The ARC (A Risc Computer) Instruction Set

Computer Architecture and Organization: An Integrated Approach; ©2007
by M. Murdocca & V. Heuring

Opcode 00: SETHI/Branch Format

SETHI

```
|0 0|    rd   | op2 |           imm22                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
```

```
rd = imm22 << 10 (left shift 10)
```

The purpose of the SETHI instruction is to set the high-order bits of a register to a 22 bit quantity.

Branch

```
|0 0|0| cond  | op2 |           disp22                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
```

```
Eaddr = sign_extend(disp22) << 2 (note word alignment)
%pr = %pc + EAddr
```

The N,V, Z, and C bits set by a previous "cc"-type operation are tested if the condition is met a relative jump is taken by adding the distance (in word) between the branch instruction and the target address to the program counter.

ba – Branch Always

bn – Branch Never

Branch on Flags

bpos – Branch on Positive (i.e. N == 0)

bneg – Branch on Negative (i.e. N == 1)

be – Branch on Equal (i.e. Z == 1)

bne – Branch in No Equal (i.e. Z == 0)

bvc – Branch on Overflow Clear (i.e. V == 0)

bvs – Branch on Oversflow Set (i.e. V == 1)

bcc – Branch on Carry Clear (i.e. C == 0)

bcs – Branch on Carry Set (i.e. C == 1)

Unsigned Comparison Branching

bgu – Branch on Greater than unsigned (i.e. C | Z == 0)

bleu – Branch on Less than or Equal to Unsigned (i.e. C | Z == 1)

Note: Unlike the ARC, the SPARC supports bgeu (branch on greater than or equal unsigned) and blu (branch on less than unsigned) instructions which are synonyms for the bcs and bcc instructions respectively. Note that a 22 bit displacement gives you a *restricted* range of from -2^{21} to $+2^{21}-1$ words between the branch instruction and the target.

Signed Comparison Branching

bg – Branch on Greater than	ble – Branch on Less than or equal
bl – Branch on Less than	bge – Branch on Greater than or Equal

Example of signed comparison

```
      subcc %r1, %r2, %r0      ! if %r1 > %r2
      bg L1                    !    goto L1
```

Example of count-down loop from 10 to 0

```
      add %r0, 10, %r10        ! %r10 initialized to 10
Loop:                          ! loop begins here

      subcc %r10, 1, %r10      ! decrement %r10
      bg Loop                  ! loop if > 0
```

Opcode 01: Call Format (1 instruction in this class)

```
|0 1|                        disp30                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
```

```
%r15 = %pc
%pc = %pc + sign_extend(disp) << 2
```

Store the program counter in register %r15 (for eventual return) and branch to the subroutine. Like the branch instruction this is relative jump taken by adding the sign-extended displacement to the program counter but with a 30 bit displacement left shifted by 2 to enforce word-alignment, this given you a range covering the entire ARC memory.

Note: A return is performed by the `jmpl %15+4, %r0` instruction (below)

Opcode 02: Arithmetic/Logic Operations

```
|1 0|    rd   |     op3   |   rs1   |0|               |   rs2   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
```

The two operands are the contents of the two source registers `rs1` and `rs2`.

```
|1 0|    rd   |     op3   |   rs1   |1|        simm13           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
```

The two operands are source register `rs1` and the `simm13` field sign extended to a 32 bit quantity.
`op2` (below) refers to the second operand – either `rs2` or sign-extend(`simm13`)

Arithmetic Operations

add addcc

examples

```
      add r1%, r2%, r%r3       ! %r3 = %r1 + r%2 – sum contents of two registers
      add r4%, 1, %r4          ! increment %r4
```

sub subcc

examples

```
      sub %r5, %r6, %r7 ! %r7 = %r5 - %r6 - order is important
      sub %r8, 1, %r8   ! decrement %r8
```

Logical Operations

and	andcc	used to mask and test bits
andn (a and not b)	andncc	
or	orcc	used to set bits
orn (a or not b)	orncc	Note: this is NOT equivalent to nor `%r0 orn %r1` complements %r1
xor	xorcc	eXclusive OR
xnor	xnorcc	eXclusive NOR - Note: $a\, xnor\, b = \overline{a\, xor\, b}$

Shift Operations

sll (shift left logical)	rd = left-shift(rs1, op2) where op2 is number of bits to shift
shr (shift right logical)	rd = right-shift(rs1, op2)
sra (shift right arithmetic)	rd = arithmetic-right-shift(rs2, op2)

Special Return Operation/Halt Instruction

jmpl – jump and link This instruction is used as a return from subroutine call. Essentially the following operations are executed

%rd = %pc ! save %pc in destination register %rd
%pc = %rs1 + op2 ! jump to address %rs1 + op2

Example

```
      jmpl %r15,4,%r0 ! return from subroutine
```

Note: Since %r15 contains the address of call instruction itself, you must add 4 to branch to the instruction following the Call instruction.

halt this (non-standard) instruction will halt the simulator; programs should end with this.

Opcode 03: Memory Format Instructions – Load and Store

```
|1 1|    rd    |     op3    |   rs1    |0|                |    rs2   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
```

EAddr = %rs1 + %rs2

The effective address is obtained by adding the contents of %r1 and %r2

```
|1 1|    rd    |     op3    |   rs1    |1|        simm13           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
```

EAddr = %rs1 + sign_extend(simm13)

The effective address is obtained by adding the sign-extended simm3 field to register %rs1

```
ld    load word (address must be  word-aligned - i.e. end in 00)

examples

      ld [x], %r3       ! load x to %r3
      ld %r1, %r4       ! %r1 contains address of word to load
      ld %r1, 4, %r5    ! load next word
      ld [%r1 + 4], %r5 ! alternate notation

ldub  load unsigned byte
ldsb  load sign-extended byte
```

`lduh  load unsigned half-word` (address must be half-word aligned – address ends in 0)
`ldsh  load sign-extended halfword` (address must be half word aligned)

`st    store word` (note rd field used for source!)

```
examples
      st %r3, [y]       ! store %r3 at y
      st %r4, %r1       ! store %r4 at address contained in %r1
      st %r5,4,%r1      ! store %r5 at next location
      st %r5, [%r1 + 4] ! alternate notation ?

stb   store byte
sth   store halfword
```

Example: A simple ARC programming example

```
!
!  A simple ARC program to add two numbers
!
       .begin
       .org 2048
main: ld [x], %r1              ! load x into %r1
      ld [y], %r2              ! load y into %r2
      addcc %r1, %r2, %r3      ! %r3 <- %r1 + %r2
      st %r3, [z]              ! store %r3 into z
      halt                     ! halt simulator
      jmpl %r15+4, %r0         ! standard return
x:    15
y:    9
z:    0
      .end
```

Observe the following

1. `.begin` and `.end` directives must enclose the program

2. By convention code begins at address 2048; hence the `.org 2048` directive

3. The ARC is a *load and store* architecture; all arithmetic operations are register to register; the only memory reference operations are load (ld) and store (st).

4. Data flow is left to right (i.e. src -> dest) ; that is `ld [x] , %r1` loads x into %r1 while `st %r3, [z]` stores %r3 into z. The same is true for arithmetic operations: `addcc %r1, %r2, %r3` adds %r1 and %r2 into destination register %r3 (and sets the condition code registers accordingly)

5. Comments begin with !

6. There is a limited number of memory addressing modes; `ld [x], %r1` is an example of *direct* addressing. Other addressing modes are

   register indirect: `ld %r2, %r1` where %r2 contains the effective address of the source operand

   base plus offset: `ld [%r2+x], %r2` where the effective address of the source operand is found by *adding* x to the contents of register %r2

   In addition arithmetic operations have an immediate mode; for example `jmpl %r15+4,%r0` adds 4 to register %r15

7. Since %r0 is always zero, any time it is used as a destination register, the results are thrown away (see `jmpl %r15+4,%r0` above).

8. The data area (where x, y, and z are located) can go anywhere; here the data area immediately follows the code. Optionally a .org directive could have been used to locate the data area in memory (see below).

9. The instruction `jmpl %r15+4,%r0` is technically a return to the operating system (assuming that %r15 contains the address of the call statement that invoked the program from the operating system). Since the ARC simulator does not support an operating system, %r15 is probably empty which means execution of this instruction would cause a branch to address 4. Hence the `halt` command before the `jmpl` instruction.

Doing I/O using ARC Memory Mapped I/O

There are no explicit I/O instructions for the ARC; instead I/O is done by reading from (loading) and writing to (storing) fixed memory locations in upper memory which are *assigned* to I/O devices. This is referred to as *memory-mapped I/O*. For each device there is a data port and a status port both of which are memory addresses; the later used to test the status of the I/O device to insure it is ready.

The ARC simulator has a console output port to display characters and a keyboard input port to read characters.

Output: Address `0xffff0000` is the console (output) data port; address `0xffff0004` is the console (output) status port where bit 7 is the r*eady* flag (0 not ready / 1 ready). Printing a character is done by *storing* the character to the output data port after first checking that bit 7 (ready flag) of the output status port is set to 1 (ready).

Example: A "Hello World!" program

```
! Prints "Hello World!\n" in the message area

      .begin
BASE  .equ 0x3fffc0     ! starting point of memory mapped area
COUT  .equ 0x0          ! 0xffff0000 Console Data Port
CONSTAT.equ 0x4         ! 0xffff0004 Console Status Port

      .org  2048
Main: add %r0, %r0, %r2       ! clear %r2
      add %r0, %r0, %r4       ! clear %r4
      sethi BASE, %r4
Loop: ld [%r2 + String], %r3  ! load next character
      addcc %r3, %r0, %r3     ! test for null
      be End
Wait: ldub  [%r4+CONSTAT],%r1 ! test console
      andcc %r1, 0x80, %r1    !   status
      be Wait                 ! if busy loop
      stb %r3, [%r4+COUT]     ! otherwise print
      add %r2, 4, %r2         ! increment string offset
      ba Loop
End:  halt
      ba Main

      .org 3000
String:0x48, 0x65, 0x6c, 0x6c, 0x6f ! Hello
       0x20, 0x77, 0x6f, 0x72, 0x6c !  Worl
       0x64, 0x21, 0x0a, 0x00       ! d!\n

      .end
```

Input: Address `0xffff0008` is keyboard (input) data port; address `0xffff000C` is the keyboard (input) status port where bit 7 is the ready flag (0 not ready / 1 ready). To read a character first check that the keyboard status bit 7 is set (ready) then *load* the character from the keyboard data port

Example: A program to read and echo a character

```
! Typewriter.asm

      .begin
BASE  .equ 0x3fffc0     ! Starting point of memory mapped area
COUT  .equ 0x0          ! 0xffff0000 Console Data Port
CONSTAT.equ 0x4         ! 0xffff0004 Console Status Port
CIN   .equ 0x8          ! 0xffff0008 Keyboard Data Port
CICTL .equ 0xc          ! 0xffff000c Keyboard Control Port

            .org 2048
Main:       add %r0, %r0, %r4 ! clear %r4
            sethi BASE, %r4
InWait:     ldub [%r4+CICTL], %r1   ! check kbd status
            andcc %r1, 0x80, %r1
            be InWait

            ldub [%r4 + CIN], %r3   ! get character
            subcc %r3, 27, %r0      ! check for (esc)
            be End

Wait:       ldub [%r4 + CONSTAT],%r1      ! check console status
            andcc %r1, 0x80, %r1
            be Wait
            stb %r3, [%r4 + COUT]   ! echo character
            ba InWait

End:        halt
            ba Main

            .end
```