1 The ARC processor

1.1 Introduction

For educational purposes Murdocca and Heuring developed the ARC processor (Computer Architecture and Organization; An Integrated Approach, ISBN 978-0-471-73388-1). The instruction set of the ARC processor is a subset of the instruction set of the commercial SPARC processor.

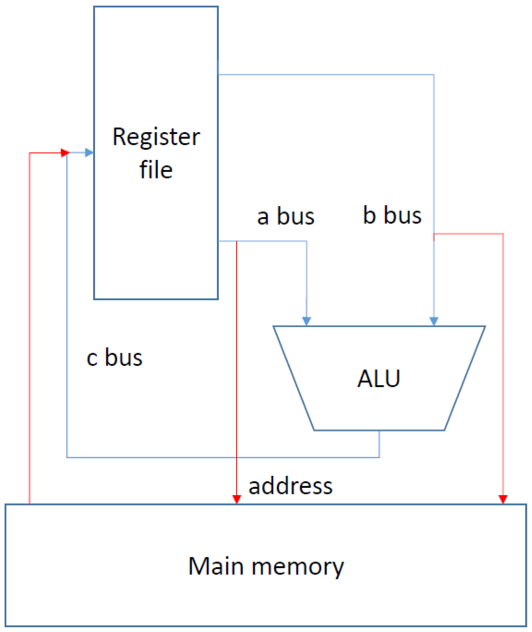


Figure 1: simplified data path of the ARC processor

Properties of the ARC processor:

- It has a LOAD/STORE architecture: only the load and store instruction can access the memory
- The data path and address bus width are 32 bits
- All instructions are 32 bits
- The register file
    - It has 32 visible addresses. ‘visible’ means that it can be addressed by an instruction. Internally a few more register addresses are available for execution of an instruction and to store the instruction that is executed.
    - Each register address has 32 bits
    - Two register addresses can be read at the same time, and at one address data can be stored.
    - Register %r0 is fixed to zero (even if you write to this address).
- The memory is byte addressable; each instruction is on 4 consecutive memory addresses.
- Aligned addressing is used; the ARC processor can read/write 4 bytes at the same time.

1.2 Simple program

You can download an assembler/simulator for the ARC processor on:
http://iiusatech.com/murdocca/CAO/Tools.html (use mirror site)
Note: the JRE (Java runtime environment) is required.

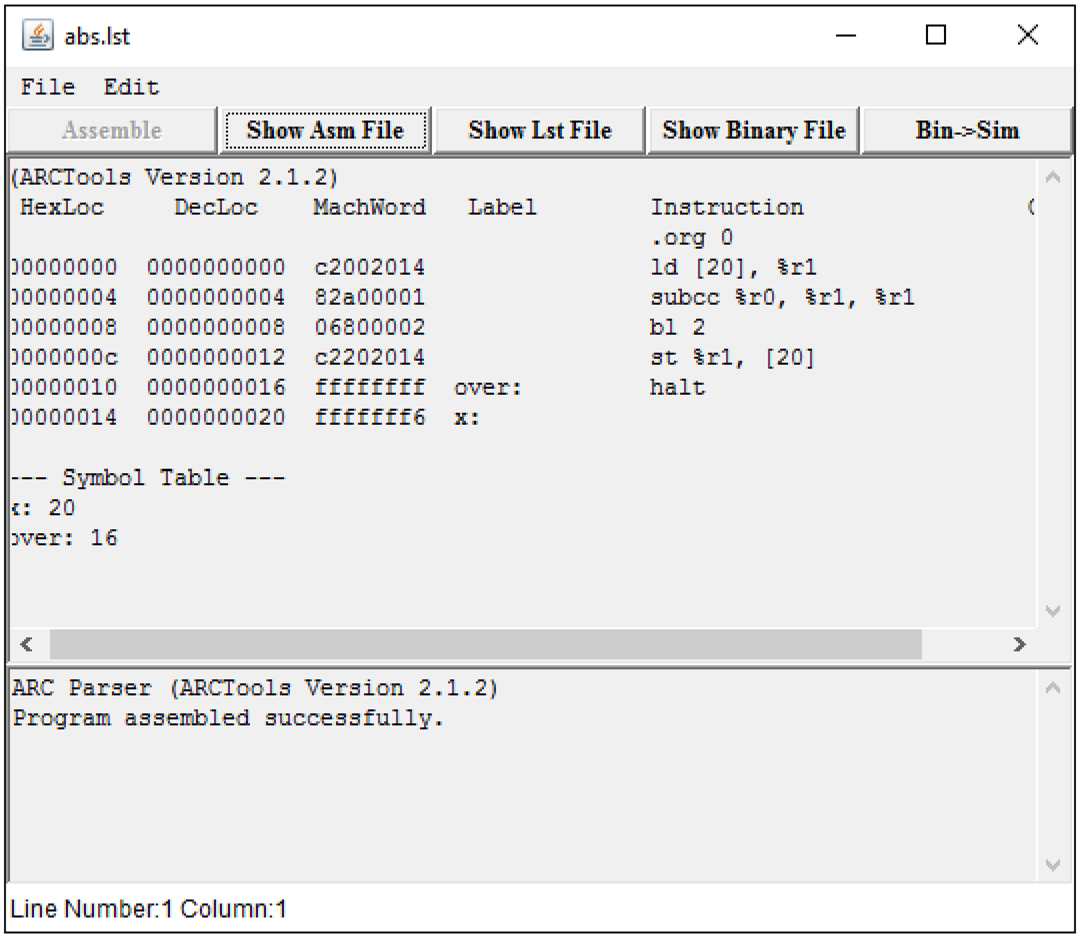
```
1       .begin
2       .org 0
3               ld[x], %r1
4               subcc %r0, %r1, %r1
5               bl over
6               st %r1, [x]
7       over:   halt
8       x:      -10
9       .end
10
```

Note: The line numbers are not part of the program.
This program determines the absolute value of x.

Explanation:

Lines 1 and 9:	Directives .begin and .end must enclose the program
Line 2:	Location of the program in main memory. .org 0 means first instruction (line 3) is located at memory address 0
Line 3:	ld [x], %r1 The data located at memory address x is retrieved from that location and stored in register file location %r1. Note 1: memory location x, x+1, x+2 and x + 3 contain the 32 bits. Note 2: when this line is assembled the address of x is not yet known. Therefore a second pass is required (2 pass assembler).
Line 4:	subcc %r0, %r1, %r2 means %r2 ←%r0 - %r1 sub**cc** %r0, %r1, %r1 has as effect that the twos complement of the contents of %r1 is stored in %r1 (remember: %r0 is always 0). An instruction that ends with “cc” also set the flags N (negative), Z (zero), C (carry) and/or V (oVerflow; signed interpretation) in the processor state register (PSR).
Line 5	bl over means: branch less than goto over In this program. If x is positive, than 0-x is negative. Hence the program will continue with the instruction at address with label over.
Line 6	st %r1, [x] The content of register %r1 is stored in main memory address labeled with over. This address is not yet known (2 pass assembler). In this context: if x was negative than 0-x is positive. The Negative bit is not set (by the subcc instruction on line 4). And now the absolute value is stored at address x.
Line 7	Halt Stop simulation

Line 8	x is a label; at this address the decimal number -10 is stored.
Line 9/10	NOTE: always an empty line after the .end directive! Otherwise the ARCTools wil give an error.



```
(ARCTools Version 2.1.2)
 HexLoc      DecLoc     MachWord   Label        Instruction
                                                .org 0
)0000000  0000000000  c2002014                  ld [20], %r1
)0000004  0000000004  82a00001                  subcc %r0, %r1, %r1
)0000008  0000000008  06800002                  bl 2
)000000c  0000000012  c2202014                  st %r1, [20]
)0000010  0000000016  ffffffff  over:           halt
)0000014  0000000020  fffffff6  x:

--- Symbol Table ---
::  20
over: 16
```

```
ARC Parser (ARCTools Version 2.1.2)
Program assembled successfully.
```

Line Number:1 Column:1

Figure 2: assembled program, output of the ARCTools

The output of the assembler is shown in figure 2. Each instruction is 32 bits (4 memory addresses). *DecLoc* is the decimal address of an instruction in main memory. Label *over* is at address 16 (decimal) and *x* is at 20 (decimal). The machine code of an instruction, in hexadecimal notation, is in column *MachWord*. In the column “shown Binary File” the labels are replaced with the values.

Push “Bin -> Sim” and you can simulate the program.

1.3 Instruction set

The ARC processor supports a subset of the SPARC instruction. The same instruction format for an instruction is used. The following categories:

- Call instruction
- Arithmetic, logical and shift instructions
- Load/store instructions
- Branch and sethi instructions

A category supports more instructions than listed in this document. Only the instructions that are realized in the data path & controller chapter of the book of Murdocca are shown (chapter 5).

1.3.1 Call instruction

call disp30 ! %r15 ←PC and PC ←PC+4xdisp30

The call instruction is used for a function call. After the function is executed the return address must be known. This return address is stored in %r15. The displacement field (disp30) contains the number of instructions you have to go ahead or back. The signed number disp30 is multiplied with 4 since each instruction uses 4 consecutive memory addresses.

For the call instruction the two MSB bits are 01.

		Disp30																													
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0

call 10

1.3.2 Arithmetic, logical and shift instructions

Each instruction has two formats, e.g.

addcc %rs1, %rs2, %rd ! %rd ←%rs1 + %rs2 and set condition code

addcc %rs1, simm13, %rd ! %rd ←%rs1 + sign extension (simm13 field)

The two MSB bits are 10 for this category. Field op3 is 010000 for instruction addcc. The field i ("immediate") is used to distinguish between the two instruction formats. If i (bit13) is 0 the register + register format is used else register + constant.

		rd					op3						rs1					i	not used								rs2				
1	0	0	0	1	1	1	0	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0

addcc %r5, %r6, %r7

		rd					op3						rs1					i	simm13												
1	0	0	0	1	1	1	0	1	0	0	0	0	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

addcc %r5, -1 , %r7

For arithmetic operations the 13 bits in the simm13 field is sign extended to 32 bits.
For logical operation the lower 13 bits is zero extended to 32 bits.
For the shift operation (SRL). The shift only depends on the lower 5 bits (hence only a shift 0 to 31 to the right)

Other arithmetic, logical and shift instructions

instruction	op3 field	The second parameter can be rs2 or simm13
addcc	010000	rd←rs1+rs2; set flags
andcc	010001	rd←rs1 AND rs2; set flags
orcc	010010	rd←rs1 OR rs2; set flags
orncc	010110	rd←rs1 OR NOT(rs2); set flags
Srl	100110	rd←shift right rs1
jmpl	111000	Jump and link, e.g. Jmpl %r15+4,%r0 Add 4 to the content of %r15 and that is the new PC value. The next instruction that is executed is on memory address %r15+4 Also the memory address of the jmpl is stored. Since we do not use it. It is stored in %r0 (this register is always 0).

1.3.3 Load/store instructions

The two MSB bits are 11 for this category.

1.3.3.1 Load (op3 is 000000)

Two formats:

ld [%rs1+%rs2], %rd ! %rd ←MEM[%rs1 + %rs2]

ld[%rs1 + simm13], %rd ! %rd ←MEM[%rs1 + sign extension (simm13)]

		rd					op3						rs1					i	not used								rs2				
1	1	0	0	1	1	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1

ld [%r2 + %r5], %r7

		rd					op3						rs1					i	simm13												
1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	1	0	0

ld [%r1 + 12], %r4

Note:

ld [%r2], %r7 is replaced with ld[%r2+%r0], %r7

ld [5], %r7 is replaced with ld[%r0+5], %r7

1.3.3.2 store (op3 is 000100)

Two formats:

st %rd, [%rs1+%rs2] ! MEM[%rs1 + %rs2] ← %rd

st %rd, [%rs1 + simm13] ! MEM[%rs1 + sign extension (simm13)] ← %rd

		rd					op3						rs1					i	not used								rs2				
1	1	0	0	1	0	1	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0

st %r5, [%r2 + %r6]

		rd					op3						rs1					i	simm13												
1	1	0	0	1	0	1	0	0	0	1	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1

st %r5, [%r2 + 15]

1.3.4 Branch and sethi instructions

The two MSB bits are 00 for this category.

1.3.4.1 Branch instructions

			cond							disp22																					
0	0	0					0	1	0																						

If the condition is true than jump, e.g.

be disp22 ! %r15 ←PC and PC ←PC+4xdisp22

	Cond		Status flags
be	0001	Branch on equal	Z
bcs	0101	Branch on Carry Set	C
bneg	0110	Branch on Negative	N
bvs	0111	Branch on oVerflow set	V
ba	1000	Branch always	

1.3.4.2 Sethi instruction

		rd								imm22																					
0	0						1	0	0																						

sethi imm22, %rd ! %rd[31 ..10]←imm22 and %rd[9..0]←zeros

Why sethi? All instructions have a width of 32 bits and the data width is also 32 bits. How to instruct the processor to store a 32 bits value using an instruction size of 32 bits? That is not possible, therefore the two instructions are used.

Example:

%r1 ←$(ABCD9876)_{16}$

$(ABCD9876)_{16}$ = 1010 1011 1100 1101 1001 1000 0111 0110

The two instructions are:

sethi 1010101111001101100110b, %r1

orcc %r1, 0001110110b, %r1

Or do you prefer hex?

sethi 2AF366h, %r1

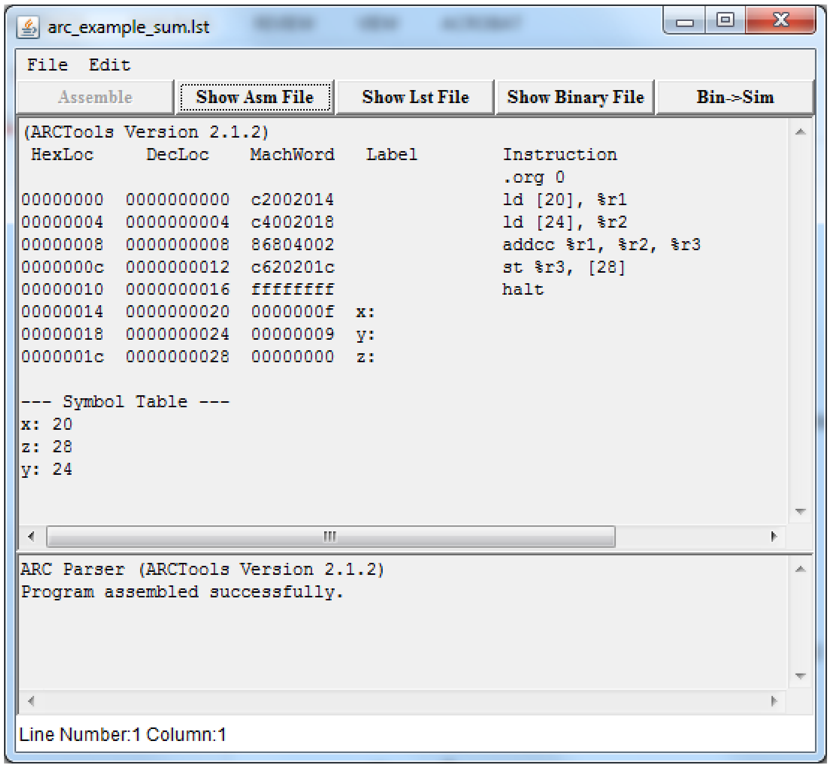
orcc %r1, 073h, %r1

1.4 Exercises

Exercise 1:
What is the machine code of the following instruction (in HEX).
(halt is ffffffff)

```
! This program adds two numbers
        .begin
        .org 0
        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
x:      15
y:      9
z:      0
        .end
```

Answer:



```
(ARCTools Version 2.1.2)
 HexLoc      DecLoc     MachWord    Label        Instruction
                                                 .org 0
00000000  0000000000  c2002014                   ld  [20], %r1
00000004  0000000004  c4002018                   ld  [24], %r2
00000008  0000000008  86804002                   addcc %r1, %r2, %r3
0000000c  0000000012  c620201c                   st %r3, [28]
00000010  0000000016  ffffffff                   halt
00000014  0000000020  0000000f  x:
00000018  0000000024  00000009  y:
0000001c  0000000028  00000000  z:

--- Symbol Table ---
x: 20
z: 28
y: 24
```

```
ARC Parser (ARCTools Version 2.1.2)
Program assembled successfully.
```

Exercise 2

The SUBCC (subtract instruction with status update) is supported by the ARCTools and the instruction format is that of arithmetic instruction.

What is the contents of field op3 for: subcc %r1, %r2, %r3

Answer

		rd					op3						rs1					i	not used								rs2				
1	0	0	0	0	1	1	?	?	?	?	?	?	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0

Almost all information is known (from the arithmetic, logical and shift format).

Reverse engineering:

subcc %r1, %r2, %r3 in the ARCTools -->86A04002

1	0	0	0	0	1	1	0	1	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0

Op3 is 010100

Exercise 3

In this document only the following arithmetic and logical instructions are described:

addcc
andcc
orcc
orncc
Srl

Write a program that using the previous instructions that has effect:

%r3←%r1-%r2

Answer:

```
        orncc %r0,%r2, %r3      ! 1-complement in %r3
        addcc %r3,1,%r3         ! 2-complement
        addcc %r1,%r3,%r3
```

Exercise 4

```
        ! %r10 number of even numbers in array (not included end of array)
        ! %r1 address of array element
        .begin
        .org 0
        sethi arr1, %r1
        srl %r1,10,%r1   ! %r1 is address of arr1
<<<<your program>>>>>
rdy:    halt
        .org 200
arr1:   5,4,3,6,2,3,4,2,1,2,0
        .end
```

Write a program in assembly that counts the number of even numbers in an array.
The array of numbers (each number is 32 bits) is at location with *arr1*. The end of the array is marked with decimal value 0 (not to be counted as an even number).
With the two lines, in red, the memory address of the array is stored in %r1. With *sethi arr1,%r1* the address is stored in %r1, but shifted 10 positions to the left! Therefore the *srl %r1,10,%r1* is required.

Answer:

```
        ! %r10 number of even numbers in array (not included end of array)
        ! %r4 is used to filter LSB bit (odd/even?)
        ! %r3 stores an element of the array is stored
        ! %r1 address of array element
        .begin
        .org 0
        sethi arr1, %r1
        srl %r1,10,%r1   ! %r1 is address of arr1
        addcc %r0,1,%r4        ! %r4 is filter of LSB
        addcc %r0,0,%r10       ! number of even numbers; initial 0
loop:   ld [%r1], %r3
        addcc %r3,%r0,%r0      ! end of array?
        be rdy
        andcc %r3,%r4,%r0      ! filter LSB of array element
        be even
        ba next
even:   addcc %r10,1,%r10      ! incrementend number of even numbers
next:   addcc %r1,4,%r1        ! %r1 address of next element of array
        ba loop
rdy:    halt
        .org 200
arr1:   5,4,3,6,2,3,4,2,1,2,0
        .end
```