1 Introduction

This document collects together various pieces of information about the hardware in the LegoMindstorms 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:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>C</td>
<td>Carry flag</td>
</tr>
<tr>
<td>1</td>
<td>V</td>
<td>Overflow flag</td>
</tr>
<tr>
<td>2</td>
<td>Z</td>
<td>Zero flag</td>
</tr>
<tr>
<td>3</td>
<td>N</td>
<td>Negative flag</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>User bit</td>
</tr>
<tr>
<td>5</td>
<td>H</td>
<td>Half-carry flag</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>User bit</td>
</tr>
<tr>
<td>7</td>
<td>I</td>
<td>Interrupt mask bit</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rn</code></td>
<td>register</td>
</tr>
<tr>
<td><code>@rn</code></td>
<td>register indirect</td>
</tr>
<tr>
<td><code>@(d:16,rn)</code></td>
<td>register indirect with displacement</td>
</tr>
<tr>
<td><code>@rn+</code></td>
<td>register indirect with post-increment</td>
</tr>
<tr>
<td><code>@rn-</code></td>
<td>register indirect with pre-decrement</td>
</tr>
<tr>
<td><code>@a:8</code></td>
<td>8-bit absolute (use <code>0xff</code> as high bits)</td>
</tr>
<tr>
<td><code>@a:16</code></td>
<td>16-bit absolute</td>
</tr>
<tr>
<td><code>#x:8</code></td>
<td>8-bit immediate</td>
</tr>
<tr>
<td><code>#x:16</code></td>
<td>16-bit immediate</td>
</tr>
<tr>
<td><code>@(d:16,pc)</code></td>
<td>PC relative</td>
</tr>
<tr>
<td><code>@@a:8</code></td>
<td>Memory indirect</td>
</tr>
</tbody>
</table>

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:

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

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

¹These operations are used to manipulate the on-chip device registers that are mapped into the address space.
<table>
<thead>
<tr>
<th>Name</th>
<th>RAM vector</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NMI</td>
<td>0xfd92</td>
<td>Non Maskable Interrupt</td>
</tr>
<tr>
<td>IRQ0</td>
<td>0xfd94</td>
<td>Interrupt 0</td>
</tr>
<tr>
<td>IRQ1</td>
<td>0xfd96</td>
<td>Interrupt 1</td>
</tr>
<tr>
<td>IRQ2</td>
<td>0xfd98</td>
<td>Interrupt 2</td>
</tr>
<tr>
<td>ICIA</td>
<td>0xfd9a</td>
<td>16 bit Timer – Input Capture A</td>
</tr>
<tr>
<td>ICIB</td>
<td>0xfd9c</td>
<td>16 bit Timer – Input Capture B</td>
</tr>
<tr>
<td>ICIC</td>
<td>0xfd9e</td>
<td>16 bit Timer – Input Capture C</td>
</tr>
<tr>
<td>ICID</td>
<td>0xfda0</td>
<td>16 bit Timer – Input Capture D</td>
</tr>
<tr>
<td>OCIA</td>
<td>0xfda2</td>
<td>16 bit Timer – Output Compare A</td>
</tr>
<tr>
<td>OCIB</td>
<td>0xfda4</td>
<td>16 bit Timer – Output Compare B</td>
</tr>
<tr>
<td>FOVI</td>
<td>0xfda6</td>
<td>16 bit Timer – Overflow</td>
</tr>
<tr>
<td>CMI0A</td>
<td>0xfd8</td>
<td>8 bit Timer 0 – Compare Match A</td>
</tr>
<tr>
<td>CMI0B</td>
<td>0xfd9a</td>
<td>8 bit Timer 0 – Compare Match B</td>
</tr>
<tr>
<td>OVI0</td>
<td>0xfdac</td>
<td>8 bit Timer 0 – Overflow</td>
</tr>
<tr>
<td>CMI1A</td>
<td>0xfdace</td>
<td>8 bit Timer 1 – Compare Match A</td>
</tr>
<tr>
<td>CMI1B</td>
<td>0xfdbe</td>
<td>8 bit Timer 1 – Compare Match B</td>
</tr>
<tr>
<td>OVI1</td>
<td>0xfdbe</td>
<td>8 bit Timer 1 – Overflow</td>
</tr>
<tr>
<td>ERI</td>
<td>0xfd92</td>
<td>Serial Receive Error</td>
</tr>
<tr>
<td>RIX</td>
<td>0xfd6</td>
<td>Serial Receive End</td>
</tr>
<tr>
<td>TXI</td>
<td>0xfd8</td>
<td>Serial TDR Empty</td>
</tr>
<tr>
<td>TEI</td>
<td>0xfdb</td>
<td>Serial TSR Empty</td>
</tr>
<tr>
<td>ADI</td>
<td>0xfdbc</td>
<td>A/D Conversion End</td>
</tr>
<tr>
<td>WOVF</td>
<td>0xfdbe</td>
<td>Watchdog Timer Overflow</td>
</tr>
</tbody>
</table>
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
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)
bcl#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
bland #x:3,@a:8  bit invert and
bland #x:3,@rd  bit invert and
bland #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
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
bles 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  rnr,a:8  bit not
bnot  rnr,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  rnr,a:8  bit set
bset  rnr,rd  bit set
bset  rnr,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  rnr,a:8  bit test
btst  rnr,rd  bit test
btst  rnr,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
da  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
mov.b #x:8, rd
mov.b @(x:16, rs), rd
mov.b @a:16, rd
mov.b @a:8, rd
mov.b @rs+, rd
mov.b @rs, rd
mov.b rs, @(x:16, rd)
mov.b rs, @-rd
mov.b rs, @a:16
mov.b rs, @a:8
mov.b rs, @rd
mov.b rs, rd
mov.w #x:16, rd
mov.w @(x:16, rs), rd
mov.w @a:16, rd
mov.w @rs+, rd
mov.w @rs, rd
mov.w rs, @(x:16, rd)
mov.w rs, @-rd
mov.w rs, @a:16
mov.w rs, @rd
mulxu rs, rd
neg rd
not rd
or #x:8, rd
or rs, rd
orc #x:8, ccr
rotl rd
rotr rd
rotxl rd
rotxr rd
rte
rts
shal rd
shar rd
shll rd
shlr rd
sleep
stc ccr, rd
sub.b rs, rd
sub.w rs, rd
subs #1, rd
subs #2, rd
subx #x:8, rd

load ccr
8-bit load signed immediate
8-bit load
8-bit load
8-bit load
8-bit load
8-bit load
8-bit store
8-bit store
8-bit store
8-bit store
8-bit store
8-bit register-to-register move
16-bit load immediate
16-bit load
16-bit load (also called pop, when rs is sp)
16-bit load
16-bit store
16-bit store (also called push, when rd is sp)
16-bit store
16-bit store
16-bit store
16-bit register-to-register move
8-bit by 8-bit unsigned multiply (16-bit result)
8-bit 2’s complement negation
no operation
8-bit 1’s complement negation (logical not)
8-bit logical or
8-bit logical or
8-bit logical or with ccr
8-bit rotate left
8-bit rotate right
8-bit rotate with carry left
8-bit rotate with carry right
return from exception
return from subroutine
8-bit arithmetic left shift
8-bit arithmetic right shift
8-bit logical left shift
8-bit logical right shift
put processor to sleep
8-bit store from ccr
8-bit subtraction
16-bit subtraction
16-bit decrement by 1 (does not affect ccr)
16-bit decrement by 2 (does not affect ccr)
8-bit subtract with carry
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>subx</code></td>
<td>rs,rd</td>
<td>8-bit subtract with carry</td>
</tr>
<tr>
<td><code>xor</code></td>
<td>#x:8,rd</td>
<td>8-bit exclusive or</td>
</tr>
<tr>
<td><code>xor</code></td>
<td>rs,rd</td>
<td>8-bit exclusive or</td>
</tr>
<tr>
<td><code>xorc</code></td>
<td>#x:8,ccr</td>
<td>8-bit exclusive or with ccr</td>
</tr>
</tbody>
</table>