NICE
an elegant and powerful 32-bit architecture

B. Ulmann
ulmann@fafner.zdv.uni-mainz.de

September 8, 1997

Abstract

The architecture described in the following articel is a direct successor of μ–EP–1 (cf. [1]) and was developed by the author and Robert Linden (Universität Bonn, FB Informatik). **NICE** is a 32-bit processor, utilizing a fixed instruction format, a register set of sixteen general purpose registers and an extremely powerful but simple and easy to use instruction set which is supported by a variety of addressing modes. The smallest direct addressable data item in main memory is the 32-bit machine word thus utilizing a simple addressing scheme for instruction operands.

The aim of this design is reflected by the acronym "**NICE** " which means "**NICE** is charmingly elegant" (Robert Linden).

At the moment **NICE** contains neither any hardware support for memory management nor floating point instructions, but using the instruction selection scheme described below features like these can be added without changing the overall machine design.

1 Registers

NICE contains a register set of sixteen registers, R0. . . R15, featuring a predecrement and postincrement capability which can be used for implementing stack like structures etc. R1. . . R12 are true general purpose registers, while R0, R13. . . R15 have special meanings as follows:

R0: This register contains the hardcoded value 0 which is returned by each read operation. It can be also used as a destination register but writing to R0 will discard the value to be written.

R13: After receiving a non masked interrupt request, this register (denoted by **IR** in the following) contains the appropriate return address.

R14: This special register serves as the system status register, abbreviated **SR**. It contains the following status flags[1] and address fields as shown in figure 1:

1	X	C	Z	N	V	M	I	Address of interrupt-vectorstructure	
31	30	29	28	27	26	25	24	23	0

Figure 1: The **SR** structure

1 – Alway TRUE, used in conditional instruction execution, see section 2.

X – TRUE if the result obtained by the last operation was \$FFFFFFFF.

C – contains the carry of the last ALU/SHIFT-instruction.

Z – A value of 1 denotes that the last result was zero.

N – TRUE if a negative result was generated, i.e. the most significant bit of the result value was set.

V – TRUE denotes an overflow condition.

M – TRUE if a maskable interrupt occured, see section 4.

I – The interrupt-control-flag, if this flag is set, maskable interrupts are enabled, else disabled. And the

address of the interrupt vector structure: This value is used as a pointer to the beginning of the interrupt vector structure located in main memory which is used for interrupt processing, see section 4.

R15: This register is used as the program counter for **NICE** and is denoted by **PC**.

It should be emphasized that – despite the special purpose of the above mentioned four registers – all of the sixteen registers R0. . . R15 of **NICE** can be used in every instruction context without restrictions. This feature plays

[1]Note that it is possible to inhibit the modification of the status flags by an ALU- or SHIFT-instruction using a special flag when coding the desired operation.

Condition field	Opcode	Destination	Source_1	Source_2
31 28	27 21	20 14	13 7	6 0

Figure 2: General instruction format

3 select bits	/

Figure 3: Condition coding

a keyrole for the powerful but simple programming concept of **NICE**.

2 Instruction coding

NICE features a fixed format instruction coding scheme, shown in figure 2, thus simplifying the necessary hardware for instruction decoding. Instructions have a length of 32 bits (starting at word boundaries in the main memory since the 32-bit machine word is the smallest addressable item). Generally each instruction is composed of the following five fields:

- The *condition field*, (refer to figure 3) which is used to select one out of the eight most significant status register bits (for a description of these flags see section 1) by using bits 31…29 of the instruction word as selector field. Depending on the value of the status flag selected this way, the instruction can be executed or skipped. An instruction is executed only, if the desired bit of **SR** is set. To simplify programming, bit 28, the /-bit, of the instruction can be used to negate the result of testing the condition flag. The 1-bit at position 31 of the status register provides a convenient way for coding instructions to be always executed.

- The *deselect and opcode field*: **NICE** instructions are grouped into categories – at the moment there exist the following three instruction groups:

  1. ALU-instructions,
  2. SHIFT-instructions and
  3. the HALT-instruction.

Mode bits	Register number
2 1 0	4 bits

Figure 4: Operand coding

To select a particular instruction out of these groups, a *deselection scheme* is used in the first stage of instruction decoding the following way[2]:

```
if bit 27 is cleared then
    executed ALU-instruction
else
    if bit 26 is cleared then
        execute SHIFT-instruction
    else
        HALT-instruction
    endif
endif
```

The remaining bits of the *deselect and opcode field* are used as an opcode field for controlling the functional unit selected by the above scheme.

- Three fields for coding one destination and two source operands. The format of each of these fields is shown in figure 4. The three mode bits are used to specify the desired addressing mode and are interpreted as outlined in the following table, while the remaining four bits select one out of the sixteen available registers:

mode bits 2	1	0	destination operand	source operand
0	0	0	Rxx	Rxx
0	0	1	Rxx--	--Rxx
0	1	0	Rxx++	Rxx++
0	1	1	@#<const>[Rxx]	#<const>[Rxx]
1	0	0	@Rxx	@Rxx
1	0	1	@--Rxx	@--Rxx
1	1	0	@Rxx++	@Rxx++
1	1	1	@#<const>[Rxx]	@#<const>[Rxx]

Mode bit 2 is used to select indirect addressing, which is denoted by the symbol @. In this case, the value of the operand is used as a pointer to the main memory location to be used as the actual operand. In the coding scheme shown above are two exceptions to be noted:

[2]Note that this deselection scheme provides an easy way for extending the **NICE** instruction set.

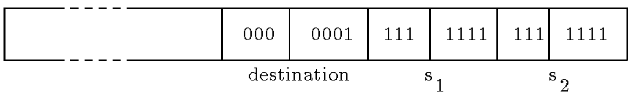

Figure 5: Addressing mode coding example

- Coding a constant as the destination operand of an instruction is quite meaningless and thus forced to be interpreted as a pointer to the destination location, i.e. coding #<const> in the destination of an instruction is equal to @#<const>.

- Selecting a register in the predecrement mode as the destination of an operation would have no effect on the value to be stored, so in this special case the value stored in the desired register is postdecremented!

Note that, when coding a constant value as an operand, there always will be added the value of the register selected by the four register number bits prior to any other operation taking place on this operand! This allows a powerful indexed addressing capability and is the main reason for the special role of R0. Using this register in conjunction with constants allows the usage of a pure constant value without interfering with actual register contents.

The following example may show how the address resolution scheme implemented in **NICE** actually works: Two values have to be added which are located \$10 and \$11 memory locations after the instruction itself, while the result should be stored in R1. This could be accomplished by coding the instruction `add r1,@#10[PC],@#$11[PC]`. The three destination and source operand fields then contain the values shown in figure 5 while the constants \$11 and \$10 are stored in the two memory locations directly following `add`-instruction itself[3].

Utilizing this addressing feature, writing position independent code is extremely simplified by using the program counter R15 in conjunction with constant address pointers. It is important to notice that the **PC** points to the actual instruction location during the whole execution cycle!

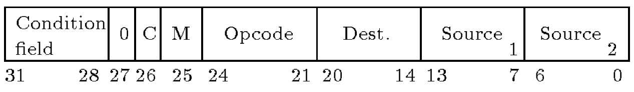

Figure 6: ALU-instruction format

3 Instructions

All mnemonics used in the following sections are identical to those implemented by the **NICE** crossassembler written by Robert Linden.

3.1 ALU-instructions

NICE implements the following sixteen different ALU-instructions which are divided into arithmetic and logic functions:

Opcode	Mnemonic	Description	arithmetic/ logic
0000	move	$d = s_1$	a
0001	sub	$d = s_1 - s_2$	a
0010	mdbl	$d = s_1 + (s_1 \wedge s_2)$	a
0011	add	$d = s_1 + s_2$	a
0100	dbl	$d = 2s_1$	a
0101	dec	$d = s_1 - 1$	a
0110	not	$d = \neg s_1$	l
0111	nor	$d = \neg(s_1 \vee s_2)$	l
1000	iand	$d = \neg s_1 \vee s_2$	l
1001	nand	$d = \neg(s_1 \wedge s_2)$	l
1010	xor	$d = s_1 \oplus s_2$	l
1011	ior	$d = \neg s_1 \vee s_2$	l
1100	xnor	$d = \neg(s_1 \oplus s_2)$	l
1101	and	$d = s_1 \wedge s_2$	l
1110	one	$d = 1$	l
1111	or	$d = s_1 \vee s_2$	l

All instructions are coded as shown in figure 6. The two bits denoted by **C** and **M** control the usage of the input carry and the modification of the status flags contained in **SR**.

If the **M**-bit (the *modify-bit*, bit 25) is set, the status flags are affected by the result obtained from the current instruction. If bit 26, the **C**-flag, of an ALU-instruction is set and the instruction to be executed is one of the arithmetic operations, the result is incremented by one if the carry bit in the status register **SR** is set. (Logical ALU-instructions are not affected by this mechanism.)[4]

[3] Constants used as operands for instructions are stored in the memory locations directly following the instruction itself beginning with the first source operand and ending with the destination operand.

[4] As a result of the instruction interpretation scheme described above, a value of zero is treated as *move r0,r0* – some kind of a *nop*-instruction.

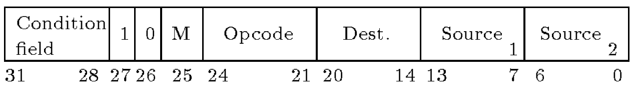

Figure 7: SHIFT-instruction format

3.2 SHIFT-instructions

Additional to the 16 ALU-instructions there are sixteen SHIFT-instructions which can be executed on **NICE**:

Opcode	Mnemonic	Alternative
0000	0<0	shl
0001	0<1	
0010	0<C	
0011	C<0	
0100	C<1	
0101	C<C	
0110	<<<	rol
0111	C<<	bpl
1000	0>0	shr
1001	1>0	
1010	C>0	
1011	0>C	
1100	1>C	
1101	C>C	
1110	>>>	ror
1111	>>C	bpr

To clearify the notation used in the table above consider the SHIFT-instruction 0<0: This is interpreted as a left shift filling up with 0 and discarding the most significant bit after each step. In exactly the same manner 1>0 is executed as a right shift, filling up with 1 and discarding the least significant bit after each step. The character C denotes the carry flag in the status register **SR**. <<< and >>> are used to specify circular left and right shifts respectively. C<< and >>C, on the other hand, denote circular left and right shifts with the most (least) significant bit copied to the carry bit after each step.

During a SHIFT-operation the source operand s_1 is shifted s_2 times. These instructions are coded as shown in figure 7, where the **M**-bit controls the modification of the flags contained by the status register as described in section 3.1, with the exception that SHIFT-operations using the carry bit as a destination will modify this flag regardless of the state of the **M**-bit!

3.3 The HALT-instruction

This special instruction is used to halt the entire processor – a restart is possible only by sending a reset pulse to the hardware. Its format is depicted in figure 8.

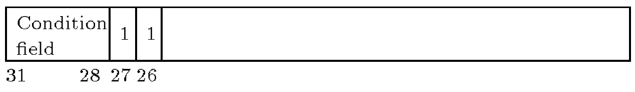

Figure 8: HALT-instruction format

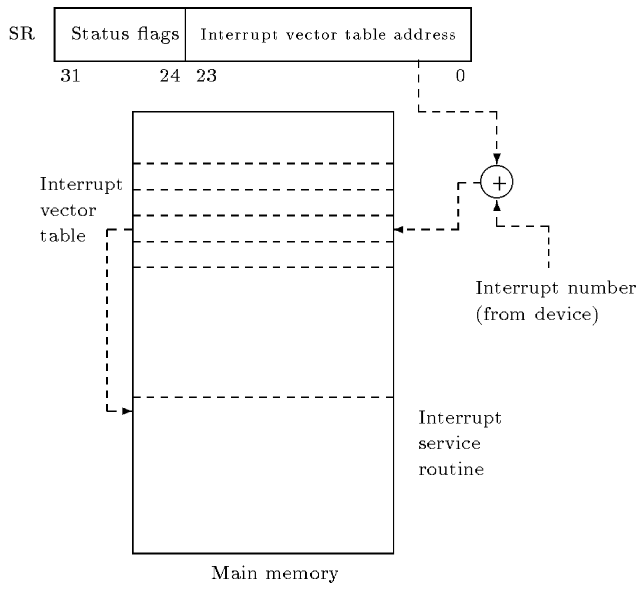
SR
Status flags
Interrupt vector table address
31
24
23
0
+
Interrupt vector table
Interrupt number (from device)
Interrupt service routine
Main memory

Figure 9: Interrupt processing scheme

4 Interrupts

NICE supports two basic types of interrupts: maskable and nonmaskable interrupts. Maskable interrupts can be disabled by clearing the **I**-bit of the status register **SR**, while nonmaskable interrupts can not be inhibited. Besides this difference, both types are serviced the same way:

A device requests an interrupt by changing the state of the `IRQ`- (for maskable interrupts) respectively `NMI`-signal from high to low. If the interrupt is accepted by **NICE**, this is signalized to the requesting device by asserting a low-level on the appropriate `IGT`- /`NMIGT`-line depending on the interrupt type which was requested. In a further step, the external device sets the lower eight bits of the data bus to the desired interrupt number – availability of this data is signalized to **NICE** by a special control line. This value is read by the **NICE**-hardware and used as the eight least significant bits of a pointer to the memory location containing the address of the requested interrupt service routine. The upper 24 bits of this pointer are obtained from the bits 0...23 of the status register.

To allow a proper termination of the interrupt handler routine, the current value of the program counter R15 is stored in R13, which can be used by the service routine to return the program flow to the interrupted program. This procedure is shown in figure 9.

The occurence of a maskable interrupt sets the **M**-bit of the status register, which enables the interrupt handler routine to distinguish between the two interrupt types.

5 Examples

The following section gives a short description of the syntax used by the **NICE**crossassembler written by Robert Linden, which is used to compile the examples below:

5.1 Coding format

To code **NICE**-instructions, the mnemonics for ALU- and SHIFT-operations as described in the preceeding sections are used. If an instruction shall be able to change the status flags contained in the status register R14, the **M**-bit is set by coding <M> after the instruction mnemonic itself. The same syntax is used for controlling the value of the **C**-flag used in ALU-instructions. To set both flags, <CM> can be coded.

Constants are detected by a leading #-sign – this is also necessary when using a label or a constant defined by an .equ-statement!

Indirect addressing is denoted by a @, while indexed addressing is coded by @#<const>[Rxx]. The ";"-sign indicates a comment.

"?"denotes a conditional executed instruction and is followed by the name of the status flag to be tested. The sign "!" denotes that the value of the flag selected by "?" has to be negated prior to testing the condition.

5.2 A 32-bit multiplication routine

The following routine allows multiplication of two 32-bit integer values using a simple shift-and-add algorithm:

```
;mult_32
;calculates value_1*value_2
;and stores the result in
;the memory locations
;result_lo and result_hi.
;
          .org   $0
;load operands:
          mov    r1,@#value_1
          mov    r2,@#value_2
;clear all used registers:
          mov    r3,r0
          mov    r4,r0
          mov    r6,r0
;multiplication loop:
loop:     0>c<m> r1,r1,#1  ;rightshift to carry
      ?!c mov    r15,#skip ;zero? => skip add
          add<m> r3,r3,r1  ;add to result_lo
          add<c> r4,r4,r5  ;add to result_hi
skip:     c<0    r2,r2,#1  ;double value_2_lo
          0<c    r5,r5,#1  ;double value_2_hi
          mov<m> r6,--r6   ;decrement counter
      ?!z mov    r15,#loop ;not done? => loop
;store result:
          mov    @#result_lo,r4
          mov    @#result_hi,r5
          halt
;
value_1:  .dat   $0
value_2:  .dat   $0
result_lo:.dat   $0
result_hi:.dat   $0
```

5.3 An interrupt service routine

The following code fragment shows how interrupt service routines can be implemented on **NICE**[5]:

```
;main program:
;
;initialize r12 which serves as a stack
; pointer:
          mov    r12,#start_stack
;initialize the vector interrupt table pointer:
          0<0    r14,#ivt,#8
;set the interrupt enable flag:
          or     r14,r14,#$0200
;
          ...                ;code of main
;
          halt               ;end of main
;
;first interrupt routine:
;
ihandler_0:
;save return address and status register value:
          mov    @r12++,r13
          mov    @r12++,r14
;
          ...                ;code of the
                             ;interrupt handler
;restore status register and program counter:
          move   r14,@--r12
          move   r15,@--r12
;
;end of the interrupt handler.
;
```

[5]The assembler directive .ivtable garanties that the following data starts at a 256 machine word boundary, so the upper 24 bits of the associated label can be stored directly to the interrupt vector structure address field of the status register **SR**.

```
          .ivtable
ivt:      .dat   #ihandler_0
          .dat   #ihandler_1
          ...              ;and so on.
start_stack:
          .reserve $1000   ;reserves the
                           ;following $1000
                           ;memory locations
```

It should be noted that the assembler allows the definition of some kind of macros thus avoiding the necessity to code things like `or r14,r14,#$0200` over and over again. This operation could be defined as for example `si` – set interrupt flag. In the same way pseudoinstructions like `bgi` (*begin interrupt handler*) and `rti` (*return from interrupt*) could be defined, serving as aliases for instruction sequences like

```
          mov    @r12++,r13
          mov    @r12++,r14
```

and

```
          mov    r14,@--r12
          mov    r15,@--r12.
```

5.4 A self replicating program

The purpose of the following program is to create a copy of itself:

```
          mov    r1,r15    ;get start and
          mov    r2++,#end ;end location.
          mov    r3,r2
loop:                      ;copy one word:
          mov    @r2++,@r1++
          xor<m> r0,r1,r3  ;r1=r3? => set z
      ?!z mov    r15,#loop ;not done? => loop
end:      halt
```

6 Conclusion

It may be concluded that **NICE** extends the capabilities of μ–EP–1 in a way that leads to a very powerful 32-bit computer with a wide range of possible applications. Due to its simple assembly language this architecture is well suited for educational purposes as well as for small control applications.

Further steps are an implementation of the **NICE** architecture using FPGAs instead of discrete TTL-circuits as in μ–EP–1, and the development of a common interface bus.

References

[1] Ulmann, Bernd: *μ–EP–1, a simple 32-bit architecture*, Computer Architecture News, 6/95.