QNICE
a nice 16 bit architecture

Bernd Ulmann
ulmann@vaxman.de

DEC-2006

Goal

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.

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

Basics

- 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...7 is actually a window to a high speed RAM so in fact there are $256 \cdot 8 + 8 = 2056$ registers all in all
- moving the register window is accomplished in a single operation making push/pop operations virtually unnecessary
- Very small instruction set (17 instructions)
- 4 addressing modes

Registers

- 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
R14:	Statusregister (sr for short)
R15:	Program counter

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

The status register R14

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	M	I	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
I:	1 if an interrupt occured
M:	If set to 1, maskable interrupts are allowed

- 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.

Input/Ouput

- 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 1k word of memory is reserved for I/O controllers which can be easily accessed using normal instructions with addressing modes referring to memory cells.

Instruction set basics

- QNICE utilizes 17 basic instructions, all of which (apart from `HALT`, which takes no operands) 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`).

Instruction format

- 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`	`dst rxx`	`dst mode`	`src rxx`	`src mode`

- The four jump and branch instructions use the following instruction format:

4 bit	4 bit	2 bit	2 bit	1 bit	3 bit
`opcode`	`dst rxx`	`dst mode`	`mode`	`negate condition`	`select condition`

- The four jumps and branches `ABRA`, `ASUB`, `RBRA` and `RSUB` have the corresponding mode bits 00, 01, 10 and 11 respectively.

List of instructions

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

Opc	Instr	Operands	Effect
9	NOT	src, dst	dst := !src
A	AND	src, dst	dst := dst & src
B	OR	src, dst	dst := dst | src
C	XOR	src, dst	dst := dst ^ src
D	ABRA	dest, [!]cond	Absolute branch
D	ASUB	dest, [!]cond	Absolut subroutine call
D	RBRA	dest, [!]cond	Relative branch
D	RSUB	dest, [!]cond	Relative subroutine call

Branches and subroutine calls

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!

Addressing modes

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

Using constant operands

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.

Examples of the addressing modes

- 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
```

Examples of branches and subroutine calls

- 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
  ```

Examples of binary coded instructions

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

Instruction	Binary representation						Hex
`MOVE @--R13, R15`	0001	1111	00	1101	11		0x1F37
`ADD R0, @R1`	0010	0001	01	0000	00		0x2140
`ASUB 0x1234, 1`	1101	1111	10	01	0	000	0xDF90
	0001	0010	0011	0100			0x1234

Code example: $\sum_{i=0}^{0x1000} i$

```
0x0000 0x1BC0        MOVE 0x0000, R0
0x0001 0x0000
0x0002 0x1BC1        MOVE 0x1000, R1
0x0003 0x1000
0x0004 0x2040 LOOP   ADD R1, R0
0x0005 0x4BC1        SUB 0x0001, R1
0x0006 0x0001
0x0007 0xDBCB        ABRA LOOP, !Z
0x0008 0x0004
0x0009 0x0000        HALT
```

Subroutines

- 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.

Typical subroutine structure

```
          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
```

The assembler

- Thanks to Thomas Kratz there exists a Perl based assembler that is capable of reading QNICE assembler source files and produces binary load files as well as corresponding listing files.
- The assembler as well as all other QNICE related information and files is available at `http://www.vaxman.de/projects/qnice/qnice.html`

The emulator

- Currently a simple C written emulator exists as a proof of concept.
- The emulator is available as source code at `http://www.vaxman.de/qnice/qnice.html`
- The emulator features a rich command set (`DEBUG`, `DIS`, `DUMP`, `HELP`, `LOAD`, `QUIT`, `RESET`, `RDUMP`, `RUN`, `SET`, `SAVE`, `STAT`, `STEP`, `VERBOSE`) and extensive statistical features which proved rather useful during the design and development of the instruction set and addressing modes.

Using the emulator: Run the summation program

Load and disassemble the summation program:

```
Q> load sum.bin
Q> dis 0,9
Disassembled contents of memory locations 0000 - 0009:
0000:  1BC0 MOVE 0x0000, R00
0001:  0000
0002:  1BC1 MOVE 0x1000, R01
0003:  1000
0004:  2040 ADD R01, R00
0005:  4BC1 SUB 0x0001, R01
0006:  0001
0007:  DBCB ABRA 0x0004, !Z
0008:  0004
0009:  0000 HALT
```

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-R04:  0800 0000 0000 0000
R04-R08:  0000 0000 0000 0000
R08-R0c:  0000 0000 0000 0000
R0c-R10:  0000 0000 0009 000a
```

Print the statistics of this run:

```
Q> stat

12291 instructions have been executed so far:

 INSTR    ABSOLUTE    RELATIVE    INSTR    ABSOLUTE    RELATIVE
 HALT:           1    ( 0.01%)    MOVE:           2    ( 0.02%)
 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 :           0    ( 0.00%)    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  :       12288    (42.85%)     rx  :        8194    (28.57%)
 @rx :           0    ( 0.00%)     @rx :           0    ( 0.00%)
 @rx++:       8194    (28.57%)     @rx++:          0    ( 0.00%)
 @--rx:          0    ( 0.00%)     @--rx:          0    ( 0.00%)
```

Future plans

- The emulator has shown the power of the QNICE instruction set and its four addressing modes.
- The register bank feature is most useful in subroutines and saves lots of memory accesses for saving and restoring register contents.
- The features described in these slides can be assumed as being fixed and may serve as the basis for hardware implementations of QNICE.
- The following months will see a TTL based implementation of QNICE as well as maybe a FPGA based implementation.

Bibliography

- 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`