This article was published in an Elsevier journal. The attached copy is furnished to the author for non-commercial research and education use, including for instruction at the author’s institution, sharing with colleagues and providing to institution administration.

Other uses, including reproduction and distribution, or selling or licensing copies, or posting to personal, institutional or third party websites are prohibited.

In most cases authors are permitted to post their version of the article (e.g. in Word or Tex form) to their personal website or institutional repository. Authors requiring further information regarding Elsevier’s archiving and manuscript policies are encouraged to visit:

http://www.elsevier.com/copyright
The QC-2 parallel Queue processor architecture

Ben A. Abderazek*, Arquimedes Canedo, Tsutomu Yoshinaga, Masahiro Sowa

Parallel and Distributed Systems Laboratory, Graduate School of Information Systems, National University of Electro-Communications, 1-5-1 Chofu-shi, 1828585 Tokyo, Japan

Received 16 June 2006; received in revised form 6 August 2007; accepted 24 August 2007
Available online 6 September 2007

Abstract

Queue based instruction set architecture processor offers an attractive option in the design of embedded systems. In our previous work, we proposed a novel queue processor architecture as a starting point for hardware/software design space exploration for embedded applications. In this paper, we present a high performance 32-bit Synthesizable QueueCore (QC-2)—an improved and optimized version of the produced order parallel Queue processor (PQP), with single precision floating-point support. The QC-2 core also implements a novel technique used to extend immediate values and memory instruction offsets that were otherwise not representable because of bit-width constraints in the PQP processor. A prototype implementation is produced by synthesizing the high-level model for a target FPGA device. We present the architecture description and design results in a fair amount of details.
© 2007 Elsevier Inc. All rights reserved.

Keywords: Queue computing; Queue processor; Parallel; Design; Circular queue-register

1. Introduction

Since conventional processors are already capable of starting one operation per cycle, reducing CPI further requires starting more than one operation per cycle. To extract ILP, these processors keep many in-flight instructions, use dynamic scheduling, and register renaming [34]. As results, hardware complexity, power dissipation, and resource underutilization in these architectures become critical performance limiting factors [17,38]. There are many efforts in architectures design that address these problems. Thus, computer architects are continuously challenged to bring innovations to design microarchitectures, instruction set, and compilers, which help to keep the balance between performance, complexity, and power. Several processors have achieved success as two- or four-way Superscalar implementations. However, adding still more functional units is not useful if the rest of the processor is not capable of supplying those functional units with continuous and independent instructions to perform.

A wish to have simple but still fast machine pushed us to look for alternatives. Our research was inspired by several original ideas [1,30,15,19], which proposed the usage of queue (first-in-first-out memory) instead of registers (random access memory) as intermediate storage of results. In these architectures, each instruction removes the required amount of data from the head of operand queue (GERG) and then stores the result of computation at the tail of operand queue.

In [31] it was also argued that a queue machine application can be easily mapped to an appropriate hardware. However, no real hardware was proposed or designed and only theoretical techniques were proposed to virtualize the hardware.

We proposed a produced order parallel Queue processor (PQP) [1,3,4,5,35]. The key ideas of the produced order queue execution model are the operands and results manipulation schemes. The Queue execution model stores intermediate results in a circular queue-register (QREG). A given instruction implicitly reads its first operand from the head (QH) of the QREG, its second operand from a location explicitly addressed with an offset from the first operand location. The computed result is finally written into the QREG at a position pointed by a queue tail pointer (QT).

The PQP processor has several promising advantages over register-based machines. First, PQP programs have higher
instruction level parallelism because they are constructed using breadth-first algorithm [1]. Second, instructions of PQP are shorter because they do not need to specify operands explicitly [2]. That is, data are implicitly taken from the head of operand queue and the result is implicitly written at the tail of the operand queue. This makes instruction lengths shorter and independent from the actual number of physical queue words. Finally, PQP instructions are free from false dependencies. This eliminates the need for register renaming [1].

In this paper, we present an optimized and improved architecture, named QC-2 core, of the PQP processor. The QC-2 core implements all hardware features found in PQP core, supports single precision floating-point accelerator (FPA), and uses QCaEXT scheme—a novel technique used to extend immediate values and memory instruction offsets that were otherwise not representable because of bit-width constraints in the PQP processor. The aim of the QCaEXT technique is to achieve code density that is similar to the PQP code with performance similar to architecture set on 32-bit memory. Moreover, the QC-2 core implements new instructions for controlling the QREG unit (discussed later). We consider the well-known prototyping-based emulation approach, that substitutes real time hardware emulation for slow simulator-based execution [27,37]. To achieve good synthesis results for FPGAs implementation with sufficient performance, we have created the synthesizable model of the QC-2 processor for the integer and floating subset of the parallel Queue processor instruction set architecture [35,1,5]. A prototype implementation is produced by synthesizing the high-level model for the Stratix FPGA device [26,7]. As a result, we were able to evaluate performance and relative circuit area, speed, and power. We also describe a quantitative study of the impact of synthesis and optimization methods on performance and design quality achieved for QC-2 processor. The rest of this paper is organized as follows: Section 2 reviews the produced order Queue computation model. In Section 3, we describe the QC-2 core architecture. Section 4 gives the QC-2 synthesis approach. In Section 5, we present the design results and discussions. The last section gives our conclusion.

2. Produced order Queue computing overview

As mentioned, the produced order Queue computing model uses a circular queue-register (also named operand queue) instead of random access registers to store intermediate results. A data is inserted in the QREG in produced order scheme and can be reused. This feature has a profound implication in the areas of parallel execution, program compactness, hardware simplicity, and high execution speed [35,1]. This section gives a brief overview about the produced order Queue computation model. We show in Fig. 1(a) a sample data flow graph for the expressions:

\[ e = \frac{ab}{c} \quad \text{and} \quad f = ab(c + d) \]

Datum is loaded with load instruction (ld), computed with multiply (mul), add (+), and divide (/) instructions. The result is stored back in the data memory with store instruction (st).

In Fig. 1(a), producer and consumer nodes are shown. For example, A2 is a producer node and A4 is a consumer node (A4 is also a producer to m6 node). The instruction sequence for the queue execution model is correctly generated when we traverse the data flow graph (shown in Fig. 1(a)) from left to right and from the highest to the lowest level [1].

In Fig. 1(b), the augmented data flow graph that can be correctly executed in the proposed queue execution model is given. The generated instruction sequence from the augmented graph is shown in Fig. 1(c) and the content of the QREG at each execution stage is shown in Fig. 1(d).

Fig. 1. Sample data flow graph and queue-register contents for the expressions:

\[ e = \frac{ab}{c} \quad \text{and} \quad f = ab(c + d) \]: (a) original sample program, (b) translated (augmented) sample program, (c) generated instructions sequence, (d) circular queue-register content at each execution stage.
is stored. A live queue head pointer (LQH) is also used to keep used data that could be reused and thus should not be overwritten. These data, which are found between QH and LQH pointers, are called live-data (discussed later). The live-data entries in the QREG are statically controlled. Two special instructions are used to stop or release the LQH pointer. Immediately after using the data, the QH is incremented so that it points to the data for the next instruction. QT is also incremented after the result is stored. The four load instructions load in parallel $a$, $b$, $c$, and $d$ data and place them into the QREG. At this state, QH and LQH point to datum $a$ and the QT points to an empty location as shown in Fig. 1(d) (State 1). The fifth and sixth instructions are also executed in parallel. The $mul$ refers $a$ and $b$ then inserts $(a \times b)$ into the QREG. QH is incremented by 2 and the QT is incremented by 1. The $add$ refers $c$ and $d$ then inserts $(c + d)$ into the QREG. At this state, the QH, QT, and LQH are incremented as shown in Fig. 1(d) (State 2). The seventh instruction $(div - 2)$ divides the data pointed by QH (in this case $(a \times b)$) by the data located at $-2$, negative offset, from QH (in this case $c$). The QH is increased and points to $(c + d)$. The eighth instruction multiplies the data pointed by QH (in this case $(c + d)$) with the data located at $-1$ from QH (in this case $(a \times b)$). After parallel execution of these two instructions, the QREG content becomes as shown in Fig. 1(d) (State 3). The last two instructions store back the result in the data memory. Since the QREG becomes empty, LQH, QH, and QT point to the same empty location (State 4).

3. QC-2 core architecture

3.1. Instruction set design considerations

The QC-2 supports a subset of the produced order queue processor instruction set architecture [5]. All instructions are 16-bit wide, allowing simple instructions fetch and decode stages and facilitating pipelining of the processor. The QC-2 integer instruction format is illustrated in Fig. 2. Several examples showing the operations of five different instructions are also given in Fig. 2(a)-(e). In the current version of our implementation, we target the QC-2 core for small applications where our concerns focus on the ability to execute Queue programs on a processor core with small die size and low power consumption characteristics when compared to other conventional 32-bit architectures. However, the short instructions may limit the memory addressing space as only 8-bit are left for offset (6-bit) and base address (2-bit—00:a0/d0, 01:a1/d1, 10:a2/d2, and 11:a3/d3). To cope with this shortage, QC-2 core implements QCAEXT technique, which uses a special “covop” instruction that extends load and store instructions offsets and also extends immediate values if necessary. The Queue processor compiler
Fig. 3. QC-2 architecture block diagram. During RTL description, the core is broken into small and manageable modules using modular approach structure for easy verification, debugging, and modification.

[12] outputs full addresses and full constants and it is the duty of the QC-2 assembler to detect and insert a “covop” instruction whenever an address or a constant exceeds the limit imposed by the instruction’s field sizes. Conditional branches are handled in a particular way since the compiler does not handle target addresses, instead it generates target labels. When the assembler detects a target label, it looks if the label has been previously read and fills the instruction with the corresponding value and “covop” instruction if needed. There is a back-patch pass in the assembler to resolve all missing forward referenced instructions [12].

3.2. Instruction pipeline structure

The execution pipeline operates in six stages combined with five pipeline-buffers to smoothen the flow of instructions through the pipeline. The QC-2 block diagram is shown in Fig. 3. Data dependencies between instructions are automatically handled by hardware interlocks. Below we describe the salient characteristics of the QC-2 core.

(1) Fetch (FU): The instruction pipeline begins with the fetch stage, which delivers four instructions to the decode unit each cycle. This is the same bandwidth as the maximum execution rate of the functional units. At the beginning of each cycle, assuming no pipeline stalls or memory wait states occur, the address pointer hardware (APH) of the fetched instructions issues a new address to the memory system. This address is the previous address plus 8 bytes or the target address of the currently executing flow-control instruction.

(2) Decode (DU): The DU decodes four instructions in parallel during the second phase and writes them into the decode buffer. This stage also calculates the number of consumed (CNBR) and produced (PNBR) data for each instruction [35]. The CNBR and PNBR are used by the next pipeline stage to calculate the sources (source1 and source2) and destination locations for each instruction. Decoding stops if the queue buffer becomes full or/and a halt signal is received from one or more stages following the decode stage.

(3) Queue computation (QCU): Four instructions arrive at the QCU unit each cycle. The QCU calculates the first operand (source1) and destination addresses for each instruction. The mechanism used for calculating the source1 address is given in Fig. 4. The QCU unit keeps track of the current value of the QH and QT pointers.

(4) Barrier: Inserts barrier flags for dependency resolutions.

(5) Issue (IS): Four instructions are issued for execution each cycle. In this stage, the second operand (source2) of a given instruction is first calculated by adding the address source1 to the displacement that comes with the instruction. The second operand address calculation is performed in the QCU stage. However, for a balanced pipeline consideration, the source2 is calculated at the beginning of the IS stage. The hardware mechanism used for calculating the source2 address is shown in Fig. 5 (discussed later).

An instruction is ready to be issued if its data and its corresponding functional unit are available. The processor reads
3.3. Dynamic operands addresses calculation

To execute instructions in parallel, the QC-2 processor must calculate each instruction's operand(s) and destination addresses dynamically. As a result, the "static" Queue data structure (compiler point of view) is regarded dynamically as a circular queue-register structure. Figs. 4 and 5 show block diagrams of the hardware used for calculating source1, destination and source2, respectively. To calculate the source1 address of a given instruction, the number of consumed data (CNBR) field is added to the current queue head value ($QH_n$).

The destination address on the next instruction ($INST_n + 1$) is calculated by adding the PNBR field (8-bit) to the current queue tail value ($QT_n$). Notice that the calculation is performed sequentially. Each QREG entry is written exactly once and it is busy until it is written. If a subsequent instruction needs its value, that instruction must wait until requested data is written. After a given entry in the QREG is written, the corresponding data in the above entry is ready and its ready bit (RDB) is set.

3.4. QC-2 FPA organization

The QC-2 FPA is a pipelined structure and implements a subset of the IEEE-754 single precision floating-point standard [20,21]. The FPA consists of a floating-point ALU (FALU), floating-point multiplier (FMUL), and floating-point divider (FDIV). The FALU, FMUL, FDIV and the floating-point queue-register employ 32-wide data paths. Most FPA operations are completed within three execution cycles. The FPA's execution pipelines are simple in design for high speed that the QC-2 core requires. All frequently used operations are directly implemented in the hardware. The FPA unit supports the four rounding modes specified in the IEEE 754 floating-point standard: round toward-to-nearest-even, round toward positive infinity, round toward negative infinity, and round toward zero.

3.4.1. Floating-point ALU implementation

The FALU does floating-point addition, subtraction, compare and conversion operations. Its first stage subtracts the operands exponents (for comparison), selects the larger operand, and aligns the smaller mantissa. The second stage adds or subtracts the mantissas depending on the operation and the signs of the operands. The result of this operation may overflow by a maximum of 1-bit position. Logic embedded in the mantissa adder is used to detect this case, allowing 1-bit normalization of the result on the fly. The exponent data path computes ($E + 1$). If the 1-bit overflow occurred, ($E + 1$) is chosen as the exponent of stage 3; otherwise, $E$ is chosen. The third stage performs either rounding or normalization because these operations are not required at the same time. This may also result in a 1-bit overflow. Mantissa and exponent corrections, if needed, are implemented exactly in this stage, using instantiations of the mantissa adder and exponent blocks. The area efficient FALU hardware is shown in Fig. 6. The exponents of the two inputs (Exponent A and Exponent B) are fed into the exponent comparator, which is implemented with a subtractor and a multiplexer. In the pre-shifter, a new mantissa is created by right
We have to notice that the LD anticipator can also be predicted. Tissas are multiplied and the exponents are added. The output is the mantissa of the operands. In the second stage, the normalization module used in addition to insert the implied one. The normalizer transforms the mantissa and exponent into normalized format. It first uses a leading-one detector (LD) circuit to locate the position of the most significant one in the mantissa. Based on the position of LD, the resulting mantissa is left shifted by an amount subsequently deducted from the exponent. If there is an exponent overflow (during normalization), the result is saturated in the direction of overflow and the overflow flag is set. Underflows are handled by setting the result to zero and setting an underflow flag.

3.4.2. Floating-point multiplier implementation

Fig. 7 shows the data path of the FMUL unit. As with other conventional architectures, QC-2’s FMUL operation is much like integer multiplication. Because floating-point numbers are stored in sign-magnitude form, the multiplier needs only to deal with unsigned integer numbers and normalization. Similar to the FALU, the FMUL unit is a three-stage pipeline that produces the result on every clock cycle. The bottleneck of this unit was the 24 * 24 integer multiplications.

The first stage of the floating-point multiplier is the same de-normalization module used in addition to insert the implied one to the mantissa of the operands. In the second stage, the mantissas are multiplied and the exponents are added. The output of the module is registered. In the third stage, the result is normalized or rounded.

The multiplication hardware implements the radix-8 modified Booth [10] algorithm. Recoding in a higher radix was necessary to speed up the standard Booth multiplications algorithm since greater numbers of bits are inspected and eliminated during each cycle. It effectively reduces the total number of cycles necessary to obtain the product. In addition, the radix-8 version was implemented instead of the radix-4 version because it reduces the multiply array in stage 2.

3.5. Circular queue-register structure

The QREG structure is shown in Fig. 8. For clarity, only the first 16 entries (ENTRY0 to ENTRY15) are shown. Fig. 8(a) shows the QREG initial state. In this state, the QREG is empty and the QH, QT, and LQH pointers point to the same location (QREG logical entry 0). When the first data is written into the QREG storage, the QT is incremented by 1. Since no data is consumed yet, the QH and LQH still point to the initial location. This scenario is shown in Fig. 8(b). The QH pointer is incremented by 1 after data 1 (dat1) is consumed as shown in Fig. 8(c). Because dat1 maybe reused again, the LQH pointer still points to ENTRY0, which holds data 1 (dat1). A special instruction, named stplqh (stop LQH) was implemented to stop the automatic movement of LQH. The automatic LQH movement (default setting) is restored with another special instruction, named autlqh (automatic LQH). In some situation the QREG storage may have three types of entries as shown in Fig. 8(d). These entries are: dead entries—data is no longer needed, live entries—data can be reused, and (3) empty entries—no data in these entries.

4. Synthesis of the QC-2 core

4.1. Design approach

To make the QC-2 design easy to debug, modify, and adapt, we decided to use a high-level description, which was also used by other system designers, such as works in [14,33,32,23,27,37]. We have developed the QC-2 core in Verilog HDL. After synthesizing the HDL code, the designed processor has characteristics that enable investigation of the actual hardware performance and functional correctness. It also gives us the possibility to study the effect of coding style and instruction set architectures over various optimizations. For the QC-2 processor to be useful for these purposes, we identified the following requirements: (1) High-level description: the format of the QC-2 description should be easy to understand and modify; (2) Modular: to add or remove new instructions, only the relevant parts should have to be modified. A monolithic design would make experiments difficult; and (3) the processor description should be synthesizable to derive actual implementations.

The QC-2 has been designed with a distributed controller to facilitate debugging and future adaptation for specific application requirements since we target embedded applications. This
distributed controller approach replaces a monolithic controller which would be difficult to adapt. The distributed controller is responsible for pipeline flow management and consists of communicating state machines found in each pipeline.

In this design, we have decided to break up the unstructured control unit to small, manageable units. Each unit is described in a separate HDL module. That is, instead of a centralized control unit, the control unit is integrated with the pipeline data path. Thus, each pipeline stage is mainly controlled by its own simple control unit. In this scheme, each distributed state machine corresponds to exactly one pipeline stage, and this stage is controlled exclusively by its corresponding state machine. Overall flow control of the QC-2 processor is implemented by cooperation of the control units in each stage based on communicating state machines. Each pipeline stage is connected to its immediate neighbors, and indicates whether it is able to supply or accept new instructions. Communication with adjacent pipeline stages is performed using two asynchronous signals, AVAILABLE and PROCEED. When a stage has finished processing, it asserts the AVAILABLE signal to indicate that data are available to the next pipeline stage. The next stage will, then, indicate whether it can proceed these data by using the PROCEED signal.

Since all fields, necessary to find what actions are to be taken next, are available in the pipeline stage (for example operation status ready bits and synchronization signals from adjacent stages), computing the next stage is simple. The state transition of a pipeline stage in the QC-2 is illustrated in Fig. 9. This basic
state machine is extended to cover the operational requirements of each stage, by dividing the PROCEED state into sub states as needed. An example is the implementation of the Queue computation stage, where PROCEED is divided into sub states for reading initial addresses values, calculating next addresses values, and addresses fixup (when needed).

We have synthesized the QC-2 core for Stratix FPGAs and HardCopy devices with Altera Quartus II professional edition [7] tool. In order to estimate the impact of the description style on the target FPGA efficiency, we have explored logic synthesis for FPGAs. The idea of this experiment was to optimize critical design parts for speed or resource optimizations. In this work, our experiments and the results described are based on the Altera Stratix architecture [26]. We selected Stratix FPGA device because it has good tradeoffs between routability and logic capacity. In addition it has an internal embedded memory that eliminates the need for external memory module and offers up to 10 Mbits of embedded memory through the TriMatrix TM memory feature. We also used Altera Quartus II professional edition [7] for simulation, placement and routing. Simulations were also performed with Cadence Verilog-XL tool [11].

5. Results and discussions

### 5.1. Execution speedup and code analysis

Before describing the QC-2 synthesis results, we first present the execution time, speed up and programs size (binaries) evaluation results for several benchmark programs. We obtained these results by using our back-end tool (QC2ESTM) and QueueCore/QC-2 compiler [12,13]. The embedded applications are selected from MediaBench [25] and MiBench [18] suites. The selected benchmarks include two video compression applications: H.263, MPEG2; one graph processing algorithm: Susan; two encryption algorithms: AES, Blowfish; and one signal processing: FFT.

Table 1 shows the normalized code size of several benchmark programs compiled with a port of GCC 4.0.2 for every target architecture. We selected MIPS I ISA [22] as the baseline and include other three embedded RISC processors and a CISC representative. The last column shows the normalized code size for the applications compiled using the QC-2 compiler [12,13]. The table shows that the binaries for the QC-2 processor are about 70% smaller than the binaries for MIPS and about 50% smaller than ARM [28]. Compared to dual-instruction set embedded RISC processors, MIPS16 [24] and Thumb [16], QC-2 binaries are about 20% and 40% denser, respectively. When compared to the CISC architecture, Pentium processor [6], QC-2 binaries are about 14% denser. Table 2 shows the execution time in cycles for serial (PQP-S) and parallel (QC-2) architectures. The last column in the table shows the speedup of the parallel execution scheme over serial configuration. This table shows that the Queue computation model extracts natural parallelism found in programs speeding up these embedded applications by factors from 1.49 to 3.57.

### 5.2. Synthesis results

Table 3 shows the hardware configuration parameters of the designed QC-2 core. Table 4 summarizes the synthesis results of the QC-2 for the Stratix FPGA and HardCopy targets. The complexity of each module as well as the whole QC-2 core are given as the number of logic elements (LEs) for the Stratix FPGA device and as the total combinational functions (TCF) count for the HardCopy device (Structured ASIC). The design was optimized for balanced optimization guided by a properly implemented constraint table. We also found that the processor consumes about 95.3% of the total logical elements of the target device.

The achievable throughput of the 32-bit QC-2 core on different execution platforms is shown in Fig. 10. For the hardware
platforms, we show the processor frequency. For comparison purposes, the Verilog HDL simulator performance has been converted to an artificial frequency rating by dividing the simulator throughput by a cycle count of 1 CPI. This chart shows the benefits which can be derived from direct hardware execution using a prototype when compared to processor simulation. The data used for this simulation are based on event-driven functional Verilog HDL simulation [5].

The critical path of the QC-2 core with 16 registers configuration is 44.4 ns, that was 22.5 MHz of clock frequency. For QC-2 core with 256 registers, the critical path is 39.2 ns. The clock frequencies for both configurations are low due to the fact that we synthesized the processor library to random logic of standard cell. However, the performance may be much more improved by using specific layout generation tools.

Fig. 11 compares two different target implementations for $256 \times 33$ QREG for various optimizations. Depending on the target implementations device, either LEs or TCF are generated as storage elements. Implementations based on HardCopy device, which generates TCF functions give almost similar complexity for the three used optimizations—area (ARA), speed (SPD), and balanced (BLD). For FPGA implementation, the complexity for SPD optimization is about 17% and 18% higher than that for ARA and BLD optimizations, respectively.

Fig. 12 shows the floorplan of the placed and routed QC-2 core. The modules of the processor show considerable overlap as logic is mapped according to interconnect requirements.

5.3. Speed and power consumption comparison with synthesizable CPU cores

Queue computing and architecture design approaches take into account performance and power consumption considerations early in the design cycle and maintain a power-centric focus across all levels of design abstraction. In QC-2 processor, all instructions designed are fixed format 16-bit words with minimal decoding effort. As a result, the QC-2 architecture has much smaller programs than either RISC or CISC machines. As we showed in the previous section, programs sizes for our architecture are found to be 50–70% smaller than programs for conventional architectures. The importance of the system memory size translates to an emphasis on code size since data are dictated by application. Larger memories mean more power, and optimization power is often critical in embedded applications. In addition, instructions of QC-2 processor specify operands implicitly. This design decision makes instructions independent of the actual number of physical queue words (QREG). Instructions are then free from false dependencies. This feature eliminates the need for register renaming unit, which consumes about 4% of the overall on-chip power in conventional RISC processors [29,9].

Performance of QC-2 in terms of speed and power consumption is compared with various synthesizable CPU cores as illustrated in Table 5. The SH-2 is a popular Hitachi SuperH based instruction set architecture [8,36]. The SH-2 has RISC-type instruction sets and $16 \times 32$ bit general purpose registers. All instructions have 16-bits fixed length. The SH-2 is based on five stages pipelined architecture, so basic instructions are executed in one clock cycle pitch. Similar to our QC-2 core, the SH-2 also has an internal 32-bit architecture for enhanced data processing ability. LEON2 is a SPARC V8 compliant 32-bit RISC processor. The power consumption values are based on Synopsis software based on reasonable input activities. ARM7 is a simple 32-bit RISC processor and the power consumption values are manufacturer given for hard core. The MicroBlaze core is a 32-bit soft processor. It features a RISC architecture with Harvard-style, separate 32-bit instruction and data buses [39]. From the result shown in Table 3, the QC-2

<table>
<thead>
<tr>
<th>Items</th>
<th>Configuration</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IW</td>
<td>16-bit</td>
<td>Instruction width</td>
</tr>
<tr>
<td>FW</td>
<td>8 bytes</td>
<td>Fetch width</td>
</tr>
<tr>
<td>DW</td>
<td>8 bytes</td>
<td>Decode width</td>
</tr>
<tr>
<td>SI</td>
<td>85</td>
<td>Supported instructions</td>
</tr>
<tr>
<td>QREG</td>
<td>256</td>
<td>Circular queue-register</td>
</tr>
<tr>
<td>ALU</td>
<td>4</td>
<td>Arithmetic logical unit</td>
</tr>
<tr>
<td>LD/ST</td>
<td>2</td>
<td>Load/store unit</td>
</tr>
<tr>
<td>BRAN</td>
<td>1</td>
<td>Branch unit</td>
</tr>
<tr>
<td>SET</td>
<td>4</td>
<td>Set unit</td>
</tr>
<tr>
<td>MUL</td>
<td>1</td>
<td>Multiply unit</td>
</tr>
<tr>
<td>FPU</td>
<td>2</td>
<td>Floating-point unit</td>
</tr>
<tr>
<td>GPR</td>
<td>16</td>
<td>General purpose registers</td>
</tr>
<tr>
<td>MEM</td>
<td>2048 word</td>
<td>PROG/DATA memory</td>
</tr>
</tbody>
</table>

Fig. 11. Resource usage and timing for $256 \times 33$ bit QREG unit for different coding and optimization strategies.

Fig. 12. Floorplan of the placed and routed QC-2 core.
ARM7 (hard core) processors. The QC-2 has higher speed for various Synthesizable CPU cores over speed (SPD) and area (ARA) optimizations.

Table 4
QC-2 processor design results: modules complexity as LE (logic elements) and TCF (total combinational functions) when synthesized for FPGAs (with Stratix device) and Structured ASIC (HardCopy II) families.

<table>
<thead>
<tr>
<th>Descriptions</th>
<th>Modules</th>
<th>LE</th>
<th>TCF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction fetch unit</td>
<td>IF</td>
<td>633</td>
<td>414</td>
</tr>
<tr>
<td>Instruction decode unit</td>
<td>ID</td>
<td>2573</td>
<td>1564</td>
</tr>
<tr>
<td>Queue compute unit</td>
<td>QCU</td>
<td>1949</td>
<td>1304</td>
</tr>
<tr>
<td>Barrier queue unit</td>
<td>BQU</td>
<td>9450</td>
<td>4348</td>
</tr>
<tr>
<td>Issue unit</td>
<td>IS</td>
<td>15476</td>
<td>7065</td>
</tr>
<tr>
<td>Execution unit</td>
<td>EXE</td>
<td>7868</td>
<td>3241</td>
</tr>
<tr>
<td>Queue-register unit</td>
<td>QREG</td>
<td>35541</td>
<td>21190</td>
</tr>
<tr>
<td>Memory access</td>
<td>MEM</td>
<td>4158</td>
<td>3436</td>
</tr>
<tr>
<td>Control unit</td>
<td>CTR</td>
<td>171</td>
<td>152</td>
</tr>
<tr>
<td>QC-2 core</td>
<td>QC-2</td>
<td>77819</td>
<td>42714</td>
</tr>
</tbody>
</table>

Table 5
Speed and power consumption comparisons for various Synthesizable CPU cores over speed (SPD) and area (ARA) optimizations.

<table>
<thead>
<tr>
<th>Cores</th>
<th>Speed (SPD)</th>
<th>Speed (ARA)</th>
<th>Average power (mW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PQP</td>
<td>22.5</td>
<td>21.5</td>
<td>120</td>
</tr>
<tr>
<td>SH-2</td>
<td>15.3</td>
<td>14.1</td>
<td>187.5</td>
</tr>
<tr>
<td>ARM7</td>
<td>25.2</td>
<td>24.5</td>
<td>22</td>
</tr>
<tr>
<td>LEON2</td>
<td>27.5</td>
<td>26.7</td>
<td>458</td>
</tr>
<tr>
<td>MicroBlaze</td>
<td>26.7</td>
<td>26.7</td>
<td>135</td>
</tr>
<tr>
<td>QC-2</td>
<td>25.5</td>
<td>24.2</td>
<td>90</td>
</tr>
</tbody>
</table>

This evaluation was performed under the following constraints: (1) Family: Stratix; (2) Device: EP1S25F1020; (3) Speed: C6. The speed is given in MHz.

 QC-2 core shows better speed performance for both area and speed optimizations when compared with SH-2, PQP, and ARM7 (hard core) processors. The QC-2 has higher speed for both SPD and ARA optimizations when compared with SH-2 processor (about 40% for speed optimization and 41.73% for area optimization). QC-2 core also shows 25% less power consumption when compared with PQP and consumes less power than SH-2, LEON2, and MicroBlaze cores.

6. Conclusions

In this research work we presented the architecture, design, and evaluation of a produced order queue processor with single floating-point support (QC-2) and a novel technique used to extend immediate values and memory instruction offsets. Evaluation results reveal that the QC-2 processor achieves a speed of about 25.5 and 22.5 MHz for QREG16 (QREG size is 33 * 16 entries) and QREG256 (QREG size is 33 * 256 entries), respectively. We also found that the processor consumes about 95.3% of the total logic elements (LEs) of the Stratix EP1S80BY9 device. As a result, it fits on a single Stratix device with an internal embedded memory that eliminates the need for external memory module, thereby obviating the need to perform multi-chip partitioning which results in a loss of resource efficiency. Only a few clearly identified components, such as the Barrier and the QREG units, need to be specially optimized at the HDL source level to achieve efficient resource usage. From the comparison results, we also conclude that the QC-2 processor core shows better speed performance for both area and speed optimizations when compared with SH-2, PQP, and ARM7 (hard core) processors. On average the QC-2 has about 40.87% higher speed than SH-2 processor. QC-2 core also shows 25% less power consumption when compared with PQP, and consumes less power than SH-2, LEON2, and MicroBlaze cores.

References
