1664

public domain

March 26, 2010

Contents

Part I
Architecture

This document briefly outlines the working of the '1664' processor, the assembler and the simulator/debugger.

1 Instruction set

1.1 Opcode encoding

All instructions are conditional with the exception of **ajusta** .
All instructions are encoded with 16 bits. With the exception of **ajusta** the least significant 8 bits determine the operation and the condition. The most significant 8 bits, the parameter(s).

[0..4] opcode bits

[5..7] condition bits

[8..15] parameter bits (the specific decoding depends on the instruction).

(*) **ajusta** opcode:

[0..4] opcode bits.

[5..8] destination opcode (+ 0x10).

[9..15] opcode to be mapped from complete instruction set.

1.2 Condition bits

There are 8 condition bits, 'condition 7' is always read as 1, the others are modifiable using the condition manipulation instructions: ldb, stb, cmp, dep.
The **rev opera_rev_depende_influe** instruction allows other instructions to affect the condition bits. In this mode, the first four bits have a determined meaning.
Condition 0 indicates a zero result.
Condition 1 is the inverse of bit condition 0.
Condition 2 indicates a carry.
Condition 3 indicates an underflow.

1.3 Fixed instructions

1.3.1 ajusta : configure opcode

Parameter range : <0x10..0x1f> <0x00..0x7f>

After execution of **ajusta** the opcode specified by the second parameter is mapped to the opcode specified by the first parameter.
The 16 configurable opcodes allow access to all 128 instructions.
The first 16 non-configurable instructions can be duplicated in the 16 configurable slots providing a means of code obfuscation on a system where the translation table is not visible.
The ajusta instruction is optionally a protected instruction. Protect via **rev** instruction.
See assembler directives *.opera* and *.ajusta*

```
ajusta 0x10 and ; opcode '0x10' interpreted
                ; as 'and' instruction

;equivalent of '7 and 1 3'
.d1 {0x10 7 5 <<} {1 3 2 <<}

;fictional instruction 'new <0..3> <0..63>'
.opera new 2r6r  ; declare instruction name and parameter interpretation as '2r6r'
ajusta 0x11 new  ; opcode '0x11' interpreted as 'new' instruction
.ajusta 0x11 new ; inform assembler of change to opcode value.
0 new 0 0        ; assembled as '0x0011'
```

1.3.2 ldi : load 8b constant

Parameter format: <0..255> Least significant 8 bits of register 0 is replaced with parameter. Remaining bits are unaffected.

```
eor 0 0   ; 0 == '0x0000'
ldi 0x12  ; 0 == '0x0012'
ldi 0x34  ; 0 == '0x0034'
```

1.3.3 ldis : load 8b constant with post 8b shift

Parameter format : <0..255>
Least significant 8 bits of register 0 is replaced with parameter and register is shifted left 8 bits.
Together with **ldi** instruction, a 64 bit value can be loaded using 128 bits of code.
See memory load instruction **ldm** for an alternative to loading register 0 with a constant.

```
eor 0 0   ; 0 == '0x0000'
ldis 0x12 ; 0 == '0x1200'
ldi 0x34  ; 0 == '0x1234'
```

1.3.4 ldm : memory load

Parameter format : <[(+-)0..7(+-)]> (<size>)
<size> 1, 2 or an even multiple of 2 to size of register. (optional) Defaults to size of register.
+ increment (optional)
- decrement (optional)
(+-) before the register modifies the register by <size> before.
(+-) after the register modifies the register by <size> after.

Register 0 is loaded register with <size> element from memory pointed to by A.

If the specified register is 7 (*sIP*) then the offset is calculated adding (unsigned) register 0.

```
ldm [sIP+] 4
.d4 0x12345678
;execution continues here. 0 == '0x12345678'

ldm [6+] ;register 0 is loaded from stack.
         ;The size of the element loaded is
         ;the size of the register
         ;(the default when the <size> is omitted).
```

1.3.5 stm : memory store

store complement to **ldm** instruction.

```
ldi 0x12
stm [-6] 1 ;stack register is pre decremented by '1'
           ;and 0x12 is stored to
           ;location pointed by register 6.
```

1.3.6 ldr : register load

Parameter format: <A0..3> <B0..63>
Load register A with B

1.3.7 str : register store

Parameter format: <A0..3> <B0..63>
Store register A to B

1.3.8 cam : register swap

Parameter format: <A0..3> <B0..63>
Swap registers A, B

1.3.9 yli : relative branch by signed 8b constant

Parameter format: <-127..128>
return register (*sREVENI*) is loaded with address after **yli**
branch instruction pointer + 8b constant shifted left by 1.
Assembler parameter format: <label>
The assembler will calculate the relative offset from the instruction, which must be a multiple of 2.

1.3.10 ylr : absolute branch via register or relative via sIP

Return register (*sREVENI*) is loaded with address after **ylr**A any register

Parameter format: <[A0..63]>
Branch to address pointed to by address in register A

Parameter format: <[+A0..63]>
Branch to (address pointed to by address in register A) relative to *sIP*

Parameter format: <A0..63>
Branch to address in register A

Parameter format: <+A0..63>
Branch to (address in register A) relative to *sIP*

1.3.11 ldb : bit load

Parameter format: <A0..3> <B0..63>
Copy bit B of register A to z (condition bit 0)

1.3.12 stb : bit store

Parameter format: <A0..3> <B0..63>
Copy bit z (condition bit 0) to bit B of register A

1.3.13 cmp : register compare

Parameter format: <A0..3> <B0..63> ; set condition bits
condition bit 0 (alias z) = $A == B$
condition bit 1 (alias n) = $A \neg B$ condition bit 2 (alias c) = $A > B$ condition bit 3 (alias o) = $A < B$

1.3.14 dep : condition bit logic

Parameter format: <A0..7> <B0..7> <C0..3>

The function performed on condition bits A, B are determined by the value of C:
0: $A = A \oplus B$
1: $A = A \wedge B$
2: $A = A \vee B$
3: A swapped with B

Macro alias: *dep_eor,dep_and,dep_or,dep_cam*
clear bit : beor c c
set bit : bor c 7

1.3.15 bit : register bit statistics

Parameter format: <A0..63> <CODE0..3>
Result is stored in register 0.
CODE 0: count of clear bits in register A before first set bit counting from most significant bit
1: count of clear bits in register A before first set bit counting from least significant bit
2: count of set bits in register A
3: count of clear bits in register A

Macro aliases: *msb*, *lsb*, *ones*, *zeros*

1.3.16 rev : multiple functions specified by 8b constant

Parameter format: <0..255>
The function performed by rev is determined by the parameter.

opera_rev_reveni *sIP=sREVENI*

opera_rev_eseta If executed while in protected mode, execution returns to user mode and the user instruction pointer is loaded from *sREVENI_ESETA*. From user mode, protected *sREVENI_ESETA* is loaded with user instruction pointer and a user exception is executed. Register 0 contains the exception offset. See example source code.

opera_rev_ajusta_reinisia Operation translations set by **ajusta** are set to defaults: 16,17,..,31

opera_rev_ajusta_protejeda Protected mode instruction. Prevents user code from configuring opcodes.

opera_rev_ajusta_permete Protected mode instruction. Allows user code to configure opcodes.

opera_rev_depende_influe Allows instructions to influence condition bits.

opera_rev_depende_inoria Prevents instructions (apart from bit instruction) from changing condition bits

opera_rev_sicle_intercambia Protected mode instruction. Register 0 and CPU cycle counter are swapped

opera_rev_sicle_usor_limite_intercambia Protected mode instruction. Register 0 and user cycle limit are swapped
The user cycle limit, limits user code cycle usage before an exception returns control to protected mode code.

opera_rev_sicle_usor_intercambia From protected mode, register 0 and user cycle counter are swapped.
From user code, register 0 is loaded with user cycle counter.

opera_rev_state_reteni_usor Protected mode instruction. Save user state to memory pointed by register 0.
64 registers; 128 opcode translation (presently, stored as 8 bit per opcode), 4 byte CPU flags and conditions, a register size for user cycle counter and cycle limit, each.

opera_rev_state_restora_usor Protected mode instruction. Load user state to memory pointed by register 0.

opera_rev_bp Break point exception. Protected and user mode are distinguished

opera_rev_entra Save *RETENI* (*sR0..31*) registers to (decrease before write) stack

opera_rev_departi Load *RETENI* (*sR0..31*) registers from (increase after read) stack

1.4 Configurable instructions

Logic and arithmetic functions that operate on registers only.

1.4.1 and : register logical and

Parameter format : <A0..3> <B0..63>
condition bits : z,n

1.4.2 or : register logical or

Parameter format : <A0..3> <B0..63>
condition bits: z,n

1.4.3 eor : register logical exclusive or

Parameter format : <A0..3> <B0..63>
condition bits : z,n

1.4.4 shl : register logical shift left

Parameter format : <A0..3> <B0..63>
condition bits : z,n, c=last bit shifted

1.4.5 shr : register logical shift right

Parameter format : <A0..3> <B0..63>
condition bits : z,n, c=last bit shifted

1.4.6 sar : register arithmetic shift right

Parameter format : <A0..3> <B0..63>
condition bits : z,n, c=last bit shifted

1.4.7 plu : register addition

Parameter format : <A0..3> <B0..63>
$A = A + B$

1.4.8 sut : register subtraction

Parameter format : <A0..3> <B0..63>
$A = A - B$

1.4.9 sutr : register reverse subtraction

Parameter format : <A0..3> <B0..63>
$A = B - A$

1.4.10 mul : register multiply (integer/floating point)

Parameter format: <A0..3> <B0..63>
Register 0 contains the most significant bits of product with adjustment.
Adjusted so the most significant bit of product (*sMASIMA*) occupies the most significant bit of register 0.
Least significant bits shift from *sMINIMA* .
sDESLOCA contains the number of bits register 0 need to be shifted to produce correct result
if value is unsigned, $(A * B)$ = (register_0)<<*sDESLOCA*
if value is signed (two's complement), $(A * B)$ = (register_0)>>*sDESLOCA*
sMASIMA contains most significant bits of product
sMINIMA contains least significant bits of product

mul and **div** operations are cumulative on *sDESLOCA* . See software FPU implementation (f2) for an example.

1.4.11 div : register divide (integer/floating point)

Parameter format: <A0..3> <B0..63>
Register 0 contains the most significant bits of division with adjustment.
Adjusted so the most significant bit of product (*sMASIMA*) occupies the most significant bit of register 0.
sDESLOCA contains the number of bits register 0 need to be shifted to produce correct result
if value is unsigned, $(A * B)$ = (register_0)<<*sDESLOCA*
if value is signed (two's complement), $(A * B)$ = (register_0)>>*sDESLOCA*
sMINIMA $= A/B$

mul and **div** operations are cumulative on *sDESLOCA*. See software FPU implementation (f2) for an example.

1.5 Registers

The processor may be configured with up to 64 registers.

1.5.1 6 stack pointer

Register 6 *sPILA* is used as the the stack pointer by the example code.

1.5.2 7 instruction pointer

Register 7 *sIP* the instruction pointer. When read it points to the instruction's address+2.

1.5.3 59

Register 59 *sDESLOCA* See arithmetic instructions **mul**, **div**.

1.5.4 60

Register 60 *sMINIMA* See arithmetic instructions **mul**, **div**.

1.5.5 61

Register 61 *sMASIMA* See arithmetic instructions **mul**, **div**.

1.5.6 62 exception return pointer

Register 62 *sREVENI_ESETA* contains the address, following the instruction, that caused and exception.
It is used as the return address for the instruction **rev opera_rev_eseta** when executed as a 'privileged' instruction. See instruction **rev**.

1.5.7 63 branch return pointer

Register 63 *sREVENI* contains the link address. See branch instructions **yli, ylr**.
The instruction **rev opera_rev_reveni** assigns the value of *sREVENI* to the instruction pointer.

2 Memory mapped peripherals (simulator)

Peripheral configuration registers are mapped to a reserved memory block that are accessible from system code or user code provided the register address are mapped to the user code memory space. See memory management unit.

2.1 Memory management unit

A simple MMU is included in the simulator that provides configurable access rights per mapped area of memory. The MMU translates the requested address via a translation table with the following format (1664 assembler syntax):

```
.ds {SIZE PERMISSION |} ;least significant bits determine access.
.ds VIRTUAL_OFFSET
.ds REAL_OFFSET

;... further entries

.ds 0 ;end
```

Permission bits are as follows:
0 : read operation allowed when set to 1
1 : write operation allowed when set to 1
2 : execute operation allowed when set to 1

The first word contains the size of the mapped area (most significant bits) and 3 access permission bits (least significant bits). As a consequence *SIZE* is 8 bytes aligned.

Part II
Assembler

The assembler requires, at the minimum, a file with an instruction declaration. (See directive *.opera*).

3 Syntax

3.1 Instructions

(condition) instruction_name (parameter_1) (parameter_2) (..) The condition does not need to be specified as it is assumed to be 7 (always true) if left out.
All instructions are conditional, with the exception of the instruction **ajusta**.
Example of using conditional instructions:

```
eor 1 1
eor 0 0
ldi 1 ;register 0 == 1
cmp 0 1
z ldi 2 ;register 0 == 1
n ldi 3 ;register 0 == 3
c ldi 4 ;register 0 == 4
o ldi 5 ;register 0 == 4
```

3.2 Labels

Labels are declared simply by placing the name on a new line.
To create a local label, the label declaration must begin with a *:* . The local label is then preceded with an @ . And is referenced by the name of the :label concatinated with the local label.
Example of a loop and local label:

```
rev 5 ;instructions afect condition bits.
label0
eor 0 0
ldi 2
eor 1 1
ldi 1
:label1
label2
```

```
@local
sut 0 1
n yli label1@local
yli label2
```

4 Directives

4.1 .inclui <file name>

Include *filename*and assemble immediately.

4.2 .opera <instruction name> <parameter function name>

Before an instruction can be identified and assembled the name must be declared and the parameter format specified from hard-coded functions. The opcode is inferred from the order which the instructions are declared.
See *base.opera.1664* .

```
.opera instruction_name parameter_function
;instruction_name is given opcode 0x01 (ajusta is always 0x00)
;subsequent .opera .. are given + 1 from the previous
```

4.3 .ajusta <instruction opcode 0x10..0x1f> <instruction name>

Interpret **instruction name** as having the specified opcode.
Used to inform assembler that the **ajusta** instruction has changed the decoding.
See the instruction **ajusta** .

4.4 .defina <tag> <value>

Assign a value to a tag. Undefined tags evaluate as 0.

4.5 .nodefina <tag>

Undefine a tag. Undefined tags evaluate as 0.

4.6 .d1 <value0> (<value1>) (...)

Interpret values as 8 bit and output directly to image.

4.7 .d2 <value0> (<value1>) (...)

Interpret values as 16 bit and output directly to image.
.do, *.d2* are equivalent in current implementation

4.8 .do <value0> (<value1>) (...)

Interpret values as instruction word (16 bit) values
.do, *.d2* are equivalent in current implementation.

4.9 .d4 <value0> (<value1>) (...)

Interpret values as 32 bit and output directly to image
(*.ds*, *.d4* are equivalent when the CPU is configured with 32 bit registers)

4.10 .ds <value0> (<value1>) (...)

Interpret values as register (up to 64bit) size and output directly to image.

5 Arithmetic

Equations (including directives) are evaluated using reverse polish notation (RPN) method enclosed between {}.Nested {} evaluated.
Values are treated as 64 bit unsigned integer. Each value is added to a 'last in first out' stack.
example

```
.defina a {100 1 +} ;tag 'a' defined as 101
.defina b {a 2 -} ;tag 'b' defined as 99.
```

5.1 RPN Directives

5.1.1 .x

Exchange last two values on stack

5.1.2 .d

Duplicate value on RPN stack indexed by last value (not included in count)

5.1.3 .<

Drop n values from stack, n is determined by last value and is dropped additionally.

5.1.4 .-

Unary function: twos compliment

5.1.5 .+

Unary function: absolute value

5.1.6 ./

Unary function: $1/x$
Arithmetic performed as 64 bit integers.

5.2 RPN functions

$-$;subtract
$+$;add
$*$;multiply
/ ;divide
>> ;shift right second last value by last value

<< ;shift left second last value by last value
& ;logic: and
| ;logic: or
ˆ;logic: exclusive or

Boolean functions evaluate 1 if true otherwise 0. Last two values are replaced with result.
> ;boolean: compare last two values: greater than
< ;boolean: compare last two values: less than
>= ;boolean: compare last two values: greater than or equal
<= ;boolean: compare last two values: less than or equal
= ;boolean: compare last two values: equal
!= ;boolean: compare last two values: not equal
|| ;boolean: (or) values and compare with 0
&& ;boolean: (and) values and compare with 0
‘;boolean: (exclusive or) values and compare with 0

6 Examples

6.1 Loading constants

```
;24bits 0xaabbcc to register sT0

;using ldis and ldi 8 bytes
eor 0 0 ;0x0000000000000000
ldis 0xaa ;0x000000000000aa00
ldis 0xbb ;0x0000000000aabb00
ldi 0xcc  ;0x0000000000aabbcc

;using ldm 6 bytes
ldm [sIP+] 4
.d4 0x00aabbcc
;execution continues
```

6.2 Configuring instruction

```
ajusta 0x1f and ;configure 'and' in to last slot
.ajusta 0x1f and ;inform assembler to interpret 'and' as 0x1f
0 and 0 0 ;encoded as 0x001f
rev opera\_rev\_ajusta\_reinisia ;return to default configuration
.implicada ;inform assembler operations has default encoding
0 and 0 0 ;0x0010
```

7 Macros

7.1 Macro definition

.model <tag> {} parameter0 parameter1 ... ;Named macro. Output when invoked by name.
{} ;Single line macros are evaluated in place.

7.2 Macro functions

Use within a macro definition {}.

7.2.1 %0..9

Substituted with corresponding macro parameter.

7.2.2 %{}

Substituted with corresponding macro parameter indexed as evaluated between {}.

7.2.3 %I"<text>"

Macro information <text> printed to console.

7.2.4 %!{}"<text>"

Assembler error flagged and <text> printed to console if {} evaluates !=0.

7.2.5 %.{}

Macro expansion will stop at this evaluation if {} evaluates !=0.

7.2.6 %+{} <line>

Include <line> (the remaining text) if {} evaluates !=0

7.2.7 %-{} <line>

Remove <line> (the remaining text) line if {} evaluates !=0

7.2.8 %>{} <c>

Skip character <c> (advance one character) if {} evaluates !=0

7.2.9 %@

Replace with current instruction pointer

7.2.10 %c

Replace with parameter count

7.3 Macro examples

```
;Single line macro. Output 0
;(size of register) if instruction pointer
; evaluates greater or equal to 1001
{%+{%@ %0 >=}.ds 0} 1001
```

7.3.1 Named macro

```
.model named {
%+{%@ %0 >=}.ds 0
}
;code
named 1001 ;Output 0 (size of register) if instruction pointer
           ;evaluates greater or equal to 1001
```

7.3.2 Recursive macro

'something' repeated %0 (10) times.
Each evaluation the parameter is decreased by 1 and passed to the named macro *reveni* after executing 'something'.
After 10 passes %+ evaluates false and the next iteration is skipped.

```
.model revinente {
.defina contador {%0 1 -}
something
%+{contador 0 !=} revinente contador
}
;code
reveni 10
```

Part III
Simulator

Parameters intended for the simulated 1664 (See the instruction **imita**) are distinguished by starting with a + character (convention).

8 Command line

8.1 -v : verbose

8.2 -d : start in debugger (execute)

8.3 -e : execute after assemble

8.4 -f : assemble file

8.5 -I : search path

8.6 -m : memory allocate in MB to CPU

8.7 -b : write binary image

8.8 -B : read binary image

9 Debugger

Fo the debugger to display the correct instruction names and decode the parameters correctly, the instruction delcarations that where used to assemble the file need to be supplied. (See assembler directive *.opera*).

9.1 Key commands

9.1.1 c : run

9.1.2 n : step in

9.1.3 N : step over

9.1.4 m : break after

10 Additional instruction

To provide a way for the simulator to be used as an interpreter for 1664 code/binaries an additional instruction is included that provides an interface with some system calls (presently limited to GNU/Linux).
The example code uses **imita** at the 'operating system' level. The user code invokes the function by using the native user exception mechanism **rev opera_rev_eseta** .

10.1 imita : simulate operating system call

Parameter format: <0..255>
See *imita.inclui.1664* for specific definition
See *imita.0.1664* for code example.