QNICE – a nice 16 bit architecture

ISA v. 1.6

Bernd Ulmann

August 18, 2020

Why a new 16 bit processor architecture? Why not stay with commodity products and a wider bus width?

- First of all, there is nothing like developing your own CPU from scratch – nothing!
- The QNICE architecture was developed during 2006 with its 32 bit predecessor NICE (cf. [2] and [3]) in mind.
- The 16 bit data bus width was chosen to ease an actual implementation of the processor either using TTL chips as in many other homebrew CPU projects[1] or using more modern FPGAs with a bit of surrounding circuitry.
- Thanks to Mirko Holzer the project was revived in 2015, first with an AVR-implementation and then with a FPGA-version of this processor.

---

[1]Most notably Bill Buzbee's Magic, cf [1].

- 16 bit data and address bus width (little endian!)
- Rather fixed instruction format – every instruction occupies one 16 bit machine word
- 16 general purpose registers divided into two banks of eight registers each
- The register bank containing registers $0 \ldots 7$ is actually a window to a high speed RAM so in fact there are $256 \times 8 + 8 = 2056$ registers all in all
- moving the register window is accomplished in a single operation making push/pop operations virtually unnecessary
- Very small instruction set (18 instructions)
- 4 addressing modes

- At any moment of a program run there are 16 general purpose registers visible to the program:

| R0 | . . . | R7 | R8 | . . . | R13 | R14 | R15 |

- Some registers serve a special function in the processor:
    - R13: Normally used as a stack pointer – especially the subroutine call instructions use this register as a stack pointer (SP).
    - R14: Statusregister (SR for short).
    - R15: Program counter (PC).

- The upper eight registers R8. . . R15 are always the same while the lower set of eight registers is a window into a 256×8 register bank of 16 bit bus width.

The status register is divided into two parts: The lower 8 bits are
the status bits reflecting the current processor state while the
upper 8 bits (rbank) are used to control the register bank circuitry:

rbank	—	—	V	N	Z	C	X	1

1: Always set to 1

X: 1 if the last result was 0xFFFF

C: Carry flag

Z: 1 if the last result was 0x0000

N: 1 if the last result was negative

V: 1 if the last operation caused an overflow

- As already mentioned, the upper 8 bits of R14, called rbank, control the register bank circuitry.
- Since there are 256 times 8 registers available as R0...R7, the eight bits of rbank suffice to specify one out of these 256 pages as the actual register page to be used.
- To switch between register pages it is only necessary to change the contents of rbank – normally this will be accomplished by a simple ADD or SUB instruction.
- The multiple register banks are very handy in programming subroutines since they remove the necessity of saving lots of registers on entry and restoring them on exit of a subroutine.

- All input/output operations of QNICE take place through a memory mapped I/O system, so there are no special I/O instructions as some other processors feature.

- The upper 256 words of memory are reserved for I/O controllers which can be easily accessed using normal instructions with addressing modes referring to memory cells.

- QNICE utilizes 18 basic instructions, all of which (apart from the control group, which takes one optional operand) are two operand instructions.
- Instructions like ADD R0, R1 will actually perform an operation like R1 := R1 + R0 – the only exceptions being
    - the two shift instructions SHL and SHR where the first operand specifies the number of places to be shifted and
    - the four jump and branch instructions ABRA, ASUB, RBRA and RSUB which only take a a destination and a condition code.
- All operands, apart from the condition code of a jump or branch instruction, of course, can be specified using one out of four possible addressing modes (Rxx, @Rxx, @Rxx++ and @--Rxx).

- Most of QNICE's instructions feature a single instruction format, the only exceptions are the four branch and jump instructions:

4 bit	4 bit	2 bit	4 bit	2 bit
opcode	src rxx	src mode	dst rxx	dst mode

- The control instructions with opcode E have the following general format:

4 bit	6 bit	4 bit	2 bit
opcode	command	dst rxx	dst mode

- The four jump and branch instructions use the following instruction format with mode bits 00,01, 10, 11 specifying ABRA, ASUB, RBRA, and RSUB:

4 bit	4 bit	2 bit	2 bit	1 bit negate condition	3 bit select condition
opcode	src rxx	src mode	mode	negate condition	select condition

Opc	Instr	Operands	Effect
0	MOVE	src, dst	dst := src
1	ADD	src, dst	dst := dst + src
2	ADDC	src, dst	dst := dst + src + C
3	SUB	src, dst	dst := dst - src
4	SUBC	src, dst	dst := dst - src - C
5	SHL	src, dst	dst << src, fill with X, shift to C
6	SHR	src, dst	dst >> src, fill with C, shift to X
7	SWAP	src, dst	dst := ((src << 8) & 0xFF00) | ((src >> 8) & 0xFF)

Opc	Instr	Operands	Effect
8	NOT	src, dst	dst := !src
9	AND	src, dst	dst := dst & src
A	OR	src, dst	dst := dst | src
B	XOR	src, dst	dst := dst ^ src
C	CMP	src, dst	compare src with dst
D			reserved
E	HALT		Halt the processor
E	RTI		Return from interrupt
E	INT	dst	Issue software interrupt
E	INCRB		Increment register bank addr.
E	DECRB		Decrement register bank addr.
F	ABRA	dst, [!]cond	Absolute branch
F	ASUB	dst, [!]cond	Absolut subroutine call
F	RBRA	dst, [!]cond	Relative branch
F	RSUB	dst, [!]cond	Relative subroutine call

All control instructions share the opcode `E`. The command to be executed is specified by bits `5..0` of the instruction:

Cmd bits	Command	Description
000000	HALT	Halt the processor
000001	RTI	Return from interrupt
000010	INT	Issue a software interrupt with the address supplied by the source operand
000011	INCRB	Increment the register bank address
000100	DECRB	Decrement the register bank address

The four branch and call instructions need some clarification:

- There are absolute and relative branches and subroutine calls. Absolute branches and jumps will transfer the program execution to an absolute address specified by the destination operand of the instruction. Relative instructions will transfer the program execution to the address which is the result of the sum of the current program counter R15 and the destination operand (using two's complement implements backward jumps).

- The difference between branches and subroutine calls is that branches just change the program counter, while subroutine calls will push the current program counter to a stack before performing the actual jump.

- All branches and subroutine calls are conditional jumps – they will be executed only if a certain condition is met.
- All conditions are specified in respect to the lower eight bits of the status register R14. A branch like

    ```
    ABRA dest, C
    ```

    will only be taken if the C bit of R14 is set.
- To simplify programming it is possible to negate the status register bit used as the control condition prior to its use (this will only affect the evaluation of the condition).

    ```
    ABRA dest, !C
    ```

    will only branch when the C bit is not set.
- To allow unconditional jumps, the LSB of the status register is always set!

All `src` and `dst` operands may be specified using one out of four possible addressing modes. In particular these are the following:

Mode bits	Notation	Description
00	`Rxx`	Use Rxx as operand
01	`@Rxx`	Use the memory cell addressed by the contents of Rxx as operand
10	`@Rxx++`	Use the memory cell addressed by the contents of Rxx as operand and then increment Rxx
11	`@--Rxx`	Decrement Rxx and then use the memory cell addressed by Rxx as operand

Although there is no explicit addressing mode to specify the usage of a constant as an operand, this can be realized by using R15 as the address register as the following example shows:

- Set R0 the the fixed value 0x1234 using MOVE:

  ```
  MOVE @R15++, R0
  ```

  This assumes that the memory cell following the MOVE instruction will contain the value 0x1234. Using the QNICE assembler an instruction like this can be specified as

  ```
  MOVE 0x1234, R0
  ```

  and the assembler will take care of filling the following memory cell with the proper value.

- Move the contents of R0 to R1:

  MOVE R0, R1

- Move the contents of R0 to the memory cell addressed by the contents of R1:

  MOVE R0, @R1

- Using R1 as a stack pointer, push the contents of R0 to the stack:

  MOVE R0, @--R1

- Using R1 as a stack pointer again, read the contents of the top of stack back into R0:

  MOVE @R1++, R0

- Perform an absolute jump to a subroutine at location 0x1234:

    ASUB 0x1234, 1

    - This absolute subroutine call will take place unconditionally since the 1 bit of R14 is always set.
    - In addition to this the contents of the program counter R15 will be pushed to a stack using R13 as the stack pointer.

- To return from this subroutine it is only necessary to read the old contents of R15, which have been pushed to the stack, back into R15:

    MOVE @R13++, R15

The following examples may help in understanding the binary representation of QNICE instructions:

Instruction	Binary representation						Hex
MOVE @--R13, R15	0000	1101	11	1111	10		0x0DFC
ADD R0, @R1	0001	0000	00	0001	01		0x1005
ASUB 0x1234, 1	1111	1111	10	01	0	000	0xFF90
	0001	0010	0011	0100			0x1234

```
000001                                    .ORG   0x0000
000002  0000  B000                XOR    R0, R0
000003  0001  0F84  1000          MOVE   0x1000, R1
000004  0003  1100          LOOP  ADD    R1, R0
000005  0004  3F84  0001          SUB    0x0001, R1
000006  0006  FF8B  0003          ABRA   LOOP, !Z
000007  0008  E000                HALT
```

- Most processors require the explicit backup of register contents at the begin of a subroutine as well as a corresponding restore at the end of the routine. This normally involves the use of a stack which is time consuming due to the necessary memory references.

- QNICE simplifies the backup and restore of registers by utilizing the 256 register bank entries corresponding to the lower eight registers R0...R7.

- A normal subroutine for QNICE will use R13 as stack pointer for storing the return address, R14 to control the register bank, R8...R12 for passing arguments to the routine and R0...R7 as working registers for the subrouine itself.

```
           MOVE ..., R8        ! Setup subroutine parameters
           ...
           RSUB ROUTINE, 1     ! Unconditionally jump to the subroutine
           ...                 ! Continue with main program
ROUTINE:   ADD 0x0100, R14     ! Incr. the register bank pointer
           ...                 ! Perform subroutine operations
           SUB 0x0100, R14     ! Restore the register bank
           MOVE @R13++, R15    ! Return to the calling program
```

QNICE supports a simple interrupt scheme:

- Interrupts cannot be nested. If an interrupt occurs while another interrupt is serviced, the CPU will not grant the new interrupt.

- The processor has two sets of the registers R14 (status register) and R15 (program counter). When an interrupt is issued, the current contents of these two registers are saved into two invisible latches. An interrupt service routine (*ISR*) can only be left by the control instruction RTI. This effectively restores the values of R14 and R15 and thus resumes operation at the correct location.

- There is an extremely simple (optional) interrupt controller implemented as an external device which allows to mask interrupts using a simple register and an AND-gate.

- The CPU features two lines to deal with external interrupts:
    - /INT: By setting this line to low, an external device requests an interrupt.
    - /IGRANT: As soon as the CPU is able to service the interrupt, it will save R14 and R15 and then signals the device by pulling /IGRANT low that the device may now put the address of the desired ISR onto the data bus.
      When the data is valid, the device pulls /INT high again. The CPU now reads the data and pulls /IGRANT high to notify the device that it must release the data bus. The CPU then jumps to this address.

- A software interrupt is triggered by INT <dst op>. The source operand contains the address of the ISR.

- To return from an interrupt the instruction RTI is used.

Bernd Ulmann
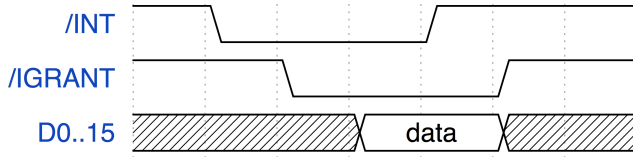
- The development suite for QNICE is available at
  `http://qnice.sf.net`. It consists of an assembler and an
  emulator.

- The sources for Mirko Holzer's FPGA-implementation of
  QNICE can be found at
  `https://github.com/sy2002/QNICE-FPGA`. This repository
  also contains a copy of the necessary QNICE software
  development utilities.

- Dr. CHRIS GILES currently (as of 2020) develops a
  microprogrammed TTL-implementation of the
  QNICE-architecture including interrupt handling.

- The emulator is quite capable and not only implements the CPU but also some IO-devices such as a basic UART and an IDE-interface.
- It features a rich command set (CB, DEBUG, DIS, DUMP, HELP, LOAD, QUIT, RESET, RDUMP, RUN, SET, SAVE, SB, STAT, STEP, VERBOSE) and extensive statistical features which proved rather useful during the design and development of the instruction set and addressing modes.

Load and disassemble the summation program:

```
Q> load sum.bin
Q> dis 0,9
Disassembled contents of memory locations 0000 - 0009:
0000: B000 XOR      R00, R00
0001: 0F84 MOVE     0x1000, R01
0002: 1000
0003: 1100 ADD      R01, R00
0004: 3F84 SUB      0x0001, R01
0005: 0001
0006: FF8B ABRA     0x0003, !Z
0007: 0003
0008: E000 HALT
0009: 0000 MOVE     R00, R00
```

Show the register contents:

```
Q> rdump
Register dump:  BANK = 00, SR = _____1

R00-R04:  0000 0000 0000 0000
R04-R08:  0000 0000 0000 0000
R08-R0c:  0000 0000 0000 0000
R0c-R10:  0000 0000 0001 0000
```

Run the program and repeat the register dump:

```
Q> run
Q> rdump
Register dump: BANK = 00, SR = ____Z__1

    R00-R03: 0800 0000 0000 0000
    R04-R07: 0000 0000 0000 0000
    R08-R11: 0000 0000 0000 0000
    R12-R15: 0000 0000 0009 0009
```

Print the statistics of this run:

```
Q> stat

20484 memory reads, 0 memory writes and
12291 instructions have been executed so far:

INSTR ABSOLUTE RELATIVE INSTR ABSOLUTE RELATIVE
------------------------------------------------
MOVE:      1 ( 0.01%) ADD :   4096 (33.33%)
ADDC:      0 ( 0.00%) SUB :   4096 (33.33%)
SUBC:      0 ( 0.00%) SHL :      0 ( 0.00%)
SHR :      0 ( 0.00%) SWAP:      0 ( 0.00%)
NOT :      0 ( 0.00%) AND :      0 ( 0.00%)
OR  :      0 ( 0.00%) XOR :      1 ( 0.01%)
CMP :      0 ( 0.00%)     :      0 ( 0.00%)
HALT:      1 ( 0.01%) ABRA:   4096 (33.33%)
ASUB:      0 ( 0.00%) RBRA:      0 ( 0.00%)
RSUB:      0 ( 0.00%)

      READ ACCESSES                 WRITE ACCESSES
MODE   ABSOLUTE RELATIVE     MODE   ABSOLUTE RELATIVE
----------------------------------------------------------
rx   :  12290 (42.86%)       rx   :   8194 (28.57%)
@rx  :      0 ( 0.00%)       @rx  :      0 ( 0.00%)
@rx++:   8193 (28.57%)       @rx++:      0 ( 0.00%)
@--rx:      0 ( 0.00%)       @--rx:      0 ( 0.00%)
```

- As of now (AUG-2015) three QNICE implementations exist: The emulator, an AVR-implementation and Mirko Holzer's FPGA-implementation.

- The next step is to develop a useful monitor program, maybe in conjunction with a Forth interpreter.

- Next, a system controller will be designed which implements an interrupt controller and a simple MMU.

- With these prerequisites a UNIX-like operating system for QNICE should be possible.

📄 Bill Buzbee's Magic Processor has been described extensively at http://www.homebrewcpu.com

📄 Bernd Ulmann, *NICE – an elegant and powerful 32-bit architecture*, Computer Architecture News, OCT-1997.

📄 Bernd Ulmann, *The NICE Processor Pages*, http://www.vaxman.de/projects/nice/nice.html