N]

ALU/SHIFT OPERATIONS

T-op-N Shift

T-op-N Shift



Description: Loads TOP with results of ALU operation cccc and shift operation ssss on TOP and NEXT registers. Original contents of NEXT are left intact unless affected by shift operation.

Number of cycles: 1

Processor operations:

shift[T-op-N] => T

Parameter Stack effect:

N T -- N shift[T-op-N]

ALU/SHIFT OPERATIONS

Invert/Shift, Pstack=>N

Invert/Shift, Pstack=>N

1	0	1	0	0	0	0	i	0	1	R	0	s	s	s	s
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Description:

If $i = 0$ Performs shift operation ssss on TOP and original contents of NEXT. Pops stack into NEXT.

If $i = 1$ Inverts TOP and performs shift operation ssss on TOP and original contents of NEXT. Pops stack into NEXT.

Number of cycles: 1

Processor operations:

If $i = 0$ Pstack => N shift[T] => T

If $i = 1$ Pstack => N shift[*T] => T

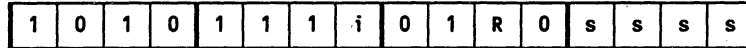
Parameter Stack effect:

If $i = 0$ N T -- shift[T]

If $i = 1$ N T -- shift[*T]

ALU/SHIFT OP N=>T, Invert/Shift, Pstack=>N

N=>T, Invert/Shift, Pstack=>N



Description:

If i = 0 Moves NEXT into TOP, performing shift operation ssss.
 Pops stack into NEXT.

If i = 1 Moves NEXT into TOP, inverting it, and performing shift
 operation ssss. Pops stack into NEXT.

Number of cycles: 1

Processor operations:

If i = 0 shift[N] => T Pstack => N

If i = 1 shift[*N] => T Pstack => N

Parameter Stack effect:

If i = 0 N T -- shift[N]

If i = 1 N T -- shift[*N]

ALU/SHIFT OPERATIONS **T-op-N, Shift, Pstack=>N**

T-op-N, Shift, Pstack=>N

1	0	1	0	c	c	c	c	0	1	R	0	s	s	s	s
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Description: Loads TOP with results of ALU operation cccc and shift operation ssss on TOP and NEXT registers. Pops stack into NEXT.

Number of cycles: 1

Processor operations:

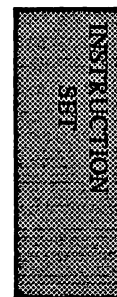
shift[T-op-N] => T Pstack => N

Parameter Stack effect:

N T -- shift[T-op-N]

ALU/SHIFT OPERATIONS

T = > N, Invert/Shift



T = > N, Invert/Shift

1	0	1	0	0	0	0	0	i	1	0	R	0	s	s	s	s
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Description:

- If i = 0** Copies TOP into NEXT, replacing original contents of NEXT. Performs shift operation ssss.
- If i = 1** Copies TOP into NEXT, inverting TOP and replacing original contents of NEXT. Performs shift operation ssss.

Number of cycles: 1

Processor operations:

- If i = 0** T => N shift[T] => T
- If i = 1** T => N shift[*T] => T

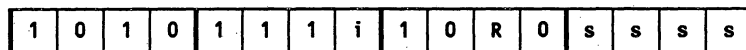
Parameter Stack effect:

- If i = 0** N T -- T shift[T]
- If i = 1** N T -- T shift[*T]

ALU/SHIFT OPERATIONS

T <=> N, Invert/Shift

T <=> N, Invert/Shift



Description:

If i = 0 Exchanges the contents of TOP and NEXT, then performs shift operation ssss.

If i = 1 Exchanges the contents of TOP and NEXT, inverting TOP (original contents of NEXT) then performs shift operation ssss.

Number of cycles: 1

Processor operations:

If i = 0 T => N shift[N] => T

If i = 1 T => N shift[*N] => T

Parameter Stack effect:

If i = 0 N T -- T shift[N]

If i = 1 N T -- T shift[*N]

ALU/SHIFT OPERATIONS**T-op-N, T=>N, Shift****T-op-N, T=>N, Shift**

1	0	1	0	c	c	c	c	1	0	R	0	s	s	s	s
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Description: Loads TOP with results of ALU operation cccc and shift operation ssss on TOP and NEXT registers. Loads NEXT with original contents of TOP.

Number of cycles: 1

Processor operations:

T => N T-op-N => T

Parameter Stack effect:

N T -- T shift[T-op-N]

ALU/SHIFT OP**N => Pstack, T => N, Invert/Shift****N => Pstack, T => N, Invert/Shift**

1	0	1	0	0	0	0	0	i	1	1	R	0	s	s	s	s
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Description:

If i = 0 Pushes original contents of NEXT onto stack, copies TOP into NEXT, and performs shift operation ssss.

If i = 1 Pushes original contents of NEXT onto stack, copies TOP into NEXT, inverts TOP, and performs shift operation ssss.

Number of cycles: 1

Processor operations:

If i = 0 N => Pstack T => N shift[T] => T

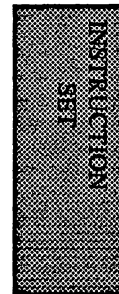
If i = 1 N => Pstack T => N shift[*T] => T

Parameter Stack effect:

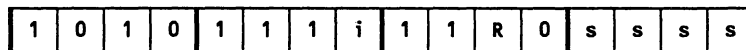
If i = 0 N T -- N T shift[T]

If i = 1 N T -- N T shift[*T]

ALU/SHIFT OP $N \Rightarrow Pstack, T \Leftarrow N, Invert/Shift$



$N \Rightarrow Pstack, T \Leftarrow N, Invert/Shift$



Description:

- If $i = 0$** Pushes NEXT onto stack, pushes TOP to NEXT, copies original contents of NEXT to TOP, and performs shift operation ssss.
- If $i = 1$** Pushes NEXT onto stack, pushes TOP to NEXT, and copies original contents of NEXT to TOP. Inverts TOP (original contents of NEXT), and performs shift operation ssss.

Number of cycles: 1

Processor operations:

- If $i = 0$** $N \Rightarrow Pstack$ $T \Rightarrow N$ $shift[N] \Rightarrow T$
- If $i = 1$** $N \Rightarrow Pstack$ $T \Rightarrow N$ $shift[*N] \Rightarrow T$

Parameter Stack effect:

- If $i = 0$** $N T \text{ -- } N T \text{ shift}[N]$
- If $i = 1$** $N T \text{ -- } N T \text{ shift}[*N]$

ALU/SHIFT OPERATIONS N=>Pstack, T-op-N, Shift

N=>Pstack, T-op-N, Shift

1	0	1	0	c	c	c	c	1	1	R	0	s	s	s	s
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Description: Pushes NEXT onto stack, pushes TOP into NEXT, loads TOP with results of ALU operation cccc and shift operation ssss on original contents of TOP and NEXT registers.

Number of cycles: 1

Processor operations:

N => Pstack T => N shift[T-op-N] => T

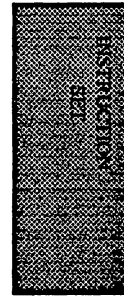
Parameter Stack effect:

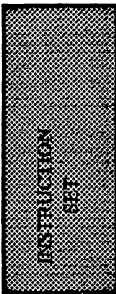
N T -- N T shift[T-op-N]

7.7 Enhanced Processor-Specific Operations

Each member of the RTX 2000 Family of Microcontrollers has on-chip hardware which is specifically designed to support operational requirements in the field of applications for which that Microcontroller is intended.

Utilization of these microprocessor hardware features to achieve enhanced performance is possible through use of the product specific instructions for each microcontroller.





RTX 2000 Specific Instructions

Unsigned Multiply

MULU



does i bit really work here?

Description: The Unsigned Multiply operation is initiated. The contents of the TOP and NEXT registers are multiplied, with the 32-bit result available in the Multiplier output registers MHR, MLR. Interrupts are disabled during the execution of this instruction. This instruction does not modify the stack.

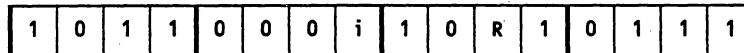
Number of cycles: 1

Processor operations:

T*N => MHR:MLR

Parameter Stack effect:

no effect

MULS

Description: The Signed Multiply operation is initiated. The contents of the TOP and NEXT registers are multiplied, with the 32-bit result available in the Multiplier output registers MHR, MLR. Interrupts are disabled during the execution of this instruction. This instruction does not modify the stack.

Number of cycles: 1

Processor operations:

T*N => MHR:MLR

Parameter Stack effect:

no effect

RTX 2000 Specific

Read Multiplier High Register

MHR@

1	0	1	1	1	1	1	1	i	0	0	R	1	0	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Description: The middle 16 bits of the Multiplier High Register (MHR) are pushed onto the Parameter Stack. The contents of TOP are pushed into NEXT, but NEXT is not pushed onto the stack; the contents of NEXT are lost. Interrupts are disabled during the execution of this instruction.

Number of cycles: 1

Processor operations:

MHR => T

T => N

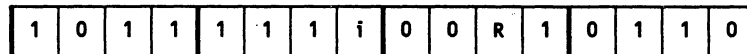
Parameter Stack effect:

N T -- T MHR

RTX 2000 Specific

Read Multiplier Low Register

MLR@



Description: The low 16 bits of the Multiplier Low Register (MLR) are pushed onto the parameter stack. The contents of TOP are pushed into NEXT, but NEXT is not pushed onto the stack; the contents of NEXT are lost. Interrupts are disabled during the execution of this instruction.

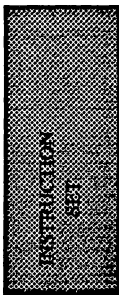
Number of cycles: 1

Processor operations:

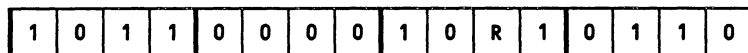
MLR => T T => N

Parameter Stack effect:

N T -- T MLR



Increment **RX**



Description: Increments the contents of **RX** by one. Incrementing the contents of the register beyond FFFF Hex results in a wrap to 0000 Hex.

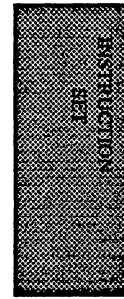
Number of cycles: 1

Processor operations:

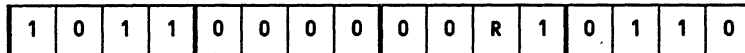
(**RX**) -> (**RX**) + 1

Parameter Stack effect:

no change



Decrement *RX*



Description: Decrements the contents of *RX* by one. Decrementing the contents of the register beyond 0000 Hex results in a wrap to FFFF Hex.

Number of cycles: 1

Processor operations:

$$(RX) \rightarrow (RX) - 1$$

Parameter Stack effect:

no change

RTX 2010 Specific Instructions

0=

0=

1	0	1	1	0	0	0	0	0	0	R	0	1	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Description:

If **TOP = 0** Change TOP to FFFF (implement Forth **0=**).

If **TOP ≠ 0** Change TOP to 0000 (implement Forth **0=**).

Number of cycles: 1

Processor operations:

If **TOP = 0** FFFF => T

If **TOP ≠ 0** 0000 => T

Parameter Stack effect:

n – b

RTX 2010 Specific

Clear MAC Accumulator

CLEARACC

1	0	1	1	0	0	0	0	0	0	R	0	1	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Description: Clear the MAC Accumulator (MXR, MHR, MLR).

Number of cycles: 1

Processor operations:

0 => MLR

0 => MHR

0 => MXR

Parameter Stack effect:

no change

DSLL

1	0	1	1	0	0	0	0	0	0	R	0	1	1	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Description: **Double Shift Left Logical** Shift the double word operand in TOP and NEXT left logically by the 5-bit count stored in the MXR Register. The result is stored in MHR and MLR

Number of cycles: 1

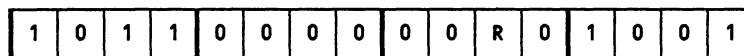
Processor operations:

DSLL (T:N) => MHR:MLR

Parameter Stack effect:

no effect

DSRA



Description: **Double Shift Right Arithmetic** Shift the double word operand in TOP and NEXT right arithmetically by the 5-bit count stored in the MXR Register. The result is stored in MHR and MLR.

Number of cycles: 1

Processor operations:

DSRA (T:N) => MHR:MLR

Parameter Stack effect:

no effect

DSRL

1	0	1	1	0	0	0	0	0	0	R	0	1	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Description: **Double Shift Right Logical** Shift the double word operand in TOP and NEXT right logically by the 5-bit count stored in the MXR Register. The result is stored in MHR and MLR.

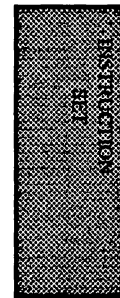
Number of cycles: 1

Processor operations:

DSRL (T:N) => MHR:MLR

Parameter Stack effect:

no effect



RTX 2010 Specific

Store MAC High Register

MHR!

1	0	1	1	1	1	1	i	1	0	R	1	0	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Description: Store the contents of TOP into the MAC Accumulator MHR. NEXT is popped into TOP and Pstack is popped into NEXT.

Number of cycles: 1

Processor operations:

T => MHR

N => T

Pstack => N

Parameter Stack effect:

MHR --

MHR@

1	0	1	1	1	1	1	1	i	0	0	R	1	0	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Description: The middle 16 bits of the MAC register (MHR) are pushed onto the Parameter Stack. The contents of TOP are pushed into NEXT, but NEXT is not pushed onto the stack; the contents of NEXT are lost. Interrupts are disabled during the execution of this instruction.

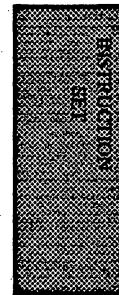
Number of cycles: 1

Processor operations:

MHR => T T => N

Parameter Stack effect:

N T -- T MHR



RTX 2010 Specific

Store MAC Low Register

MLR!

1	0	1	1	1	1	1	i	1	0	R	1	0	1	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Description: Store the contents of TOP into the MAC Accumulator MLR. NEXT is popped into TOP and Pstack is popped into NEXT.

Number of cycles: 1

Processor operations:

T => MLR

N => T

Pstack => N

Parameter Stack effect:

MLR --

RTX 2010 Specific

Read Multiplier Low Register

MLR@

1	0	1	1	1	1	1	1	i	0	0	R	1	0	1	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Description: The low 16 bits of the MAC register (MLR) are pushed onto the parameter stack. The contents of TOP are pushed into NEXT, but NEXT is not pushed onto the stack; the contents of NEXT are lost. Interrupts are disabled during the execution of this instruction.

Number of cycles: 1

Processor operations:

MLR => T T => N

Parameter Stack effect:

N T -- T MLR

MULACM

1	0	1	1	1	1	1	1	i	0	0	R	1	0	1	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Description: The Mixed Mode (signed and unsigned) Multiply Accumulate operation is initiated. The contents of the TOP and NEXT registers are multiplied (TOP contains the signed value and NEXT contains the unsigned value), the 32-bit result is added to the 48-bit accumulator (MXR, MHR, MLR). This instruction does not modify the stack.

Number of cycles: 1

Processor operations:

$(T*N)+MXR:MHR:MLR \Rightarrow MXR:MHR:MLR$

Parameter Stack effect:

no effect

MULACS

1	0	1	1	0	0	0	i	0	0	R	1	0	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Description: A Signed Multiply Accumulate operation is initiated. The contents of the TOP and NEXT registers are multiplied, the 32-bit result is added to the 48-bit accumulator (MXR, MHR, MLR). This instruction does not modify the stack.

Number of cycles: 1

Processor operations:

$(T*N)+MXR:MHR:MLR \Rightarrow MXR:MHR:MLR$

Parameter Stack effect:

no effect

RTX 2010 Specific

Unsigned Multiply Accumulate

MULACU

1	0	1	1	0	0	0	i	0	0	R	1	0	1	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Description: The Unsigned Multiply Accumulate operation is initiated. The contents of the TOP and NEXT registers are multiplied, the 32-bit result is added to the 48-bit accumulator (MXR, MHR, MLR). This instruction does not modify the stack.

Number of cycles: 1

Processor operations:

$(T*N)+MXR:MHR:MLR \Rightarrow MXR:MHR:MLR$

Parameter Stack effect:

no effect

MULM

1	0	1	1	0	0	0	i	0	0	R	1	0	0	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Description: The Mixed Sign Multiply operation is initiated. The contents of the TOP and NEXT registers are multiplied, with the 32-bit result available in the MAC output registers MHR, MLR. The operand in TOP is assumed to be signed and the operand in NEXT unsigned. Interrupts are disabled during the execution of this instruction. This instruction does not modify the stack.

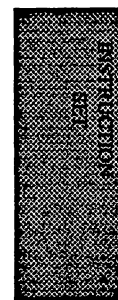
Number of cycles: 1

Processor operations:

T*N => MHR:MLR

Parameter Stack effect:

no effect



MULS

1	0	1	1	0	0	0	i	1	0	R	1	0	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Description: The Signed Multiply operation is initiated. The contents of the TOP and NEXT registers are multiplied, with the 32-bit result available in the MAC output registers MHR, MLR. Interrupts are disabled during the execution of this instruction. This instruction does not modify the stack.

Number of cycles: 1

Processor operations:

T*N => MHR:MLR

Parameter Stack effect:

no effect

MULSUB

1	0	1	1	0	0	0	i	0	0	R	1	0	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Description: The Signed Multiply and Subtract from Accumulator operation is initiated. The contents of the TOP and NEXT registers are multiplied, the 32-bit result is subtracted from the 48-bit accumulator (MXR, MHR, MLR). This instruction does not modify the stack.

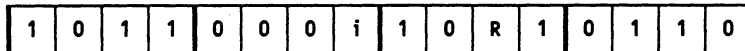
Number of cycles: 1

Processor operations:

MXR:MHR:MLR-(T*N) => MXR:MHR:MLR

Parameter Stack effect:

no effect

MULU

Description: The Unsigned Multiply operation is initiated. The contents of the TOP and NEXT registers are multiplied, with the 32-bit result available in the MAC output registers MHR, MLR. Interrupts are disabled during the execution of this instruction. This instruction does not modify the stack.

Number of cycles: 1

Processor operations:

$T*N \Rightarrow MHR:MLR$

Parameter Stack effect:

no effect

RTX 2010 Specific

Read MAC Extension Register

MXR@

1	0	1	1	1	1	1	i	0	0	R	1	0	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Description: Pushes the contents of the 16-bit extension register of the MAC output onto the parameter stack.

Number of cycles: 1

Processor operations:

MXR => T T => N N => Pstack

Parameter Stack effect:

-- MXR

RTX 2010 Specific

Store MAC Extension Register

MXR!

1	0	1	1	1	1	1	i	1	0	R	1	0	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Description: Stores the contents of TOP into the MAC Accumulator MXR. NEXT is popped into TOP and the Parameter Stack is popped into NEXT.

Number of cycles: 1

Processor operations:

T => MXR

N => T

Pstack => N

Parameter Stack effect:

MXR --

NORM

1	0	1	1	0	0	0	i	0	0	R	0	1	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Description: The **Normalize** operation counts the number of leading zeros in the double word operand in **TOP** and **NEXT**. This count is stored in **MXR**. **TOP** and **NEXT** are also shifted left logically by this count to eliminate all leading zeros. The shifted result is in **MHR** and **MLR**.

Number of cycles: 1

Processor operations:

NORM(T:N) => MHR:MLR

Count => MXR

Parameter Stack effect:

no effect

RTX 2010 Specific

Shift MAC Register Right

RSACC

1	0	1	1	0	0	0	i	0	0	R	1	0	0	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Description: This operation shifts the MAC output registers right (MXR -> MHR, MHR -> MLR, sign fills MXR, contents of MLR are lost.) This is useful in implementing double precision multiply operations.

Number of cycles: 1

Processor operations:

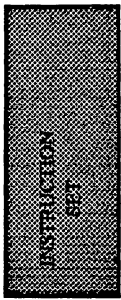
MXR => MHR

MHR => MLR

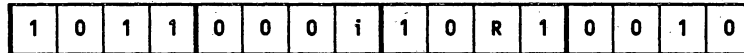
signfill => MXR

Parameter Stack effect:

no effect



SMACA



Description: Streamed MAC between ASIC Bus and Memory is an instruction which indicates to the processor that the next instruction is a streamed instruction that will initiate a streamed MAC between the ASIC bus and memory. See Section 7.7.1 for detailed information.

Number of cycles: 1

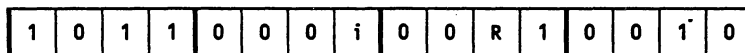
Processor operations:

N/A

Parameter Stack effect:

no effect

SMACS



Description: Streamed MAC between Stack and Memory is an instruction which indicates to the processor that the next instruction is a streamed instruction that will initiate a streamed MAC between the stack and memory. See Section 7.7.1 for detailed information.

Number of cycles: 1

Processor operations:

N/A

Parameter Stack effect:

no effect

7.7.1 Streamed MAC Instructions On The RTX 2010

One of the features of the RTX-2010 is the ability to perform a high speed (one clock cycle per iteration) multiplication and accumulation of two streams of data. One stream is a list of data in external memory, and the other is either a list of data on the parameter stack or a stream of values input from the ASIC bus.

SMACA is the instruction to initiate a streamed MAC with the ASIC bus and SMACS initiates a streamed MAC with list on the Parameter Stack. In general, the code to implement the algorithm is as follows:

```
mem_addr count-1 SMACS OF( "DMA"
mem_addr g-addr SR! count-1 SMACA OF( "DMA"
```

where:

mem_addr	is the address of first data item in memory list
count-1	is the number of items in list
OF(is the instruction to implement streaming
"DMA"	is instruction to read sequential items from memory.
g-addr	is the ASIC Bus address to be streamed

The "SMACA" and "SMACS" instructions are used to set the source of one input data stream, and are part of a special opcode sequence which is used exclusively in the RTX-2010. These commands initiate a processor state which affects the operation of the instructions that follow it. The instructions for streamed MAC must occur in the sequence previously described. SMACA and SMACS suppress interrupts so that the of(instruction is guaranteed to follow without interruptions.

The "DMA" opcode is an existing RTX instruction that normally reads data from memory into NEXT, over writing the data in NEXT. During each cycle, the contents of the memory location addressed by TOP are read into the NEXT Register. Concurrently, the contents of TOP are incremented by the value of the five bit literal field of the instruction, to generate the next memory address to be read.

When used following the SMACA or SMACS instructions, the operation of the "DMA" opcode is modified. In the special processor state that is initiated by SMACA or SMACS, the DMA instructions cause the contents of NEXT to be multiplied by the contents of a pipeline register from the stack or ASIC Bus, and added to the 48-bit accumulator. In the same clock cycle, the DMA instruction also performs its normal operation, reading the next data item from memory into NEXT and auto-incrementing the address in TOP.

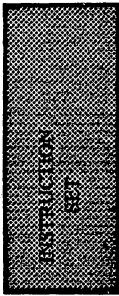
The following is an example of a streamed multiply/accumulate operation between a list of data in memory and another list of data on the stack. The stack list is assumed to be in another stack area, necessitating the saving and restoring of a stack pointer.

```
\ ( mem_addr count-1 stack_addr -- answer )
  SPR@ MD! SPR! \ save & set stack pointer
  SMACS          \ set streamed mac instruction execution
  OF(           \ set stream count
    DMA         \ execute streamed MAC DMA
  DROP DROP     \ eliminate address and last data from stack
  MD@ SPR!      \ restore the stack pointer
  MLR@ MHR@ MXR@ \ fetch 48-bit accumulated value
```

The next example performs the same operation, with the exception of the second argument being a stream of input from the ASIC Bus data. The SMACA instruction requires the desired ASIC address to be stored in the SR Register.

```
\ ( mem_addr count-1 asic_addr -- answer )
  SR!          \ set ASIC bus address
  SMACA        \ set streamed mac instruction execution
  OF(         \ set stream count
    DMA       \ execute streamed MAC DMA
  DROP DROP   \ eliminate address and last data from stack
  MLR@ MHR@ MXR@ \ fetch accumulated value
```

The opcode for DMA is E842H.



7.8 Class 11-a : ASIC Bus Access

This class of instructions manipulates the contents of devices attached to the ASIC Bus. This includes the RTX internal registers (Index Register, Configuration Register, Multi-step/Multiply Divide, Square Root, and Program Counter), On-Chip Peripheral devices (Stack Controllers, Interrupt Controller, Multiplier, Counters), and external I/O devices such as UARTs or SCSI controllers.

The processor is able to directly access 32 ASIC Bus devices/registers. The specific ASIC Bus address is encoded as a 5-bit field in the instruction (indicated by "ggggg" in the instruction formats). See Section 7.8.1 for more information in instructions which use this instruction format.

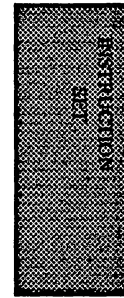
ASIC Addresses 0 - 17H are used internally by the RTX processor for registers. Chapter 4 describes the register address assignments. Some of these addresses perform special functions when referenced with different forms of the ASIC Bus instructions. Section 7.8.2 describes these special instruction forms.

ASIC Addresses 18 - 31H are provided for access to off-chip ASIC devices.

7.8.1 ASIC Bus Instructions

Instructions which access ASIC Bus locations have the specific location encoded as a 5-bit field in the instruction. This field is indicated by "ggggg" in the instruction formats. The 5-bit field enables the processor to directly access 32 ASIC Bus devices/registers.

Some of the ASIC Bus instructions perform ALU operations on the data accessed. These operations are indicated by "cccc" in the instruction formats and are the same as those described in the "Single-step Math Functions" class.



G-read, DROP, Invert



Description:

- If i = 0** Reads and discards data from ASIC address ggggg. Useful for performing "dataless" I/O accesses, in which an I/O device needs to be addressed, but no data transfer needs to take place. See Section 7.7.1 for limitations on the use of this opcode.
- If i = 1** Reads and discards data from ASIC address ggggg, inverts TOP. See remainder of description above.

Huh?
do you mean
7.7?
or does this refer to
specific instructions

Number of cycles: 1

Processor operations:

If i = 0 T => T

If i = 1 *T => T

Parameter Stack effect:

If i = 0 N T -- N T

If i = 1 N T -- N *T

ASIC Access $N \Rightarrow Pstack, T \Rightarrow N, (ggggg) \Rightarrow T, Invert$

$N \Rightarrow Pstack, T \Rightarrow N, (ggggg) \Rightarrow T, Invert$

1	0	1	1	1	1	1	i	0	0	R	g	g	g	g	g
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Description:

If $i = 0$ Pushes NEXT onto stack, TOP into NEXT, then reads data from address ggggg into TOP.

If $i = 1$ Pushes NEXT onto stack, TOP into NEXT, then reads data from address ggggg into TOP, inverting data.

Number of cycles: 1

Processor operations:

If $i = 0$ $N \Rightarrow Pstack$ $T \Rightarrow N$ $(ggggg) \Rightarrow T$

If $i = 1$ $N \Rightarrow Pstack$ $T \Rightarrow N$ $*(ggggg) \Rightarrow T$

Parameter Stack effect:

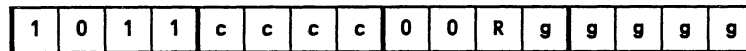
If $i = 0$ $N T -- N T d$

If $i = 1$ $N T -- N T *d$

ASIC Access

$N = > Pstack, T = > N, T-op-(ggggg)$

$N = > Pstack, T = > N, T-op-(ggggg)$



Description:

Pushes NEXT onto stack, TOP into NEXT, then reads data from ASIC address ggggg and loads TOP with results of ALU operation cccc on original contents of TOP and data.

Number of cycles:

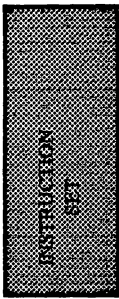
1

Processor operations:

$N \Rightarrow Pstack \quad T \Rightarrow N \quad T-op-(ggggg) \Rightarrow T$

Parameter Stack effect:

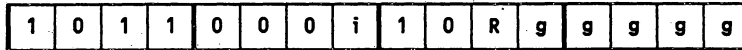
$N T \rightarrow N T T-op-d$



ASIC Access

T = > (ggggg), Invert

T = > (ggggg), Invert



Description:

If i = 0 Writes contents of TOP to ASIC address ggggg. Original contents of NEXT are left intact. See Section 7.7.1 for limitations on the use of this opcode.

If i = 1 Writes contents of TOP to ASIC address ggggg. Inverts original contents of TOP. Original contents of NEXT are left intact.

cryptic?

Number of cycles: 1

Processor operations:

If i = 0 T => (ggggg) T => T

If i = 1 T => (ggggg) *T => T

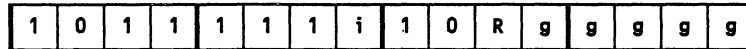
Parameter Stack effect:

If i = 0 N T -- N T

If i = 1 N T -- N *T

ASIC Access $T = > (ggggg), N = > T, Pstack = > N, Invert$

$T = > (ggggg), N = > T, Pstack = > N, Invert$



Description:

If $i = 0$ Writes contents of TOP to ASIC address ggggg. Copies NEXT into TOP. Pops stack into NEXT.

If $i = 1$ Writes contents of TOP to ASIC address ggggg. Copies NEXT into TOP, inverting value. Pops stack into NEXT.

Number of cycles: 1

Processor operations:

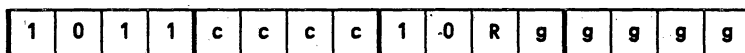
If $i = 0$ $T \Rightarrow (ggggg)$ $N \Rightarrow T$ $Pstack \Rightarrow N$

If $i = 1$ $T \Rightarrow (ggggg)$ $*N \Rightarrow T$ $Pstack \Rightarrow N$

Parameter Stack effect:

If $i = 0$ $N T -- N$

If $i = 1$ $N T -- *N$

ASIC Access**(ggggg)-op-T****(ggggg)-op-T**

Description: Reads data from ASIC address ggggg, and loads TOP with results of ALU operation cccc on contents of TOP and data. Original contents of NEXT are left unchanged.

Number of cycles: 1

Processor operations:

(ggggg)-op-T => T

Parameter Stack effect:

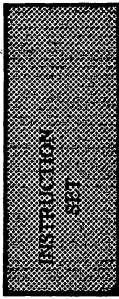
N T -- N d-op-T

7.8.2 Predefined ASIC Bus Instructions

Some RTX ASIC Bus opcodes are predefined to perform specific functions. These include "Select Data Page Register", "Select Code Page Register", "Software Interrupt", and "Remove Software Interrupt". Descriptions of these opcodes follow.

The Multiply, MAC, and Barrel Shifter are also controlled using predefined ASIC instructions. See Section 7.7 for descriptions.

why isn't this
grouped ~~set~~ with
section 7.7?



Memory Page Access

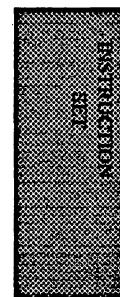
Select Data Page Register

Select Data Page Register

1 0 1 1 0 0 0 0 1 0 R 0 1 1 0 1

Description: Sets DPRSEL Bit to 1, causing all data memory accesses to be addressed through the Data Page Register.

*is this an
done this as an
work as an
1 bit?*



Select Code Page Register

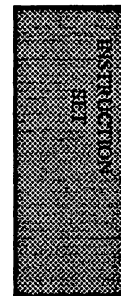
1	0	1	1	0	0	0	0	0	0	R	0	1	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Description: Sets DPRSEL Bit to 0, causing all data memory accesses to be addressed through the Code Page Register.

Set SOFTINT

1	0	1	1	0	0	0	0	1	0	R	1	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

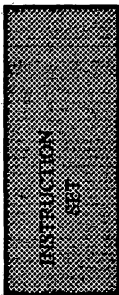
Description: Sets the Software Interrupt Request flip-flop, generating a Level 13 interrupt to the processor. Due to the time required by the processor internally to generate and process the interrupt signal, this instruction should be followed by two 1-cycle instructions which do not depend on whether or not the interrupt has been serviced (NOPs, for example).



Clear SOFTINT

1	0	1	1	0	0	0	0	0	0	0	R	1	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Description: Resets Software Interrupt Request flip-flop. The interrupt service routine for the Software Interrupt (level 13) must execute this instruction before re-enabling interrupts or executing a subroutine return.

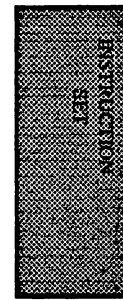


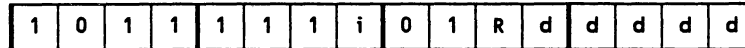
7.9 Class 11b - Short Literals

This class of instructions generates short literals (positive values 0 to 31).

The value of the literal is embedded in the instruction and is shown as "dddd" in the instruction formats. The value represented by "dddd" is loaded into bits 0-4 of the TOP register; bits 5-15 are set to 0. If the value is inverted by having the "i" bit set in the instruction, all 16 bits of the value are inverted.

Some of these instructions perform ALU operations using the literal data. These operations are indicated by "cccc" in the instruction formats and correspond to Table 7.3.



d Invert
**Description:**

If i = 0 Pushes NEXT onto stack, copies TOP into NEXT, loads the value ddddd into TOP.

If i = 1 Pushes NEXT onto stack, copies TOP into NEXT, loads the value ddddd into TOP, and inverts the value.

Number of cycles: 1

Processor operations:

If i = 0 N => Pstack T => N ddddd => T

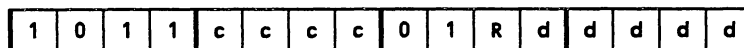
If i = 1 N => Pstack T => N *dddd => T

Parameter Stack effect:

If i = 0 N T -- N T ddddd

If i = 1 N T -- N T *dddd

Short Literals

N=>Pstack, T=>N, T-op-d**N=>Pstack, T=>N, T-op-d**

Description: Pushes NEXT onto stack, copies TOP into NEXT, then loads TOP with result of ALU operation cccc between contents of TOP and value dddd.

Number of cycles: 1

Processor operations:

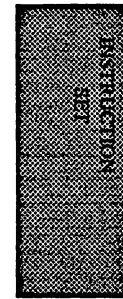
N => Pstack T => N T-op-ddddd => T

Parameter Stack effect:

N T -- N T T-op-ddddd

Short Literals

d = >T, Invert



d = >T, Invert

1	0	1	1	1	1	1	i	1	1	R	d	d	d	d	d
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Description:

- If i = 0** Loads the value ddddd into TOP. Original contents of NEXT are left unchanged.
- If i = 1** Loads the value ddddd into TOP, inverting all 16 bits of the value. Original contents of NEXT are left unchanged.

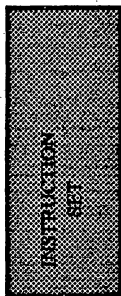
Number of cycles: 1

Processor operations:

- If i = 0** ddddd => T
- If i = 1** *dddd => T

Parameter Stack effect:

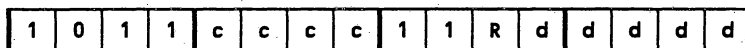
- If i = 0** N T -- N ddddd
- If i = 1** N T -- N *dddd



Short Literals

d-op-T

d-op-T



Description: Loads TOP with results of ALU operation cccc between value dddd and contents of TOP. Original contents of NEXT are left unchanged.

Number of cycles: 1

Processor operations:

dddd-op-T => T

Parameter Stack effect:

N T -- N dddd-op-T

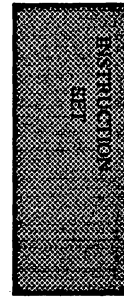
7.10 Class 12: User Memory Access

This class of instructions performs reads and writes to the User Memory Space. The User Page Register and User Base Register determine the address of the 32-word user memory block.

The address to be accessed within the User Space is encoded as a 5-bit field in the instruction, and is indicated by "uuuu" in the descriptions. Note that "uuuu" represents the word address of the location to be referenced. For example, uuuu = 3 will perform a read or write to word #3 (byte #6) in the User Space. All User Memory Space accesses read or write a 16-bit value.

The data written to or read from the User location is indicated in the descriptions as "(u)".

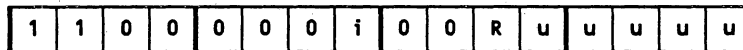
Some of these instructions perform ALU operations using the memory data. These operations are indicated by "cccc" in the instruction formats and correspond to Table 7.3.



User Memory Access

$N = > Pstack, (u) = > N, Invert$

$N = > Pstack, (u) = > N, Invert$



Description:

If $i = 0$ Pushes NEXT onto stack, then reads data from user location uuuuu into NEXT.

If $i = 1$ Pushes NEXT onto stack, then reads data from user location uuuuu into NEXT. Inverts TOP.

Number of cycles: 2

Processor operations:

If $i = 0$

1st cycle:	$N \Rightarrow Pstack$	$(u) \Rightarrow N$
2nd cycle:	NOP	

If $i = 1$

1st cycle:	$N \Rightarrow Pstack$	$(u) \Rightarrow N$
2nd cycle:	$T \Rightarrow *T$	

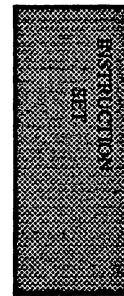
Parameter Stack Effect:

If $i = 0$ $N \ T \ \dots \ N \ (u) \ T$

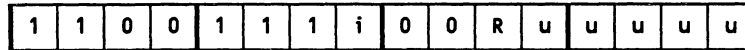
If $i = 1$ $N \ T \ \dots \ N \ (u) \ *T$

User Memory Access

(u) => T, Invert



(u) => T, Invert



Description:

- and pushes into*
- If i = 0** Reads data from user location uuuuu into TOP. Original contents of NEXT are left unchanged.
- If i = 1** Reads data from user location uuuuu into TOP, inverting data. Original contents of NEXT are left unchanged.

Number of cycles: 2

Processor operations:

- T => p stack*
- If i = 0**
- | | | |
|------------|--------|----------|
| 1st cycle: | N => T | (u) => N |
| 2nd cycle: | N => T | T => N |
- If i = 1**
- | | | |
|------------|---------|----------|
| 1st cycle: | N => T | (u) => N |
| 2nd cycle: | *N => T | T => N |

Parameter Stack effect:

- If i = 0** N-T --- N-(u) → N T -- N T (u)
- If i = 1** N-T --- N*(u) → N T -- N T *(u)

User Memory Access

$N = > Pstack, T = > N, T-op-(u) = > T$

1	1	0	0	c	c	c	c	0	0	R	u	u	u	u
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Description: Pushes NEXT onto stack, TOP into NEXT, then loads TOP with result of ALU operation cccc on contents of TOP and data read from user location uuuuu.

Number of cycles: 2

Processor operations:

1st cycle: $N \Rightarrow Pstack$ $(u) \Rightarrow N$

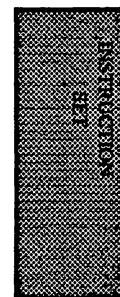
2nd cycle: $T \Rightarrow N$ $N-op-T \Rightarrow T$

Parameter Stack effect:

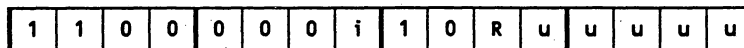
$N T -- N T T-op-(u)$

User Memory Access

T = > uuuuu, Invert



T = > uuuuu, Invert



Description:

- If i = 0** Writes the contents of TOP to user location uuuuu. Original contents of NEXT are left unchanged.
- If i = 1** Writes the contents of TOP to user location uuuuu. Inverts contents of TOP (after write operation). Original contents of NEXT are left unchanged.

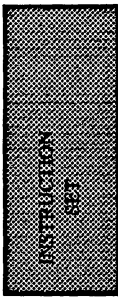
Number of cycles: 2

Processor operations:

- If i = 0** 1st cycle: T => uuuuu
 2nd cycle: T => T
- If i = 1** 1st cycle: T => uuuuu
 2nd cycle: *T => T

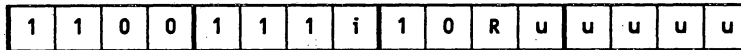
Parameter Stack effect:

- If i = 0** N T -- N T
- If i = 1** N T -- N *T



User Memory Access

T => uuuuu, N => T, Pstack => N, Invert



Description:

If i = 0 Writes contents of TOP to user location uuuuu. Moves NEXT into TOP. Pops stack into NEXT.

If i = 1 Writes contents of TOP to user location uuuuu. Moves NEXT into TOP, inverting the value. Pops stack into NEXT.

Number of cycles: 2

Processor operations:

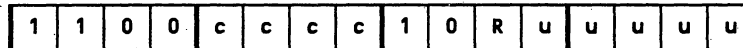
If i = 0 1st cycle: T => uuuuu
 2nd cycle: N => T Pstack => N

If i = 1 1st cycle: T => uuuuu
 2nd cycle: *N => T Pstack => N

Parameter Stack effect:

If i = 0 N T -- N

If i = 1 N T -- *N

User Memory Access**T-op-(u) = > T****T-op-(u) = > T**

Description: Loads TOP with results of ALU operation cccc on contents of TOP and data read from user location uuuu. Original contents of NEXT are left unchanged.

Number of cycles: 2

Processor operations:

1st cycle: N => Pstack (u) => N
 2nd cycle: T-op-N => T Pstack => N

Parameter Stack effect:

N T -- N T-op-(u)

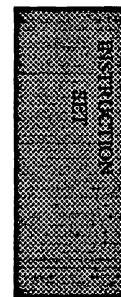
7.11 Class 13: Long Literals

This class of instructions generates 16-bit literal values. The 16-bit value is contained in the memory location following the Long Literal instruction. The value contained in this location is identified in the descriptions by "D". Long Literal instructions are the only RTX instructions which occupy two memory locations.

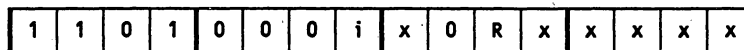
Some of these instructions perform ALU operations using the literal data. These operations are indicated by "ccc" and correspond to Table 7.3.

Long Literals

$N = > Pstack, D = > N, Invert$



$N = > Pstack, D = > N, Invert$



Description:

If $i = 0$ Pushes NEXT onto stack, loads literal value into NEXT.

If $i = 1$ Pushes NEXT onto stack, loads literal value into NEXT, inverts TOP.

Number of cycles: 2

Processor operations:

If $i = 0$

1st cycle:	$N \Rightarrow Pstack$	$D \Rightarrow N$
2nd cycle:	NOP	

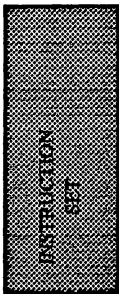
If $i = 1$

1st cycle:	$N \Rightarrow Pstack$	$D \Rightarrow N$
2nd cycle:	$*T \Rightarrow T$	

Parameter Stack effect:

If $i = 0$ N T -- N D T

If $i = 1$ N T -- N D *T



Long Literals

N => Pstack, T => N, D => T, Invert

N => Pstack, T => N, D => T, Invert

1	1	0	1	1	1	1	i	0	0	R	x	x	x	x	x
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Description:

If i = 0 Pushes NEXT onto stack, TOP into NEXT, then loads literal value into TOP.

If i = 1 Pushes NEXT onto stack, TOP into NEXT, then loads literal value into TOP, inverting the value.

Number of cycles: 2

Processor operations:

If i = 0	1st cycle:	N => Pstack	D => N
	2nd cycle:	T => N	N => T

If i = 1	1st cycle:	N => Pstack	D => N
	2nd cycle:	T => N	*N => T

Parameter Stack effect:

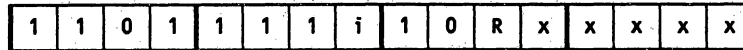
If i = 0 NT -- NT D

If i = 1 NT -- NT *D

Long Literals

D=>T, Invert

D=>T, Invert



Description:

If i = 0 Loads literal value into TOP. Original contents of NEXT are left unchanged.

If i = 1 Loads literal value into TOP, inverting the value. Original contents of NEXT are left unchanged.

Number of cycles: 2

Processor operations:

If i = 0

1st cycle:	N => Pstack	D => N
2nd cycle:	N => T	Pstack => N

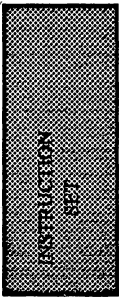
If i = 1

1st cycle:	N => Pstack	D => N
2nd cycle:	*N => T	Pstack => N

Parameter Stack effect:

If i = 0 N T -- N D

If i = 1 N T -- N *D



7.12 Classes 14 and 15: Data Memory Access

These two classes of instruction perform reads and writes to Data Memory Space. The DPRSEL bit controls which page address register selects the memory page.

The address within the page of the location to be accessed is contained in the TOP register. For memory reads, the data is moved from memory into the NEXT register. For memory writes, the data moves from NEXT into memory.

The instruction formats are identical for both word and byte access. The "s" bit (bit 12) of the instruction dictates the size of the operand (s = 0 for 16-bit word, s = 1 for 8-bit byte).

For byte writes to memory, the contents of bits 0-7 of NEXT are written to the memory location addressed by TOP. For byte reads, the memory data is read into bits 0-7 of NEXT; bits 8-15 of NEXT are set to 0.

The data read from or written to memory is identified in the descriptions as "m". Short literals are identified as "dddd". The memory location addressed by the contents of TOP is identified as "T".

Some of the instructions may perform ALU operations on the data. These operations are identified by either "cccc" or "aaa" in the instruction format, and correspond to the values in Table 7.3.

Why?
TE contents
of TOP
~~not~~
~~just~~

do you mean
"(T)"?

Data Memory Access

$N \Rightarrow T, m \Rightarrow N, \text{Invert}$



$N \Rightarrow T, m \Rightarrow N, \text{Invert}$

1	1	1	s	0	0	0	i	0	0	R	x	x	x	x	x
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Description:

If $i = 0$ Moves NEXT into TOP. Loads memory data contained in the location addressed by TOP into NEXT.

If $i = 1$ Moves NEXT into TOP, inverting value. Loads memory data contained in the location addressed by TOP into NEXT.

Number of cycles: 2

Processor operations:

If $i = 0$

1st cycle:	$N \Rightarrow T$	$m \Rightarrow N$
2nd cycle:	$T \Rightarrow T$	

If $i = 1$

1st cycle:	$N \Rightarrow T$	$m \Rightarrow N$
2nd cycle:	$*T \Rightarrow T$	

Parameter Stack effect:

If $i = 0$ N T -- m N

If $i = 1$ N T -- m *N

Data Memory Access

m = >T, Invert

m = >T, Invert



Description:

If i = 0 Loads memory data into TOP. Original contents of NEXT are left unchanged.

If i = 1 Loads memory data into TOP, inverting data. Original contents of NEXT are left unchanged.

Number of cycles: 2

Processor operations:

If i = 0 1st cycle: N => T m => N
 2nd cycle: N => T T => N

If i = 1 1st cycle: N => T m => N
 2nd cycle: *N => T T => N

Parameter Stack effect:

If i = 0 N T -- N m

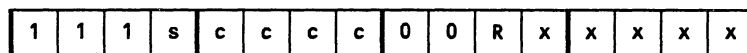
If i = 1 N T -- N *m

N-op-m

Data Memory Access

~~m-op-N => T~~

N-op-m
m-op-N => T



Description:

Loads TOP with results of ALU operation cccc between memory data and contents of NEXT. Original contents of NEXT are left unchanged.

Number of cycles:

2

Processor operations:

1st cycle: N => T

m => N

2nd cycle: T => N

T-op-N => T

Parameter Stack effect:



Data Memory Access

$\{N = > Pstack\}, m = > N$

$\{N = > Pstack\}, m = > N$



Description:

If $p = 0$ Loads memory data into NEXT. Original contents of TOP are left unchanged.

If $p = 1$ Pushes NEXT onto stack. Loads memory data into NEXT. Original contents of TOP are left unchanged.

Number of cycles: 2

Processor operations:

If $p = 0$ 1st cycle: $m \Rightarrow N$
 2nd cycle: no operation

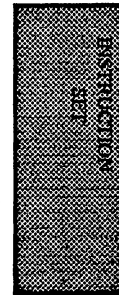
if $p = 1$ 1st cycle: $N \Rightarrow Pstack$ $m \Rightarrow N$
 2nd cycle: no operation

Parameter Stack effect:

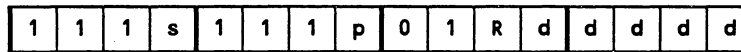
If $p = 0$ $NT -- mT$

If $p = 1$ $NT -- NmT$

Data Memory Access {N=>Pstack}, m=>N, d=>T



{N=>Pstack}, m=>N, d=>T



Description:

If $p = 0$ Loads memory data into NEXT, and short literal value dddd into TOP.

If $p = 1$ Pushes NEXT onto stack, loads memory data into NEXT, and short literal value dddd into TOP.

Number of cycles: 2

Processor operations:

If $p = 0$ 1st cycle: $m \Rightarrow N$ $dddd \Rightarrow T$
 2nd cycle: no operation

If $p = 1$ 1st cycle: $N \Rightarrow Pstack$ $m \Rightarrow N$ $dddd \Rightarrow T$
 2nd cycle: no operation

Parameter Stack effect:

If $p = 0$ N T -- m dddd

If $p = 1$ N T -- N m dddd

Data Memory Access

{N=>Pstack}, m=>N, T-op-d=>T



Description:

- If p = 0** Loads memory data into NEXT. Loads TOP with results of ALU operation aaa between the contents of TOP and short literal dddd.
- If p = 1** Pushes NEXT onto stack. Loads memory data into NEXT. Loads TOP with results of ALU operation aaa between the contents of TOP and short literal dddd.

Number of cycles: 2

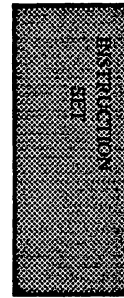
Processor operations:

- If p = 0** 1st cycle: m => N T-op-ddddd => T
 2nd cycle: no operation
- If p = 1** 1st cycle: N => Pstack m => N T-op-ddddd => T-1
 2nd cycle: no operation

Parameter Stack effect:

- If p = 0** N T -- m (dddd-op-T) — T-op-ddddd
- If p = 1** N T -- N (m dddd-op-T) — T-op-ddddd

Data Memory Access



$N \Rightarrow (T), N \Rightarrow T, \text{Invert}, \text{Pstack} \Rightarrow N$

1	1	1	s	0	0	0	i	1	0	R	x	x	x	x	x
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Description:

- If $i = 0$** Writes data in NEXT to location addressed by TOP. Copies NEXT into TOP, pops stack into NEXT.
- If $i = 1$** Writes data in NEXT to location addressed by TOP. Copies NEXT into TOP, inverting contents, then pops stack into NEXT.

Number of cycles: 2

Processor operations:

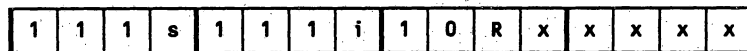
If $i = 0$	1st cycle:	$N \Rightarrow (T)$	$N \Rightarrow T$	$\text{Pstack} \Rightarrow N$
	2nd cycle:	$T \Rightarrow T$		
If $i = 1$	1st cycle:	$N \Rightarrow (T)$	$N \Rightarrow T$	$\text{Pstack} \Rightarrow N$
	2nd cycle:	$*T \Rightarrow T$		

Parameter Stack effect:

If $i = 0$	$N \ T \ \text{--} \ N$
If $i = 1$	$N \ T \ \text{--} \ *N$

Data Memory Access $N = > (T)$, $Pstack = > N, T$, Invert

$N = > (T)$, $Pstack = > N, T$, Invert



Description:

If $i = 0$ Writes contents of NEXT to location addressed by TOP.
 Pops new values into TOP and NEXT.

If $i = 1$ Writes contents of NEXT to location addressed by TOP.
 Pops new values into TOP and NEXT. Inverts new contents
 of TOP.

Number of cycles: 2

Processor operations:

If $i = 0$ 1st cycle: $N \Rightarrow (T)$ $N \Rightarrow T$ $Pstack \Rightarrow N$
 2nd cycle: $N \Rightarrow T$ $Pstack \Rightarrow N$

If $i = 1$ 1st cycle: $N \Rightarrow (T)$ $N \Rightarrow T$ $Pstack \Rightarrow N$
 2nd cycle: $*N \Rightarrow T$ $Pstack \Rightarrow N$

Parameter Stack effect:

If $i = 0$ S N T -- S

If $i = 1$ S N T -- *S

Data Memory Access

N-op-m
~~m-op-N~~ => T, Pstack => N

N-op-m
~~m-op-N~~ => T, Pstack => N

1	1	1	s	c	c	c	c	1	0	R	x	x	x	x	x
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Description: Loads TOP with results of ALU operation cccc between memory data and contents of N. Pops stack into N.

Number of cycles: 2

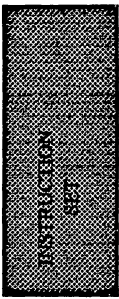
Processor operations:

1st cycle: N => T m => N

2nd cycle: T-op-N => T Pstack => N

Parameter Stack effect:

~~NT - m-op-N~~ *N-op-m*



Data Memory Access

$N = > (T), \{Pstack = > N\}$

$N = > (T), \{Pstack = > N\}$

1	1	1	s	0	0	0	p	1	1	R	x	x	x	x	x
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Description:

If $p = 0$ Writes contents of NEXT to memory location addressed by TOP. Original contents of TOP (address) are left unchanged.

If $p = 1$ Writes contents of NEXT to memory location addressed by TOP. Original contents of TOP (address) are left unchanged. Stack is popped into NEXT.

Number of cycles: 2

Processor operations:

If $p = 0$ 1st cycle: $N \Rightarrow (T)$
 2nd cycle: no operation

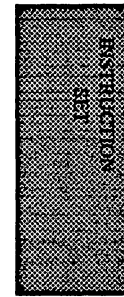
If $p = 1$ 1st cycle: $N \Rightarrow (T)$ $Pstack \Rightarrow N$
 2nd cycle: no operation

Parameter Stack effect:

If $p = 0$ $N T \text{ -- } N T$

If $p = 1$ $N T \text{ -- } T$

Data Memory Access $N = > (T), d = > T, \{Pstack = > N\}$



$N = > (T), d = > T, \{Pstack = > N\}$



Description:

If $p = 0$ Writes contents of NEXT to location addressed by TOP.
 Loads short literal ddddd into TOP.

If $p = 1$ Writes contents of NEXT to location addressed by TOP.
 Loads short literal ddddd into TOP. Pops stack into NEXT.

Number of cycles: 2

Processor operations:

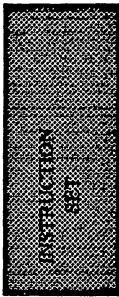
If $p = 0$ 1st cycle: $N \Rightarrow (T)$ ddddd $\Rightarrow T$
 2nd cycle: no operation

If $p = 1$ 1st cycle: $N \Rightarrow (T)$ Pstack $\Rightarrow N$ ddddd $\Rightarrow T$
 2nd cycle: no operation

Parameter Stack effect:

If $p = 0$ N T -- N ddddd

If $p = 1$ N T -- ddddd



Data Memory Access

Top-d

Top-d

$N => (T), \bar{d}\text{-op-T} => T$

$N => (T), \bar{d}\text{-op-T} => T$



Description:

- If p = 0** Writes contents of NEXT to location addressed by TOP. Loads TOP with results of ALU operation aaa between short literal ddddd and contents of TOP.
- If p = 1** Writes contents of NEXT to location addressed by TOP. Loads TOP with results of ALU operation aaa between short literal ddddd and contents of TOP. Pops stack into NEXT.

Number of cycles: 2

Processor operations:

- If p = 0**
 - 1st cycle: $N => (T)$ $dddd\text{-op-T} => T$
 - 2nd cycle: no operation
- If p = 1**
 - 1st cycle: $N => (T)$ Pstack \Rightarrow N $dddd\text{-op-T} => T$
 - 2nd cycle: no operation

Parameter Stack effects:

- If p = 0** N T -- N $dddd\text{-op-T}$ *Top-d* $Top\text{-op-ddddd}$
- If p = 1** N T -- $dddd\text{-op-T}$ ✓

7.13 Undefined opcodes

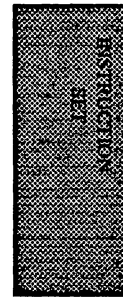
The following bit patterns are reserved for future use and should not be used for opcodes:

User Space:

1	1	0	0	x	x	x	x	x	1	x	x	x	x	x	x
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Long Literal:

1	1	0	1	x	x	x	x	x	1	x	x	x	x	x	x
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---



See table
16 in RTX 2000
Instruction set
reference
memo.

STEP MATH
FUNCTIONS

CHAPTER 8

STEP MATH FUNCTIONS

8 Step Math Functions

The Harris RTX 2000 Series Microcontrollers all include a unique and powerful set of instructions known as **Step Math Instructions**. These instructions allow the RTX microcontrollers to perform certain math operations much more quickly than would be possible without them.

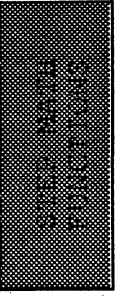
8.1 Introduction

Step math operations include signed and unsigned multiplication, unsigned division, integer square root, bit reversal and cyclic redundancy checks. They also expand the RTX processors' ability to perform logical rotation operations.

In order to achieve this increase in efficiency, the processor operates differently than when performing ordinary math. To explain this in simplified terms, intuitive mnemonics will be used here because of the number of operations that can happen in a single cycle. Forth descriptions are used only where doing so clarifies the operation. In general, it is best to consider step math operations as Forth primitives.

8.1.1 Step Math Using The RTX 2000

All of the step math functions listed above can be performed on the RTX 2000 Microcontroller. However, because of the hardware multiplier which is incorporated on-chip with this product, a special set of single cycle instructions is used to perform multiplication in the place of step math operations. See Section 6.3.1 for more detailed information about the on-chip hardware multiplier.

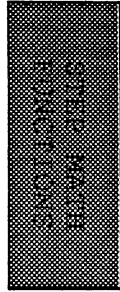


8.1.2 Step Math Using The RTX 2001A

On the RTX 2001A, step math operations are used to perform all of the functions listed in the Introduction.

8.1.3 Step Math Using The RTX 2010

All of the step math functions listed can be performed on the RTX 2010 Microcontroller. However, because this product provides the hardware Multiplier/Accumulator, Barrel Shifter, and other Floating Point Support on-chip, special instructions are used to perform some math operations in place of step math operations. See Section 6.3.2 for more detailed information about the on-chip hardware math support for the RTX 2010.



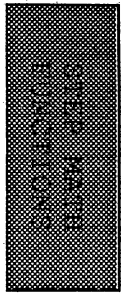
8.2 Data Flow in Step Math

Figure 8.1 shows the data flow diagram for all step math operations. Even though the hardware to perform step math is always present, much of it is inactive when not performing step math and therefore it is not emphasized outside of discussions on step math.

Note that the ALU is followed by a shifter. This allows an ALU operation and a shift to be performed in a single cycle without passing the data through the ALU twice.

Step math operations also use two special purpose registers (**MD** and **SR**) and one pseudo register (**SQ**) in their operations. There are also dedicated shift blocks and logical OR blocks used with the **MD** and **SR** registers so that data in them does not have to pass through the ALU to be updated.

The result of this architecture is that the equivalent of five ALU operations can be performed in a single cycle, and the cycles required to transfer data for these ALU operations are eliminated also.



8.3 17-Bit Math

Many of the step math operations treat the TOP register as a 17-bit wide register to accomplish their tasks. Correspondingly, the ALU is extended to 17 bits for these operations by the 17th-bit adder. Since the 17-bit result is sometimes shifted left one bit, an 18th bit is also needed to store the shifted bit. The 17th and 18th bits are held in bits zero and one of the Configuration Register (CR), and consequently change only at the end of a cycle on the rising edge of PCLK.

These bits are sometimes referred to in the data sheets as the carry (CY) and complex carry (CCY) bits, respectively, but in the context of step math, this may be misleading nomenclature. In this case, these bits are more accurately thought of as an extension of the TOP register, and will be referred to here only as CR0 and CR1.

There are values referred to as CY and CCY in other sections in this manual, which under certain conditions are clocked into CR0 and CR1 at the end of a cycle, though these values are not necessarily true carry bits. When this is the case, CR0 and CR1 may contain the CY or CCY result of the previous math or step math instruction. In this chapter, these values are sometimes referred to as W16 and W17. In working with step math it is essential to remember that CY and CCY do not exactly indicate the contents of CR0 and CR1.

which?
CR0/CR1
from
other
operations

Note that CR0 only changes when performing an ALU or shift operation, and CR1 only changes during step math operations that include an arithmetic ALU operation. Also note that CR0 and CR1 are sometimes referred to as CQ and CCQ, respectively, in other sections of this manual.

why
don't we
get rid
of all
this
confusion?

also, C16

The whole business of having 2 names for the same flip-flop is confusing to me.

8.4 The Step Math Instruction Format

All step math operations have the same format, which is similar to that of ordinary math operations. The op code for step math instructions is divided into groups of micro opcodes, each of which has a specific effect on the instruction. The general format for step math instructions is shown in Figure 8.2

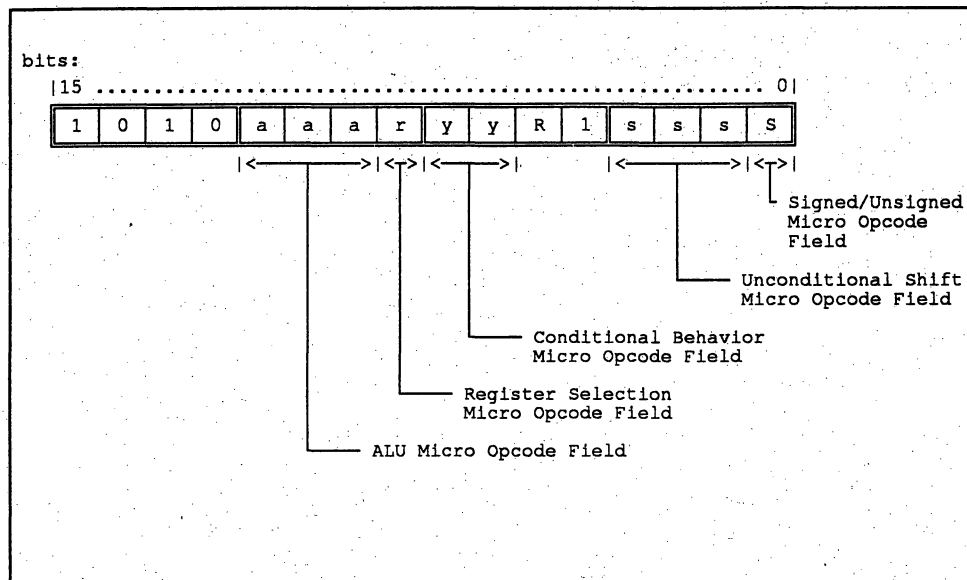


FIGURE 8.2: STEP MATH INSTRUCTION FORMAT

As is evident from the step math instruction format, there are ten bits, not including the Return bit, that determine the step math operation, which implies that there are 1024 possible step math operations. While this is true, not all of these operations are useful.

8.4.1 ALU Micro Opcode Field (aaa)

Step math operations usually include a conditional ALU operation between the TOP register and another register, either the MD or the SQ pseudo register. Step math operations may also include an ALU No-Operation or a conditional ALU load from the MD or SQ register.

TABLE 8.1: STEP MATH ALU FUNCTIONS

aaa	Function
000	No ALU operation
001	YES = 1 => TOP and REG --> TOP
010	YES = 1 => TOP - REG --> TOP
011	YES = 1 => TOP or REG --> TOP
100	YES = 1 => TOP + REG --> TOP
101	YES = 1 => TOP xor REG --> TOP
110	YES = 1 => REG - TOP --> TOP
111	YES = 1 => REG --> TOP

Whether the conditional ALU operation occurs depends on a pseudo variable, called "YES". If YES is true (1), the conditional ALU operation will be performed; if YES is false (0), the contents of TOP will be preserved, though in either case the contents of TOP may be shifted by an unconditional shift operation. The procedure for determining YES will be explained shortly.

The 18-bit result of the conditional ALU operation, as determined by YES, results in a value called "W," as shown in Figure 8.1. This value is then shifted to determine the value shifted into CR1, CR0, TOP and NEXT. If the ALU operation is not arithmetic (i.e. + or -), W17 and W16 are the value stored in CR1 and CR0 respectively.

Conditional ALU operations during step math are summarized below. References to "REG" indicate the MD or the SQ register as determined by the "r" bit (bit 8), as described in Section 8.4.2 and shown in Table 8.2.

if not arithmetic, these bits have no meaning. Always 0?

in Table 8.1

8.4.2 Register Selection Micro Opcode Field (r)

The register selection micro opcode field (bit 8) determines whether the input to the ALU operation will be the MD or the SQ register. If $r = 0$, the input will be the MD register; if $r = 1$, the input will be the SQ register.

The SQ register is actually a pseudo register: there is not a unique register associated with it. When reading the SQ register, the value obtained is the contents of the MD register shifted left one bit, and then logically OR'ed with the contents of the SR register.

← zero shifted in?

Also, the most significant bit of MD (bit 15) is fed into the 17th-bit adder. As we shall see, this allows the RTX processors to calculate the square root of an integer without using Newton's method. Also, writing data to the SQ register (ASIC Bus address 5) has the effect of multiplying the data by 256 and placing it into the MD register. Applications for this procedure is useful include calculation of some cyclic redundancy checks.

The "r" bit has another function in step math. If $r = 1$, i.e. the SQ register is selected, the data in MD and SR will be modified at the end of the cycle. The data in MD will conditionally be replaced with MD logically OR'ed with the data in SR, and the data in SR will be shifted right one bit.

what goes into high bit?

The condition that determines whether this happens is the same condition that determines whether an ALU operation will be performed (YES). This behavior is useful in both square root and bit reversal operations.

put this where you just say "conditionally" in old text otherwise, it is uncertain whether condition applies to SR as well

The behavior of MD and SR as determined by "r" is summarized in Table 8.2.

TABLE 8.2: MD AND SR OPERATION

r	YES	MD	SR	TOP
0	0	MD	SR	TOP (shift)
0	1	MD	SR	TOP (alu op) MD (shift)
1	0	MD	SR / 2	TOP (shift)
1	1	MD or SR	SR / 2	TOP (alu op) SQ (shift)

8.4.3 Conditional Behavior Micro Opcode Field (yy)

The conditional behavior micro opcode field (yy, bits 6 and 7) determines the value of the pseudo variable "YES."

In general, YES is determined by the logical combination of one or more bits in registers or by the carry out bit from the 17-bit ALU operation. The bits involved are the carry out bit (C16), bits zero and three of the opcode (Instruction Register bits IR0 and IR3), CR1, bit zero of the TOP Register (T0), and bit zero of the NEXT Register (N0).

Another factor that may affect the result is whether the ALU operation is arithmetic or logical. The behavior of YES is summarized in Table 8.3.

TABLE 8.3: BEHAVIOR OF YES

yy	YES
00	IF ARITHMETIC THEN C16 ELSE IRO
01	(IF ARITHMETIC THEN C16 ELSE IRO) or CR1
10	IF IR3 = 0 THEN T0 ELSE N0
11	T0 xor N0

So far, little has been said about the carry out bits, C15 and C16. These are outputs of the 16-bit ALU and the 17th-bit adder.

When addition is performed, the carry bits are set if the result of the addition is too large to place in the available number of bits. When subtraction is performed, however, the carry bit represents an inverted borrow bit. In this case, the carry bit is cleared if the result of the subtraction is negative.

In step math, the carry bits are used primarily in operations that involve a conditional subtraction, namely division and square roots. In these cases, the carry bit is set if the subtraction was successful, which causes the result of the subtraction to replace the original value in TOP. Both of these cases also shift the result left one bit, which is why an extra bit (CR1) is required, since it always contains the 17th bit of the result of the most recent arithmetic step math operation.



8.4.4 Subroutine Return Micro Opcode Field (R)

The operation of the subroutine return micro opcode field (bit 5) is exactly the same as for other RTX instructions.

If this bit equals one, a subroutine return is executed along with the instruction. If it equals zero, the next sequential instruction is executed.

8.4.5 Unconditional Shift Micro Opcode Field (sss)

Every step math operation may include an unconditional shift operation, (bits 1, 2, and 3) which is performed on W. Since the shift is performed in a separate section from the ALU, the shift may occur in the same cycle as the ALU operation.

Shift operations occurring during step math differ from those occurring during ordinary math operations. In particular, the sources of carry in bits to the shifts may be different, and may come from YES and CR1.

Shifts may operate on TOP, NEXT or both. The various shift operations are summarized with Forth mnemonics in Table 8.4.

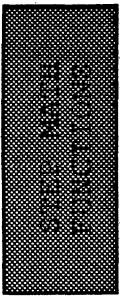
TABLE 8.4: STEP MATH SHIFT OPERATIONS

sss	NAME	CR1	CR0	T15	Tn	T0	N15	Nn	N0
000	NONE	W17	W16	W15	Wn	W0	N15	Nn	N0
001	2*'	W16	W15	W14	Wn-1	YES	N15	Nn	N0
010	c2/'	W17	W17	W16	Wn+1	W1	N15	Nn	N0
011	2/'	W17	W17	YES	Wn+1	W1	N15	Nn	N0
100	N2*'	W17	W16	W15	Wn	W0	N14	Nn-1	YES
101	D2*'	W16	W15	W14	Wn-1	N15	N14	Nn-1	YES
110	cD2/'	W17	W17	W16	Wn+1	W1	W0	Nn+1	N1
111	D2/'	W17	W17	YES	Wn+1	W1	W0	Nn+1	N1

Note that bits W0 through W17 are the result of the conditional ALU operation as determined by YES.

There are several important exceptions to Table 8.4. First, the value clocked into CR1 for opcode A057 (hex) is W17 instead of W16 as indicated by the table. The second special case occurs when the shift operation is cD2/' and the "S" bit (bit zero of the opcode) is 1, as it is in the case of signed multiply step instructions. For these instructions, the value of the bit shifted into T15 may differ from W16.

How?



8.4.6 Signed/Unsigned Micro Opcode Field (S)

The signed/unsigned micro opcode field (bit 0) has a double purpose. FirstMwit determines whether right shifts and additions are treated as signed or unsigned during multiply steps; and second, it is used with yy = 01 to allow manual control of conditional logical ALU operations. This allows unconditional 17-bit shifts, for example. When used with arithmetic ALU operations, the "S" bit affects the inputs to the 17th-bit adder.

If S = 0 (unsigned), the adder operates on CR0 and a zero bit. This is because the zero is treated as the 17th bit of the MD register. Since MD is considered to be unsigned, its 17th bit is always a zero.

If S = 1 (signed), the adder will operate on CR0 and MD15. In this case, the sign bit of MD is extended into its 17th bit. If the operation is subtraction, one of the inputs to the adder will be inverted, as will be discussed below. The "S" bit also affects right shifts during signed multiplies, to determine the value shifted into TOP.

8.5 Operation of the 17th-Bit Adder

The 17th-bit adder is an extension of the ALU for addition and subtraction operations. It can be used for subtraction as well as addition because its inputs may be inverted depending on the operation.

When performing addition, the carry out indicates that the result is negative, or has overflowed, as is the case with the ALU.

When performing subtraction, the carry indicates an inverted borrow, i.e. a carry out indicates that the subtraction of the 17th bit did not require a borrow, and that the result is non-negative. This is also the same as a carry out during subtraction for the ALU.

The operation of the 17th-bit adder is summarized in Table 8.5.

TABLE 8.5: 17th-BIT ADDER OPERATION

T	Y	CI	Z	CO
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

8.6 Interrupting Step Math Operations

In general step math operations may be interrupted as long as the interrupt handlers are well behaved; i.e. they save and restore any shared resources they may use. This is in agreement with good interrupt handler design for any processor.

The shared resources that affect step math operations are CR bits 0 and 1, MD and SR. Any handler that affects these should make sure they are in the same state upon returning from the interrupt that they were in when the interrupt occurred.

One exception is the signed step multiply operation. If it is interrupted, there is a probability that the result will be incorrect, though this depends on the values being multiplied. This is because the original signs of TOP, NEXT and MD are stored for the duration of the signed step multiply sequence. If the sequence is interrupted, these values will be lost. Also note that using streamed mode for signed multiplication is not sufficient to prevent interrupts, as this does not prevent the last step (which is not streamed) from being interrupted.

8.7 Some Useful Opcodes

It may help to look at some specific step math operations to get a feel for how they are used before considering specific cases. Table 8.6 provides a list of useful step math opcodes with their Forth mnemonics and a brief description. The following sections describe these operations in more detail.

TABLE 8.6: SOME USEFUL STEP MATH OPCODES

OPCODE	FORTH	DESCRIPTION
A012	2*'	17 Bit left shift
A09E	RDR	Rotate TOP:NEXT right
A096	RTR	Rotate TOP right
A89D	**'	Signed multiply steps 1-15
A49D	**"	Signed multiply step 16
A89C	U**'	Unsigned multiply steps 1-16
A49C	U**"	Mixed sign multiply step 16
A894	BU**'	Byte unsigned multiply steps 1-8
A494	BU**"	Byte mixed sign multiply step 8
A41A	U/1'	Unsigned divide step 1
A45A	U/'	Unsigned divide steps 2-15
A458	U/'"	Unsigned divide step 16
A418	U/1'"	Alternate unsigned divide step 16
A412	BU/'	Byte unsigned divide steps 1-8
A51A	S1'	Square root step 1
A55A	S'	Square root steps 2-15
A558	S'"	Square root step 16
A512	BS1'	Byte Square root step 1
A552	BS'	Byte square root steps 2-8
A196	R'	Bit reversal step
AADE	C'	CRC step



8.8 Step Multiplication

In applications in which multiplication is needed, the RTX 2001A Microcontroller uses the step multiplication operations which are described in the following sections. These step math multiplication operations would not normally be performed on the RTX 2000 Microcontroller due to the increased speed available through its on-chip hardware multiplier.

8.8.1 Signed Step Multiplication

The primitive signed step multiplication operation operates on two signed numbers. One of these numbers, the multiplier, is initially in **NEXT**, and the other, the multiplicand, is in the **MD** register. The product is a signed, double precision number on the stack.

Prior to performing signed step multiplication, both **CR0** and **CR1** should be initialized to zeros and **TOP** should be initialized to zero. **TOP** may optionally be initialized to a signed number which will be added to the product.

If step signed multiplication is interrupted, the result may be incorrect.

8.8.2 Signed Step Multiplication Op Codes

```

*'
Signed Multiplication Steps 1 Through 15
A89D = 1010 1000 1001 1101
aar =      1000      ==> ALU OP = TOP + MD --> TOP
yy  =          10      ==> YES   = NØ
sssS =          1101 ==> shift  = cD2/' signed

```

```

If NØ = 1
    TOP + MD --> TOP
TOP:NEXT / 2 --> TOP:NEXT

```

```

**
Signed Multiplication Step 16
A49D = 1010 0100 1001 1101
aar =      0100      ==> ALU OP = TOP - MD --> TOP
yy  =          10      ==> YES   = NØ
sssS =          1101 ==> shift  = cD2/' signed

```

```

If NØ = 1
    TOP - MD --> TOP
TOP:NEXT / 2 --> TOP:NEXT

```

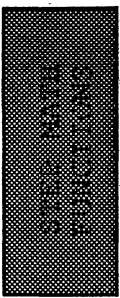
This step differs from ***** in that MD is conditionally subtracted from TOP instead of added. This is because the value originally in NEXT is in two's complement form and as such the most significant bit represents a negative multiple of a power of 2, i.e. 2^{15} .

8.8.3 Signed Step Multiplication Example Program

```

HEX
: M* ( n n -- d )
  CR@ DUP 2* 0< 10 AND OR >R      \ Save state of int disable bit
  CR@ 10 OR CR!                    \ Disable interrupts
  MD!                               \ Set up MD
  0                                 \ Set up TOP
  0 +                               \ Clear CR0
  2**                               \ Clear CR1
  ** ** ** ** ** ** ** ** ** ** ** **    \ Perform the multiplication
  ** ** ** ** ** ** ** ** ** ** ** **
R> CR! ;                          \ Restore int disable bit

```



8.8.4 Mixed Sign Multiplication Type A

Type A mixed sign multiplication is similar to signed multiplication except that the value originally in NEXT is treated as a 16-bit unsigned integer. Because of this, the last step is the same as the first 15 steps, which are the same as for signed multiplication.

This type of multiplication is useful for calculating the partial product of a multiple precision multiplication.

Because all the multiplication steps are the same, the operation can be streamed to disable interrupts, which reduces the overhead for this type of multiplication.

8.8.5 Type A Mixed Sign Multiplication Example Program:

```
: MA* ( u n -- d )
MD!           \ Set up MD
0             \ Set up TOP
0 +          \ Clear CRO
2**         \ Clear CR1
F OF( ** ;   \ Perform the multiplication
```

8.8.6 Unsigned Multiplication

Unsigned multiplication is similar to signed multiplication except that the multiplier and multiplicand are both treated as unsigned 16-bit values. Also, 8-bit unsigned multiplication is supported. This allows a faster multiplication in the event the multiplier and multiplicand are both 8-bit values.

Since the multiplicand (MD) is always positive in unsigned multiplication, CR0 is added to a zero bit instead of the most significant bit of MD in the the 17th-bit adder. Otherwise unsigned multiplication is similar to signed multiplication.

8.8.7 Unsigned Multiplication Op Codes

U*

Unsigned Multiplication Steps 1 Through 16

A89C = 1010 1000 1001 1100

aaar = 1000 ==> ALU OP = TOP + MD --> TOP

YY = 10 ==> YES = N0

ssss = 1100 ==> shift = cD2/' unsigned

If N0 = 1

TOP + MD --> TOP

TOP:NEXT / 2 --> TOP:NEXT

BU*

8-Bit Unsigned multiplication steps 1 through 8

A894 = 1010 1000 1001 0100

aaar = 1000 ==> ALU OP = TOP + MD --> TOP

YY = 10 ==> YES = T0

ssss = 0100 ==> shift = c2/' unsigned

If T0 = 1

TOP + MD --> TOP

TOP / 2 --> TOP



8.8.8 Unsigned Multiplication Example Program:

```

: UM* ( u u -- du )
  MD!           \ Set up MD
  0             \ Set up TOP
  0 +           \ Clear CR0
  2**          \ Clear CR1
  U** U** U** U** U** U** U** U** \ Perform the multiplication
  U** U** U** U** U** U** U** U** ;

```

8.8.9 8-Bit Unsigned Multiplication Example Program:

```

: BUM* ( bu bu -- u )
  SQ!           \ Set up multiplicand
  0 +           \ Clear CR0
  2** 2/        \ Clear CR1
  8U** 8U** 8U** 8U** \ Perform the multiplication
  8U** 8U** 8U** 8U** ;

```

8.8.10 Mixed Sign Multiplication Type B

Type B mixed sign multiplication is similar to unsigned multiplication except that the value originally in NEXT is treated as a 16-bit signed integer. Because of this, the last step is different (i.e. MD is subtracted from instead of added to TOP) from the first 15 steps, which are the same as for unsigned multiplication. This is for the same reason that a different step is required in signed multiplication.

This type of multiplication is useful for calculating the partial product of a multiple precision multiplication, and is also supported for 8-bit operands.

8.8.11 Mixed Sign Multiplication Type B Op Codes

U*

Unsigned Multiplication Step 16

A49C = 1010 0100 1001 1100

aaar = 0100 ==> ALU OP = TOP - MD --> TOP

yy = 10 ==> YES = NØ

ssss = 1100 ==> shift = cD2/' unsigned

If NØ = 1

TOP - MD --> TOP

TOP:NEXT / 2 --> TOP:NEXT

BU*

8-Bit Unsigned Multiplication Step 8

A494 = 1010 0100 1001 0100

aaar = 0100 ==> ALU OP = TOP - MD --> TOP

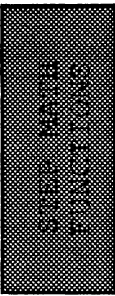
yy = 10 ==> YES = TØ

ssss = 0100 ==> shift = c2/' unsigned

If TØ = 1

TOP - MD --> TOP

TOP / 2 --> TOP



8.8.12 Type B Mixed Sign Multiplication Example Programs:

```

: MB* ( n u -- d )
MD!           \ Set up MD
0            \ Set up TOP
0 +          \ Clear CR0
2**         \ Clear CR1
U** U** U** U** U** U** U** U** U** \ Perform the multiplication
U** U** U** U** U** U** U** U** ;

```

8.8.13 8-Bit Unsigned Multiplication Example Program:

```

: BMB* ( b bu -- n )
SQ!           \ Set up multiplicand
255 AND      \ Make 8-bit negative numbers
0 +          \ Clear CR0
2** 2/       \ Clear CR1
8U** 8U** 8U** 8U** \ Perform the multiplication
8U** 8U** 8U** 8U** ;

```

8.9 Step Division

Step division is performed in 16 steps using long division, and signed division is not supported.

A 32-bit dividend in **TOP** and **NEXT** is divided by a 16-bit divisor in **MD**, leaving a 16-bit remainder in **TOP** and a 16-bit quotient in **NEXT**.

Because the quotient is limited to 16 bits, not all dividends and divisors yield a quotient small enough to be represented. In these cases, the result is invalid. The only way to check the validity of a result is to multiply the quotient by the divisor and add the remainder. Also, division by zero yields an invalid result.

Two versions of the division program follow. The standard version (see Section 8.9.1) tests the value of **CR1** for steps 2 through 16. The alternate version (see Section 8.9.2) does not.

It can be proven that **CR1** will always be zero for any division that yields a valid result, so the standard and alternate versions both work the same as long as the result is valid. In the event the result is invalid, however, their results may differ.

There is also an 8-bit version of step division, which is faster than the 16-bit version. In the 8-bit version, a 16-bit unsigned number in **TOP** is divided by an 8-bit unsigned number in **MD**, leaving an 8-bit result in **TOP**. The same restrictions that apply to 16-bit division also apply to 8-bit division.

BUT,
we can
check for
overflow
in many (all?)
cases before
the division





8.9.1 Step Division Opcodes

U/1'

Unsigned Divide Step 1

A41A = 1010 0100 0001 1010

aaar = 0100 ==> ALU OP = TOP - MD --> TOP

yy = 00 ==> YES = C16

ssss = 1010 ==> shift = D2*' unsigned

If C16 = 1 (TOP - MD >= 0)

TOP - MD --> TOP

TOP:NEXT * 2 --> TOP:NEXT

YES --> NØ

U/'

Unsigned Divide Steps 2 through 15

A45A = 1010 0100 0101 1010

aaar = 0100 ==> ALU OP = TOP - MD --> TOP

yy = 01 ==> YES = C16 + CR1

ssss = 1010 ==> shift = D2*' unsigned

If C16 + CR1 = 1 (TOP - MD >= 0)

TOP - MD --> TOP

TOP:NEXT * 2 --> TOP:NEXT

YES --> NØ

This step differs from U/1' in that the conditional subtraction will be performed also if the previous subtraction also produced a result with 1 in the most significant bit (now shifted into CR1) even though the subtraction was successfully performed.

It can be proven that this only happens when the dividend is too large to produce a 16-bit quotient when divided by the current divisor.

U/"
Unsigned Divide Step 16
A458 = 1010 0100 0101 1010
aaar = 0100 ==> ALU OP = TOP - MD --> TOP
yy = 01 ==> YES = C16 + CR1
ssss = 1010 ==> shift = N2*' unsigned

If C16 + CR1 = 1 (TOP - MD >= 0)
TOP - MD --> TOP
NEXT * 2 --> NEXT
YES --> NØ

This step differs from U/' in that only NEXT is shifted. This allows the correct remainder to be left in TOP.

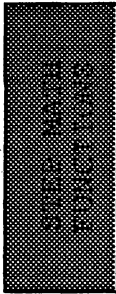
U/1"
Alternate Unsigned Divide Step 16
A418 = 1010 0100 0101 1010
aaar = 0100 ==> ALU OP = TOP - MD --> TOP
yy = 00 ==> YES = C16
ssss = 1010 ==> shift = N2*' unsigned

If C16 = 1 (TOP - MD >= 0)
TOP - MD --> TOP
NEXT * 2 --> NEXT
YES --> NØ

This step differs from U/1' in that only NEXT is shifted. This allows the correct remainder to be left in TOP.

BU/'
8-Bit Unsigned Divide Steps 1 through 8
A412 = 1010 0100 0001 0010
aaar = 0100 ==> ALU OP = TOP - MD --> TOP
yy = 00 ==> YES = C16
ssss = 0010 ==> shift = 2*' unsigned

If C16 = 1 (TOP - MD >= 0)
TOP - MD --> TOP
TOP * 2 --> TOP
YES --> TØ



8.9.2 Standard Division Program Example

This version of the division program tests the value of CR1 for steps 2 through 16.

```

: UM/MOD ( ud u -- ur uq )
MD!           \ Set up divisor
D2*          \ Clear CR0, 17 divide steps not needed
U/1'        \ Step 1
U/' U/' U/' U/' U/' \ Steps 2 - 15
U/' U/' U/' U/' U/'
U/' U/' U/' U/'
U/"         \ Step 16
SWAP ;      \ Put quotient, remainder in right places

```

8.9.3 Alternate Division Program Example

This version of the division program does not test the value of CR1 for steps 2 through 16.

```

: UM/MOD ( ud u -- ur uq )
MD!           \ Set up divisor
D2*          \ Clear CR0, 17 divide steps not needed
U/1' U/1' U/1' U/1' \ Steps 1 - 15
U/1' U/1' U/1' U/1'
U/1' U/1' U/1' U/1'
U/1' U/1' U/1'
U/1"        \ Step 16
SWAP ;      \ Put quotient, remainder in right places

```

8.9.4 8-Bit Division Program Example

This version of the division program also does not test the value of CR1 for steps 1 through 8.

```

: 8U/ ( ud u -- ur uq )
SQ!         \ Set up divisor
2*         \ Clear CR0
8U/' 8U/' 8U/' 8U/' \ Divide it
8U/' 8U/' 8U/' 8U/'
FF AND ;   \ Discard remainder * 2

```

8.10 Step Square Root

The RTX 2000 Series of Microcontrollers implement an exact algorithm (vs. an approximate algorithm such as Newton's method) for finding the square root of an integer.

In the event the input value is not a perfect square, a root and a remainder are found. The remainder is similar to the remainder of division: if the remainder is added to the square of the root, the original input value is found. It is possible for the remainder to exceed the 16 bits of the TOP register, though the 17th bit will always be contained in CR0.

Prior to executing the square root steps, the 32-bit input value is placed into TOP and NEXT, MD is cleared, and a value of 8000 hex is placed in SR.

Square root steps are similar to division steps but subtract the SQ register instead of MD from TOP. They also use CR1 to determine whether to perform the conditional subtraction. This is needed for square root steps because the value being subtracted changes from step to step.

Step Square Root Algorithm

The pseudo-division square root algorithm is similar to restoring long division. That is, it consists of repeatedly subtracting a subtrahend from the input value. If the result of the subtraction is negative, however, the value prior to the subtraction is restored. This can also be thought of as a conditional subtraction. The difference from long division lies mostly in the value of the subtrahend, which changes from step to step depending on which of the previous subtractions were successful. A subtraction is successful if it leaves a non-negative intermediate result. If a subtraction is successful, a one is shifted into the least significant bit of the result; otherwise a zero is shifted in. To get a feeling for how the subtrahends are generated, let us consider the 8 bit square root of a 16 bit number using hardware similar to that in the RTX. The bits of the root are represented by r_n with r_7 being the most significant bit. The trial subtrahends are shown aligned with the square from which they are to be subtracted.



TABLE 8.7: SQUARE ROOT TRIAL SUBTRAHENDS

S _f	S _a	S _d	S _c	S _b	S _a	S _b	S _c	S _d	S _e	S _f	S _g	S _h	S _i	S _j	S _k	S _l
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	r ₇	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	r ₇	r ₆	0	1	0	0	0	0	0	0	0	0	0	0	0
0	0	0	r ₇	r ₆	r ₅	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	r ₇	r ₆	r ₅	r ₄	0	1	0	0	0	0	0	0	0
0	0	0	0	0	r ₇	r ₆	r ₅	r ₄	r ₃	0	1	0	0	0	0	0
0	0	0	0	0	0	r ₇	r ₆	r ₅	r ₄	r ₃	r ₂	0	1	0	0	0
0	0	0	0	0	0	0	0	r ₇	r ₆	r ₅	r ₄	r ₃	r ₂	r ₁	0	1

As is evident from the table, each trial subtrahend is dependent only on the results of the trial subtractions executed prior to the current trial. The values in this table are also the same as the first 8 subtrahends for the 16-bit square root, and it should be apparent how the remaining 8 subtrahends are generated. These subtrahends are generated directly by MD, SR and the surrounding circuitry.

Unless you are a mathematician, it is probably not obvious why subtracting these values should give you the square root of a number. The remainder of this section may help you to understand how this algorithm works.

Consider two numbers, r (an N-bit unsigned integer) and s (an unsigned integer with 2N bits). Let s be the square of r. One can represent r as a polynomial of powers of 2:

$$r = r_n 2^n + r_{n-1} 2^{n-1} + \dots + r_1 2 + r_0,$$

where $n=N-1$ and r_0 through r_n are either zero or one. Since s is the square of r, s can also be expressed as a polynomial of powers of 2:

$$\begin{aligned} s &= rr \\ &= [r_n 2^{2n} + r_n r_{n-1} 2^{2n-1} + \dots + r_n r_0 2^n] + \\ &\quad [r_{n-1} r_n 2^{2n-1} + r_{n-1} 2^{2n-2} + \dots + r_{n-1} r_0 2^{n-1}] + \\ &\quad \dots + \\ &\quad [r_0 r_n 2^n + r_0 r_{n-1} 2^{n-1} + \dots + r_0] \end{aligned}$$

Notice that since r_n can only equal 1 or 0, $r_n r_n = r_n$. It is useful to visualize the terms of s as values in a square array, as shown in Table 8.8:

TABLE 8.8: TERMS OF s AS VALUES IN A SQUARE ARRAY

	$r_n 2^n$	$r_{n-1} 2^{n-1}$...	$r_1 2$	r_0
$r_n 2^n$	$r_n 2^{2n}$	$r_n r_{n-1} 2^{2n-1}$...	$r_n r_1 2^{n+1}$	$r_n r_0 2^n$
$r_{n-1} 2^{n-1}$	$r_n r_{n-1} 2^{2n-1}$	$r_{n-1} 2^{2n-2}$...	$r_{n-1} r_1 2^n$	$r_{n-1} r_0 2^{n-1}$
...
$r_1 2$	$r_n r_1 2^{n+1}$	$r_{n-1} r_1 2^n$...	$r_1 2^2$	$r_1 r_0 2$
r_0	$r_n r_0 2^n$	$r_{n-1} r_0 2^{n-1}$...	$r_1 r_0 2$	r_0

Notice that all the perfect squares are on the diagonal running from the upper left corner to the lower right corner, and the rest of the array is symmetrical about the diagonal, and each term in the lower left half of the array has an identical term in the upper right half. Like terms can be combined by adding the terms in the upper right half to those in the lower left half. Each row in the new triangular matrix may then be used to form a trial subtrahend, t_0 through t_n , that may be used for finding the square root of s :

$$\begin{aligned}
 t_n &= r_n 2^{2n} \\
 t_{n-1} &= r_n r_{n-1} 2^{2n} + r_{n-1} 2^{2(n-1)} \\
 t_{n-2} &= r_n r_{n-2} 2^{2n-2} + r_{n-1} r_{n-2} 2^{2n-1} + r_{n-2} 2^{2(n-2)} \\
 &\dots \\
 t_1 &= r_n r_1 2^{n+2} + r_{n-1} r_1 2^{n+1} + \dots + r_2 r_1 2^4 + r_1 2^2 \\
 t_0 &= r_n r_0 2^{n+1} + r_{n-1} r_0 2^n + \dots + r_1 r_0 2^2 + r_0
 \end{aligned}$$

Notice that t_n depends only on r_n , t_{n-1} depends only on r_n and r_{n-1} , t_{n-2} depends only on r_n , r_{n-1} and r_{n-2} , and so on.

The technique for using these trial subtrahends for finding the root of s is straightforward. If r_n is assumed to be 1, and t_n can be subtracted from s , then r_n is indeed 1. If, however, the result of the subtraction is negative, r_n is 0, and s must be restored to its value prior to the subtraction. Once r_n is known, it can be used to find r_{n-1} by the same method, and so on to r_0 . This is exactly what happens when subtracting the trial subtrahends in Table 8.7.

For further illustration we can find the subtrahends for the 8 bit square root in table 1. In this case the root is:

$$r = r_7 2^7 + r_6 2^6 + r_5 2^5 + r_4 2^4 + r_3 2^3 + r_2 2^2 + r_1 2 + r_0 .$$

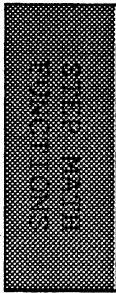
The square is:

$$\begin{aligned} s &= rr \\ &= [r_7 2^{14} + r_7 r_6 2^{13} + \dots + r_7 r_0 2^7] + \\ &\quad [r_6 r_7 2^{13} + r_6 2^{12} + \dots + r_6 r_0 2^6] + \\ &\quad \dots + \\ &\quad [r_0 r_7 2^7 + r_0 r_6 2^6 + \dots + r_0] . \end{aligned}$$

By combining like terms and grouping into subtrahends we obtain:

$$\begin{aligned} t_7 &= r_7 2^{14} \\ t_6 &= r_7 r_6 2^{14} + r_6 2^{12} \\ t_5 &= r_7 r_5 2^{13} + r_6 r_5 2^{12} + r_5 2^{10} \\ &\dots \\ t_0 &= r_7 r_0 2^8 + r_6 r_0 2^7 + \dots + r_1 r_0 2^2 + r_0 , \end{aligned}$$

which is identical to the description of the subtrahends in Table 8.7.



An Example:

For a specific example, let us find the 8 bit square root of a 16 bit number. Let the input value equal 8163H (hex), i.e. 100000101100011 binary.

```

      10110101
    )100000101100011
    -0100000000000000 1st subtraction
      0100000101100011 Successful
    -0101000000000000 2nd Subtraction
      1111000101100011 Not Successful
      0100000101100011 Restore Previous Value
    -0010010000000000 3rd Subtraction
      0001110101100011 Successful
    -0001010100000000 4th Subtraction
      0000100001100011 Successful
    -0000101101000000 5th Subtraction
      1111110100100011 Not Successful
      0000100001100011 Restore Previous Value
    -0000010110010000 6th Subtraction
      0000001011010011 Successful
    -0000001011010100 7th Subtraction
      1111111111111111 Not Successful
      0000001011010011 Restore Previous Value
    -0000000101101001 8th Subtraction
      0000000101101010 Successful, Also Remainder

```

Note that the remainder is a 9 bit value though the root is an 8 bit value. This is possible because the difference between two successive perfect squares is:

$$(n+1)^2 - n^2 = (n^2 + 2n + 1) - n^2 = 2n + 1 .$$

Therefore the largest possible remainder, which is one less than this difference is $2n$, which requires one more bit to represent than n does.

8.10.1 Step Square Root Opcodes

S1'

Square Root Step 1

A51A = 1010 0101 0001 1010

aaar = 0101 ==> ALU OP = TOP - SQ --> TOP

yy = 00 ==> YES = C16

sssS = 1010 ==> SHIFT = D2*' UNSIGNED

If C16 = 1 (TOP - SQ >= 0)

TOP - SQ --> TOP

TOP:NEXT * 2 --> TOP:NEXT

YES --> NO

MD + SR --> MD

SR / 2 -- SR

S'

Square Root Steps 2 Through 15

A55A = 1010 0101 0101 1010

aaar = 0101 ==> ALU OP = TOP - SQ --> TOP

yy = 01 ==> YES = C16 + CR1

sssS = 1010 ==> SHIFT = D2*' UNSIGNED

If C16 + CR1 = 1 (TOP - SQ >= 0)

TOP - SQ --> TOP

TOP:NEXT * 2 --> TOP:NEXT

YES --> NO

MD + SR --> MD

SR / 2 -- SR

S' differs from S1' in that CR1 is considered to determine YES. This is necessary because sometimes a successful subtraction will result in the most significant bit being 1. Once this bit is shifted into CR1, it cannot be subtracted.

S''

Square Root Step 16

A558 = 1010 0101 0101 1000

aaar = 0101 ==> ALU OP = TOP - SQ --> TOP

yy = 01 ==> YES = C16 + CR1

sssS = 1000 ==> SHIFT = N2*' UNSIGNED

If C16 + CR1 = 1 ; TOP - SQ >= 0

TOP - SQ --> TOP

NEXT * 2 --> NEXT

YES --> NO

MD + SR --> MD

SR / 2 -- SR



S'' differs from S' in that only NEXT is shifted instead of both NEXT and TOP. This is done so that the remainder is correct.

BS1'

8-Bit Square Root Step 1

A512 = 1010 0101 0001 0010

aaar = 0101 ==> ALU OP = TOP - SQ --> TOP

YY = 00 ==> YES = C16

ssss = 0010 ==> SHIFT = 2*' UNSIGNED

If C16 = 1 (TOP - SQ >= 0)

TOP - SQ --> TOP

TOP * 2 --> TOP

YES --> T0

MD + SR --> MD

SR / 2 -- SR

BS'

8-Bit Square Root Steps 2 Through 8

A552 = 1010 0101 0101 0010

aaar = 0101 ==> ALU OP = TOP - SQ --> TOP

YY = 01 ==> YES = C16 + CR1

ssss = 0010 ==> SHIFT = 2*' UNSIGNED

If C16 + CR1 = 1 (TOP - SQ >= 0)

TOP - SQ --> TOP

TOP * 2 --> TOP

YES --> T0

MD + SR --> MD

SR / 2 -- SR

8.10.2 Square Root Program Example

HEX

```
: ROOT ( du - uroot uremainder )
8000 SR!           \ Set up SR
0000 MD!           \ Set up MD
D2*                \ Get lined up for 1st subtraction
S1'                \ 1st step
S' S' S' S' S' S' \ Steps 2 - 15
S' S' S' S' S' S'
S" ;               \ Last step
```

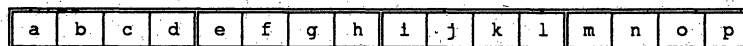
8.10.3 8-Bit Square Root Program Example

HEX

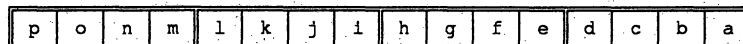
```
: BROOT ( word -- byte )
8000 SR!           \ Set up SR
0000 MD!           \ Set up MD
2*                 \ Line up for first subtraction
BS1'               \ Step 1
BS' BS' BS' BS' BS' BS' BS' \ Steps 2-8
FF AND ;           \ Discard remainder * 2
```

8.11 Step Bit Reversal

Bit reversal reverses the order in which bits appear in a word. For example:



would become:



Using step math allows bit reversal of a 16-bit word in 23 cycles compared to 48 cycles without step math. Bit reversal is useful for calculating address during FFT operations and also is needed to calculate certain types of CRC's.

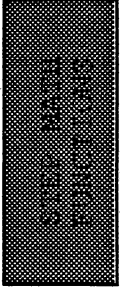
8.11.1 Step Bit Reversal Opcodes

```
R'  
Bit Reversal Step  
A196 = 1010 0001 1001 0110  
aaar =      0001      ==> ALU OP = NO OP, OPERATE ON SQ  
uu   =      10       ==> YES   = TO  
sssS =      0110    ==> SHIFT = 2/' UNSIGNED
```

```
If TO = 1  
  MD or SR --> MD  
SR / 2 --> SR  
TOP / 2 --> TOP
```

8.11.2 Step Bit Reversal Example Program

```
HEX  
  
: BIT-REVERSE ( u -- u' )  
 8000 SR!           \ Set up SR  
 0000 MD!           \ Set up MD  
 R' R' R' R' R' R' R' \ Put bit reversed version of TOP into MD  
 R' R' R' R' R' R' R'  
 R' R'  
 DROP              \ Discard garbage in TOP  
 MD@ ;             \ Retrieve result
```



8.12 Step Cyclic Redundancy Check (CRC)

The cyclic redundancy check (CRC) is used for identification and error checking of blocks of data, in much the same way a checksum is used.

The advantage of the CRC is that more errors can be detected with a CRC than by a checksum. For example, a 16-bit CRC can detect all errors in a 16-bit frame of a stream of data. There are several de facto standard data transfer protocols, including XMODEM, X.25 and Kermit, that use variations of CRCs for error checking.

The basis for calculating a CRC of a stream of bits is to perform a modulo-2 long division of the stream (multiplied by an appropriate power of two) by an irreducible modulo-2 polynomial. The quotient is discarded, and the remainder is the CRC.

In modulo-2 subtraction, there is no carrying or borrowing from bit to bit, so subtraction is the same as a bitwise logical exclusive OR function. The polynomial is a value that cannot be evenly divided modulo-2 by another polynomial, much like a prime number.

Most 16-bit CRCs are calculated with the polynomial $x^{16} + x^{12} + x^5 + 1$, which can also be expressed as 10001000000100001.

For example, to calculate the XMODEM style CRC of the ASCII character "T" (54 hex), perform the following long division:

```
10001000000100001)010101000000000000000000
                    10001000000100001
                    10000000010000100
                    10001000000100001
                    10000101001010000
                    10001000000100001
                    0001101001110001 = 1A71 hex
```

To calculate the CRC of a stream of characters, simply XOR the CRC's of each character with the CRC of the preceding characters. This can also be accomplished by replacing the 16 right-hand bits of the above dividend with the previous CRC.

One useful aspect of the CRC calculation is that it can be implemented in hardware with 16 shift registers and 3 exclusive OR gates. To see how this is done, think of the above dividend as being in a 16-bit shift register.

When a 1 is shifted out of the register, the contents of the register are exclusive OR'ed with the 16 least significant bits of the polynomial.

When a zero bit is shifted out, no exclusive OR'ing takes place. After eight shifts, the register holds the CRC.

Since many serial data protocols transmit the least significant bit of the data first, many CRC's are calculated on the bit reversed image of the transmitted character so that the CRC may be calculated in the simple hardware noted above. In such cases, the resulting CRC is also bit reversed and must be un-reversed, though this only needs to be done once for each packet of data. Examples of protocols that use this type of CRC are X.25 and Kermit.

Other variations on the CRC are to use a non-zero initial value for the CRC, usually FFFF hex. The other variation is to exclusive OR the CRC with a non-zero value, also usually FFFF hex, before transmitting it. An example of this is the X.25 protocol.

The CRC in RTX step math is implemented such that bit reversed CRC's are generated directly. The result does not need to be bit reversed in these cases because the shifting is done to the right instead of to the left. The polynomial, however, must be bit-reversed before exclusive OR'ing it with the data stream. This causes no performance loss, however, because it is usually a constant. For implementing CRC's that do not use bit-reversal, such as those used by XMODEM, the data and CRC's must be bit reversed.

8.12.1 Step CRC Opcodes

```
C'
Cyclic Redundancy Check Step
AADE = 1010 1010 1101 1110
aaar =      1010      ==> ALU OP = TOP xor MD --> TOP
yy    =      11      ==> YES   = T0 xor N0
sssS =      1110    ==> SHIFT = 2/' UNSIGNED

If (T0 xor N0) = 1
TOP xor MD --> TOP
TOP:NEXT / 2 --> TOP:NEXT
```

8.12.2 Step CRC Example Program

```
HEX
0811 CONSTANT POLY          \ Bit reversed generator polynomial
: CRC ( crc byte -- crc' )
  SWAP                      \ Rearrange data and original CRC
  POLY MD!                  \ Set up MD
  C' C' C' C' C' C' C' C' \ Calculate new CRC
  NIP ;                     \ Discard partial quotient
```

To use this program to calculate a CRC for Kermit or CRC-CCITT, use an initial CRC value of zero. To use this program to calculate a CRC for X.25 protocol, use an initial CRC value of FFFF hex and invert the resulting CRC before transmitting it with the packet.

The CRC for two characters can be calculated at once by using 16 C' steps instead of eight, while placing the first character received in the least significant byte of the data word.

8.12.3 XMODEM CRC Example Program

```
HEX
: CRCK ( crc byte -- crc' )
  BIT-REVERSE              \ Reverse data byte
  SQ! MD@                  \ Left justify data byte
  SWAP BIT-REVERSE SWAP   \ Reverse original CRC
  CRC                      \ Calculate new CRC
  BIT-REVERSE ;           \ Un-reverse new CRC
```

Note that in calculating the CRC for a long string of characters, the CRC only needs to be reversed and un-reversed at the beginning and end of the string of characters.

If the initial CRC is zero, as it is in XMODEM, it only needs to be un-reversed at the end of the string of characters. Every character, however, needs to be reversed.

8.13 Step Math Reference

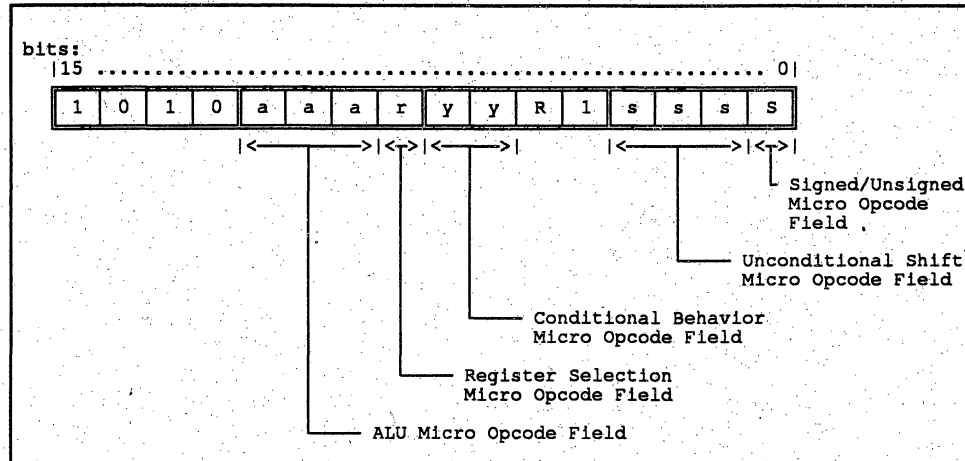


FIGURE 8.2: STEP MATH INSTRUCTION FORMAT

TABLE 8.8: STEP MATH ALU FUNCTIONS

aaa	Function
000	No ALU operation
001	YES = 1 => TOP and REG --> TOP
010	YES = 1 => TOP - REG --> TOP
011	YES = 1 => TOP or REG --> TOP
100	YES = 1 => TOP + REG --> TOP
101	YES = 1 => TOP xor REG --> TOP
110	YES = 1 => REG - TOP --> TOP
111	YES = 1 => REG --> TOP

TABLE 8.9: MD AND SR OPERATION

r	YES	MD	SR	TOP
0	0	MD	SR	TOP (shift)
0	1	MD	SR	TOP (alu op) MD (shift)
1	0	MD	SR / 2	TOP (shift)
1	1	MD or SR	SR / 2	TOP (alu op) SQ (shift)

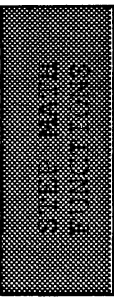


TABLE 8.10: BEHAVIOR OF YES

yy	YES
00	IF ARITHMETIC THEN C016 ELSE IRO
01	(IF ARITHMETIC THEN C016 ELSE IRO) or CR1
10	IF IR3 = 0 THEN TO ELSE NØ
11	TO xor NØ

TABLE 8.11: STEP MATH SHIFT OPERATIONS

sss	NAME	CR1	CR0	T15	Tn	TO	N15	Nn	NØ
000	NONE	W17	W16	W15	Wn	W0	N15	Nn	NØ
001	2**	W16	W15	W14	Wn-1	YES	N15	Nn	NØ
010	c2/'	W17	W17	W16	Wn+1	W1	N15	Nn	NØ
011	2/'	W17	W17	YES	Wn+1	W1	N15	Nn	NØ
100	N2**	W17	W16	W15	Wn	W0	N14	Nn-1	YES
101	D2**	W16	W15	W14	Wn-1	N15	N14	Nn-1	YES
110	cD2/'	W17	W17	W16	Wn+1	W1	W0	Nn+1	N1
111	D2/'	W17	W17	YES	Wn+1	W1	W0	Nn+1	N1

TABLE 8.12: SOME USEFUL STEP MATH OPCODES

OPCODE	FORTH	DESCRIPTION
A012	2**	17 Bit left shift
A09E	RDR	Rotate TOP:NEXT right
A096	RTR	Rotate TOP right
A89D	**	Signed multiply steps 1-15
A49D	**	Signed multiply step 16
A89C	U**	Unsigned multiply steps 1-16
A49C	U**	Mixed sign multiply step 16
A894	BU**	Byte unsigned multiply steps 1-8
A494	BU**	Byte mixed sign multiply step 8
A41A	U/1'	Unsigned divide step 1
A45A	U/'	Unsigned divide steps 2-15
A458	U/"	Unsigned divide step 16
A418	U/1"	Alternate unsigned divide step 16
A412	BU/'	Byte unsigned divide steps 1-8
A51A	S1'	Square root step 1
A55A	S'	Square root steps 2-15
A558	S"	Square root step 16
A512	BS1'	Byte Square root step 1
A552	BS'	Byte square root steps 2-8
A196	R'	Bit reversal step
AADE	C'	CRC step

USING THE RTX2000 STACK CONTROLLER

Two of the powerful features of the RTX2000 are its two on-chip stacks: the parameter stack and the return stack. Because these stacks can operate simultaneously with the memory data bus and the ASIC data bus, they increase the performance of the RTX2000 by maximizing the quantity of data that can move in one processor cycle (refer to the RTX2000 block diagram). In addition to increasing the speed of the RTX2000, the stacks can interrupt the processor in the event of a stack overflow or underflow.

PARAMETER STACK STRUCTURE

The parameter stack is a 256 word by 16 bit stack that is used for storing data and addresses for arithmetic and logical operations. The top of the parameter stack is the TOP register in the RTX processor core, the next stack location is the NEXT register in the RTX core, and the remaining 256 locations are in on-chip RAM controlled by an RTX stack controller.

RETURN STACK STRUCTURE

The return stack is a 256 word by 21 bit stack that is used for storing return addresses for subroutine calls and loop counters for certain operations. The top of the return stack is in the processor core index register (I) and the remaining locations are in on-chip RAM controlled by an RTX stack controller. You may have noticed that the return stack has 21 bits per word. This is partly because the RTX2000 supports a twenty bit address. However, since op codes are on word boundaries, only nineteen bits are required to define a return address for a subroutine call. These are bits one through nineteen, where bit zero is the least significant bit and bit twenty is the most significant bit. Bit zero is used to determine whether the address is for a return from an interrupt (1) or a return from a subroutine call (0). This allows interrupts to be enabled when returning from an interrupt. Bit 20 is used to store the data page register select bit (DPRSEL) which is used to determine which memory page is used for data fetch and store operations. When the return stack is used for storing loop variables (>R and R>), only bits zero through fifteen are used for data. Bits sixteen through twenty are loaded with the content of the code page register (CPR) during an interrupt, subroutine call or >R execution.

RTX INTERRUPT CONTROLLERS

The RTX2000 interrupt controller circuitry consists of two identical sections, one for the parameter stack and one for the return stack. Each section has two user accessible 8-bit registers that control the operation of its stack: a stack pointer register (SPR) and a stack limit register (SLR). Since the data path on the RTX2000 is sixteen bits wide, the SPR for the parameter stack and the SPR for the return stack are concatenated to form a single sixteen-bit SPR for the purpose of accessing them by the processor. Similarly, the SLR for the parameter stack is concatenated with the SLR for the return stack. The SPR may be read as well as written to, but the SLR is a write-only register. In both cases, data bits 0 through 7 apply to the parameter stack and data bits 8 through 15 apply to the return stack.

STACK OPERATION

Both stacks have 256 words, numbered from zero to 255. When data is pushed onto the stack, it is written to the location above that pointed to by the SPR, and the SPR is incremented (this is equivalent to a pre-increment). When data is popped off the stack, it is read from the location pointed to by the SPR, and then the SPR is decremented (post-decrement). When the RTX2000 is reset, the SPRs are reset to zero, so the first location written to will be location one. If an attempt is made to push data onto a stack beyond word 255, the stack will wrap around and start pushing at location zero. Similarly, if an attempt is made to pop data off the stack below location zero, the stack will wrap around to location 255. For this reason the parameter stack can be used as a recirculating 256 word buffer. Also, because the stacks are circular, they must be managed carefully to prevent overflows and underflows from producing incorrect results. The stack controller helps to accomplish this by generating overflow and underflow interrupts.

STACK ERROR INTERRUPTS

The RTX stack controller requests an overflow interrupt anytime the value of the SPR is greater than the SLR. Since the SPR can never exceed 255, the stack controller cannot generate an overflow interrupt with the SLR at its reset value of 255. Therefore the user should set the SLR to a value less than 255, which also provides an overflow buffer that allows for the extra cycles required to acknowledge the interrupt. Another point about using overflow interrupts is that a push-pop sequence can generate an interrupt request that goes away before it is serviced. The result of this is that the interrupt controller causes the processor to execute code pointed to by the "no interrupt" vector. This is referred to as a phantom interrupt. Therefore, this location should always be initialized to a valid code sequence, even if it is only a "no-op, return" sequence.

The RTX stack controller handles underflows similarly to overflows. The controller requests an underflow interrupt when data is popped off location one. Also, if data is pushed onto the stack until it wraps around, an underflow interrupt will be generated when data is pushed onto location zero. Because popping data off location one causes an interrupt and location one is normally the first stack location used, the interrupt generated when data is popped off location one would occur even though the data is valid. Therefore, whenever underflow interrupts are used, the first location used should be location two. This is most easily done by pushing dummy data to location one at start up. The "DUP" instruction will accomplish this for the parameter stack, and the sequence "DUP >R" will accomplish this for the return stack. Because pushing data onto location zero will request an underflow interrupt, the validity of the stack can be managed with just the underflow interrupt, while the overflow interrupt is useful for managing a virtual stack greater than 256 words or multitasking. The underflow interrupt only method also allows the greatest number of words to be used (254 out of 256) while using stack error interrupts. If stack error interrupts are not used, the entire stack is usable.

The underflow interrupt also can exist for a short enough time to produce phantom interrupts. This can happen if a pop from location one is immediately followed by a push to location one. For this reason, it is necessary to provide valid code at the memory location pointed to by the no-interrupt vector.

USING THE STACK POINTER REGISTER

It is possible to perform stack manipulations by reading and writing the SPR. Because of the intricate interaction between the SPR and the stacks, great care must be exercised to prevent the processor from getting lost.

READING THE STACK POINTER REGISTER

The stack pointers are both read in a single cycle by executing a SPR@ instruction. The parameter stack pointer is retrieved in the least significant byte and the return stack is retrieved in the most significant byte. The value obtained is not exactly the value of the stack pointers prior to the SPR@ cycle, however. The outputs of the stack pointers normally pass through incrementers on their way to being read. In addition to this, pushes and pops on either stack during the SPR@ cycle cause the stack pointers to change in the middle of the cycle. This can cause additional increments and decrements to the values obtained and should, therefore, be avoided. For example, returning from a subroutine during a SPR@ cycle causes the value for the return stack to be decremented instead of incremented.

Reading the SPR usually results in a push on to the parameter stack. This causes the parameter stack pointer to be incremented in the middle of the @SPR cycle. The output of the incremented stack pointer then passes through the incrementer before it is read. The result is that the value obtained is the value of the parameter stack prior to executing the SPR@ plus two. Because of the number of steps that must be performed on the value from the stack pointer in the last half of the SPR@ cycle, it is recommended that the stack pointers be read without performing any stack operations during the fetch cycle. This can be done as follows:

1. Do not combine a SPR@ with a return from subroutine.
2. Combine the SPR@ with an arithmetic or logical function to prevent the result from pushing the previous top of stack down on the stack. An effective way to obtain the values of the stack pointers follows:

```

HEX    \ BASE 16 FOR THIS DISCUSSION
-102  \ CORRECTION FACTOR :
      \ RETURN STACK: 1
      \ PARAMETER STACK: 2
SPR@ + \ SINGLE OP CODE: NO STACK OPERATIONS
NOP ; \ IF A RETURN FOLLOWS, MAKE IT A SEPARATE OP CODE

```

WRITING TO THE STACK POINTER REGISTERS

Since stack operations do not change the value stored in the SPR, it may be written to by a simple SPR! instruction. There is one thing to watch out for, however. If a SPR! is combined with a return from subroutine, the value popped off the return stack into the I register will be determined by the value in the SPR prior to the execution of the SPR! instruction.