RCX internals
(Revised February 24)

1   Introduction

This document collects together various pieces of information about the hardware in the LegoMind-storms RCX brick. The RCX uses a Hitachi H8/3292 microcontroller (a member of the H8/3297 family) running at 16MHz. We use the term "RCX" to refer to both the microcontroller and the other parts of the RCX hardware.

2   Registers

The RCX has eight 16-bit general-purpose registers (`r0-r7`). These can be used both to address memory and to hold data. As data registers, they can also be viewed as 16 eight-bit registers (`r0h`, `r0l`, ...). The RCX uses `r7` as the *stack pointer* (`sp`). It also has a 16-bit program counter (`pc`) and an eight-bit condition-code register (`ccr`).

The C ABI for the RCX uses `r6` as the frame (or base) pointer. The first three function arguments are passed in `r0`, `r1`, and `r2`; additional arguments are passed on the stack, and function results are returned in `r0`. Registers `r4` and `r5` are callee-save.

The condition code register is organized as follows:

Bit	Name	Description
0	C	Carry flag
1	V	Overflow flag
2	Z	Zero flag
3	N	Negative flag
4		User bit
5	H	Half-carry flag
6		User bit
7	I	Interrupt mask bit

Most arithmetic instructions affect the `ccr`, as do data move instructions. There are also instructions for performing logical operations on the `ccr`.

3 Instructions

The RCX is largely a load-store architecture. Most arithmetic instructions work on registers, although it supports some bit operations that work on absolute addresses.[1]

The RCX processor supports a number of addressing modes:

Mode	Description
rn	register
@rn	register indirect
@(d:16,rn)	register indirect with displacement
@rn+	register indirect with post-increment
@-rn	register indirect with pre-decrement
@a:8	8-bit absolute (use 0xff as high bits)
@a:16	16-bit absolute
#x:8	8-bit immediate
#x:16	16-bit immediate
@(d:16,pc)	PC relative
@@a:8	Memory indirect

Note: when addressing words, the least bit of the address is ignored (*i.e.*, regarded as 0).

4 Memory

The RCX supports byte addressing with a 16-bit address space. The address space includes ROM, RAM, on-chip RAM, and device registers. These memories are mapped into a 16-bit address space as follows:

Address range	Memory type	Contents
0x0000–0x3fff	on-chip ROM	RCX executive
0x4000–0x7fff	Reserved (unmapped)	
0x8000–0xfb7f	off-chip RAM	program and data
0xfb80–0xfd7f	Reserved (unmapped)	
0xfd80–0xff7f	on-chip RAM	ROM data
0xfe00–0xff7f	on-chip RAM	initial program stack
0xff80–0xff87	Reserved (unmapped)	
0xff88–0xffff	on-chip device registers	H8/3293 device registers

5 Interrupts

The RCX hardware supports 23 distinct interrupts (listed in Table 1). This table includes the name, RAM interrupt-vector location, and short description of each interrupt. When an interrupt occurs, the RCX hardware handles it as follows:

1. The I bit (bit 7) of the ccr register is tested; if it is set, and the interrupt is not a NMI, then it is marked as pending and execution continues.

---

[1]This operations are used to manipulate the on-chip device registers that are mapped into the address space.

Table 1: RCX interrupts

Name	RAM vector	Description
NMI	0xfd92	Non Maskable Interrupt
IRQ0	0xfd94	Interrupt 0
IRQ1	0xfd96	Interrupt 1
IRQ2	0xfd98	Interrupt 2
ICIA	0xfd9a	16 bit Timer – Input Capture A
ICIB	0xfd9c	16 bit Timer – Input Capture B
ICIC	0xfd9e	16 bit Timer – Input Capture C
ICID	0xfda0	16 bit Timer – Input Capture D
OCIA	0xfda2	16 bit Timer – Output Compare A
OCIB	0xfda4	16 bit Timer – Output Compare B
FOVI	0xfda6	16 bit Timer – Overflow
CMI0A	0xfda8	8 bit Timer 0 – Compare Match A
CMI0B	0xfdaa	8 bit Timer 0 – Compare Match B
OVI0	0xfdac	8 bit Timer 0 – Overflow
CMI1A	0xfdae	8 bit Timer 1 – Compare Match A
CMI1B	0xfdb0	8 bit Timer 1 – Compare Match B
OVI1	0xfdb2	8 bit Timer 1 – Overflow
ERI	0xfdb4	Serial Receive Error
RXI	0xfdb6	Serial Receive End
TXI	0xfdb8	Serial TDR Empty
TEI	0xfdba	Serial TSR Empty
ADI	0xfdbc	A/D Conversion End
WOVF	0xfdbe	Watchdog Timer Overflow

Low addresses

Stack grows down

r6 →

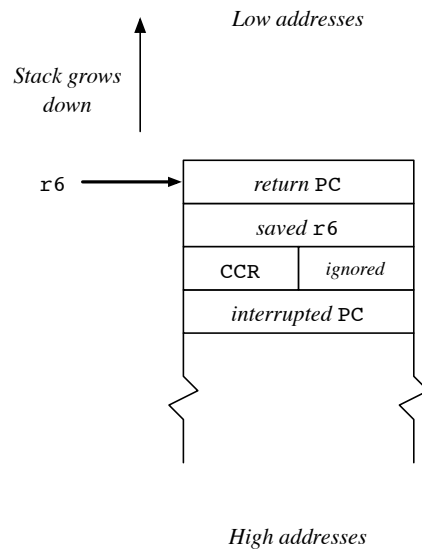| *return* PC	
saved r6	
CCR	*ignored*
interrupted PC	

High addresses

Figure 1: Stack layout upon entry in interrupt handler.

2. If the I bit is clear, or the interrupt is an NMI, then the hardware pushes the ccr register (plus a byte of padding), and the PC.

3. The I bit of the ccr is set.

4. The PC is loaded from the ROM interrupt vector, which contains the address of the dispatch code, which is also in ROM, for the particular interrupt.

5. The dispatch code saves r6 on the stack, loads the address of the interrupt handler from the RAM interrupt vector, and then does a jsr to the handler.

6. The interrupt handler runs.

7. Upon return, it restores r6 and does a rte, which restores the ccr and pc to their values at the time of the interrupt.

Figure 1 gives the state of the stack upon entry to the interrupt handler. Note that the RCX is a *big-endian* machine, so the saved ccr register will be at an even address.

To install a handler for an interrupt, one need only store the handler's address in the interrupt's RAM vector location. The ROM also contains a default handler, which just returns to the dispatch code, at address 0x046a.

Appendix — The H8/300L instruction set

```
add.b    #x:8,rd        8-bit addition
add.b    rs,rd          8-bit addition
add.w    rs,rd          16-bit addition
```

4

`adds`	#1,rd	16-bit increment by 1(does not affect `ccr`)
`adds`	#2,rd	16-bit increment by 2 (does not affect `ccr`)
`addx`	#x:8,rd	8-bit addition with carry
`addx`	rs,rd	8-bit addition with carry
`and`	#x:8,rd	8-bit logical and
`and`	rs,rd	8-bit logical and
`andc`	#x:8,ccr	8-bit logical and with `ccr`
`band`	#x:3,@a:8	bit and
`band`	#x:3,@rd	bit and
`band`	#x:3,rd	bit and
`bcc`	d:8	conditional branch on carry clear (also called `bhs`)
`bclr`	#x:3,@a:8	bit clear
`bclr`	#x:3,@rd	bit clear
`bclr`	#x:3,rd	bit clear
`bclr`	rn,@a:8	bit clear
`bclr`	rn,@rd	bit clear
`bclr`	rn,rd	bit clear
`bcs`	d:8	conditional branch on carry set (also called `blo`)
`beq`	d:8	conditional branch on equal
`bge`	d:8	conditional branch on greater or equal
`bgt`	d:8	conditional branch on greater than
`bhi`	d:8	conditional branch on high
`biand`	#x:3,@a:8	bit invert and
`biand`	#x:3,@rd	bit invert and
`biand`	#x:3,rd	bit invert and
`bild`	#x:3,@a:8	bit invert load
`bild`	#x:3,@rd	bit invert load
`bild`	#x:3,rd	bit invert load
`bior`	#x:3,@a:8	bit invert or
`bior`	#x:3,@rd	bit invert or
`bior`	#x:3,rd	bit invert or
`bist`	#x:3,@a:8	bit invert store
`bist`	#x:3,@rd	bit invert store
`bist`	#x:3,rd	bit invert store
`bixor`	#x:3,@a:8	bit invert exclusive or
`bixor`	#x:3,@rd	bit invert exclusive or
`bixor`	#x:3,rd	bit invert exclusive or
`bld`	#x:3,@a:8	bit load
`bld`	#x:3,@rd	bit load
`bld`	#x:3,rd	bit load
`ble`	d:8	conditional branch on less or equal
`bls`	d:8	conditional branch on low or same
`blt`	d:8	conditional branch on less than
`bmi`	d:8	conditional branch on minus
`bne`	d:8	conditional branch on not equal
`bnot`	#x:3,@a:8	bit not
`bnot`	#x:3,@rd	bit not

bnot	#x:3,rd	bit not
bnot	rn,@a:8	bit not
bnot	rn,@rd	bit not
bnot	rn,rd	bit not
bor	#x:3,@a:8	bit or
bor	#x:3,@rd	bit or
bor	#x:3,rd	bit or
bpl	d:8	conditional branch on plus
bra	d:8	branch always
brn	d:8	branch never
bset	#x:3,@a:8	bit set
bset	#x:3,@rd	bit set
bset	#x:3,rd	bit set
bset	rn,@a:8	bit set
bset	rn,@rd	bit set
bset	rn,rd	bit set
bsr	d:8	branch to subroutine
bst	#x:3,@a:8	bit store
bst	#x:3,@rd	bit store
bst	#x:3,rd	bit store
btst	#x:3,@a:8	bit test
btst	#x:3,@rd	bit test
btst	#x:3,rd	bit test
btst	rn,@a:8	bit test
btst	rn,@rd	bit test
btst	rn,rd	bit test
bvc	d:8	conditional branch on overflow clear
bvs	d:8	conditional branch on overflow set
bxor	#x:3,@a:8	bit exclusive or
bxor	#x:3,@rd	bit exclusive or
bxor	#x:3,rd	bit exclusive or
cmp.b	#x:8,rd	8-bit compare
cmp.b	rs,rd	8-bit compare
cmp.w	rs,rd	16-bit compare
daa	rd	decimal-adjust add
das	rd	decimal adjust subtract
dec	rd	8-bit decrement
divxu	rs,rd	16-bit by 8-bit unsigned division ((8+8)-bit result)
eepmov		move data to EEPROM
inc	rd	8-bit increment
jmp	@@a:8	jump
jmp	@a:16	jump
jmp	@rn	jump
jsr	@@a:8	jump to subroutine
jsr	@a:16	jump to subroutine
jsr	@rn	jump to subroutine
ldc	#x:8,ccr	load ccr

`ldc`	`rs,ccr`	load `ccr`
`mov.b`	`#x:8,rd`	8-bit load signed immediate
`mov.b`	`@(x:16,rs),rd`	8-bit load
`mov.b`	`@a:16,rd`	8-bit load
`mov.b`	`@a:8,rd`	8-bit load
`mov.b`	`@rs+,rd`	8-bit load
`mov.b`	`@rs,rd`	8-bit load
`mov.b`	`rs,@(x:16,rd)`	8-bit store
`mov.b`	`rs,@-rd`	8-bit store
`mov.b`	`rs,@a:16`	8-bit store
`mov.b`	`rs,@a:8`	8-bit store
`mov.b`	`rs,@rd`	8-bit store
`mov.b`	`rs,rd`	8-bit register-to-register move
`mov.w`	`#x:16,rd`	16-bit load immediate
`mov.w`	`@(x:16,rs),rd`	16-bit load
`mov.w`	`@a:16,rd`	16-bit load
`mov.w`	`@rs+,rd`	16-bit load (also called `pop`, when rs is `sp`)
`mov.w`	`@rs,rd`	16-bit load
`mov.w`	`rs,@(x:16,rd)`	16-bit store
`mov.w`	`rs,@-rd`	16-bit store (also called `push`, when rd is `sp`)
`mov.w`	`rs,@a:16`	16-bit store
`mov.w`	`rs,@rd`	16-bit store
`mov.w`	`rs,rd`	16-bit register-to-register move
`mulxu`	`rs,rd`	8-bit by 8-bit unsigned multiply (16-bit result)
`neg`	`rd`	8-bit 2's complement negation
`nop`		no operation
`not`	`rd`	8-bit 1's complement negation (logical not)
`or`	`#x:8,rd`	8-bit logical or
`or`	`rs,rd`	8-bit logical or
`orc`	`#x:8,ccr`	8-bit logical or with `ccr`
`rotl`	`rd`	8-bit rotate left
`rotr`	`rd`	8-bit rotate right
`rotxl`	`rd`	8-bit rotate with carry left
`rotxr`	`rd`	8-bit rotate with carry right
`rte`		return from exception
`rts`		return from subroutine
`shal`	`rd`	8-bit arithmetic left shift
`shar`	`rd`	8-bit arithmetic right shift
`shll`	`rd`	8-bit logical left shift
`shlr`	`rd`	8-bit logical right shift
`sleep`		put processor to sleep
`stc`	`ccr,rd`	8-bit store from `ccr`
`sub.b`	`rs,rd`	8-bit subtraction
`sub.w`	`rs,rd`	16-bit subtraction
`subs`	`#1,rd`	16-bit decrement by 1 (does not affect `ccr`)
`subs`	`#2,rd`	16-bit decrement by 2 (does not affect `ccr`)
`subx`	`#x:8,rd`	8-bit subtract with carry

`subx`	rs,rd	8-bit subtract with carry
`xor`	#x:8,rd	8-bit exclusive or
`xor`	rs,rd	8-bit exclusive or
`xorc`	#x:8,`ccr`	8-bit exclusive or with `ccr`