Modular Design Structure and High-Level Prototyping for Novel Embedded Processor Core

Ben A. Abderazek, Sotaro Kawata, Tsutomu Yoshinaga, and Masahiro Sowa

Graduate School of Information Systems
The University of Electro-Communications
1-5-1 Chofugaoka, Chofu-shi, 182-8585 Tokyo, Japan
E-mail: ben@is.uec.ac.jp

Abstract. In this research work, we present a high-level prototyping of a new processor core based on Queue architecture as starting point for application-specific processor design exploration. Using modular design structure with control logic implemented as a set of communicating state machines, we show hardware emulation and optimizations results of a parallel queue proecssor architecture (QueueCore). We also show how to to fully exploit the capabilities of the designed QueueCore, while maintaining a common source base. From the evaluation results, we show that the QueueCore prototype fits on a single conventional FPGA device, thereby obviating the need to perform multi-chip partitioning which results in a loss of resource efficiency.

1 Introduction

In our previous work, we have researched about queue computing infrastructure as a starting point for cooperative design space exploration for general and embedded applications[1, 2]. The choice of this architecture was based on a number of factors, such as its suitability for embedded applications.
A parallel Queue processor architecture[2] exploits instruction-level parallelism without considerable effort and need for heavy run time data dependence analysis, resulting in a simple hardware organization when compared with conventional superscalar processors. These features allow the inclusion of a large number of functional units into a single chip, increasing parallelism exploitation.
The key idea of the Queue processor is the manupilation scheme of instruction operands and result. The Queue computing scheme adopted in this work stores intermediate results into a so called queue-registers (QREG). In this model, a given instruction implicitly reads its first operand from the head of the QREG, its second operand from a location explicitly addressed with an integer offset from the first operand location and stores the computed result into the tail of the QREG. An important feature of this scheme is that write after read false data dependency does not occur[1]. Furthermore, since there is no explicit referencing to the QREG, it is easy to add extra storage locations to the QREG when needed.
Since the operands and result addresses of a given static-instruction (compiler

generated) are implicitly *computed* during run-time, an efficient and fast hardware mechanism is imperatively needed for parallel execution. The proposed architecture, which is named QueueCore, implements a so named queue computation mechanism that calculates operands and result addresses for each instruction. The mechanism used for calculating these addresses will be described in the coming section. An important aspect of developing any new architecture

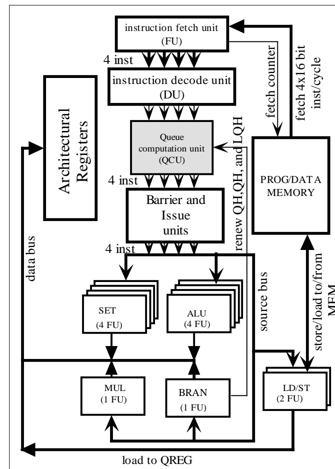

Fig. 1. QueueCore architecture block diagram. During RTL description, the core is broken into small and manageable modules using modular approach structure for easy verification, debugging and modification.

is verification which usually requires complicated and lengthly software simulation of an emulated model. An event-based or cycle level simulation becomes increasingly inadequate to verify a significant execution trace for a given problem. These software-based simulation approaches are not capable of predicting all micro-architectural issues related to final physical design[3]. To this end, we consider prototyping-based emulation that substitutes real time hardware emulation for slow simulator-based execution. Prototyping-based design itself is not a new idea as several designers already adopted this method. However, what is important for us is to investigate how to describe the novel QueueCore architecture to achieve good synthesis results for FPGA implementation with sufficient performance to support realistic evaluation.
Using a hardware description language, we have created the Synthesizable model of the QueueCore for the integer subset of parallel Queue processor (PQP) architecture[1, 2]. A prototype implementation is produced by synthesizing the high-level model for the Stratix FPGA device[4, 6]. As results, we were able to evaluate relative circuit area and speed metrics for design alternatives.

2 QueueCore Architecture

The QueueCore block diagram is shown in Fig. 1. The system has six pipeline stages:
(1) *Fetch (FU)*: at each cycle the fetch counter is sent to PROG/DATA memory and eight bytes (four instructions) are fetched then inserted into the fetch buffer (FB).
(2) *Decode (DU)*: this unit reads eight bytes from the FB every cycle, decodes them, then inserts decoded information into the decode buffer. This stage also calculates the number of consumed (CNBR) and produced (PNBR) data for each instruction. The CNBR and PNBR are used by the next pipeline stage (queue computing stage) to calculate the sources and destination locations for each instruction.
(3) *Queue computation (QCU)*: calculates the first operand ($source1$) and destination addresses for each instruction. The mechanism used for calculating the $source1$ address in given in Fig. 2. The QCU unit keeps track on the current value of the QH and QT pointers. Four instructions arrive to the QCU unit each cycle. For the first instruction, the number of the consumed operands (CNBR) (8-bit field) is added to the current QH value (QH0) to find the address of the first operand (QHN) and the number of produced results (PNBR) (8-bit filed) is added to the current QT value (QT0) to find the result address (QT1) of the first instruction. The other three instruction's first operand and result addresses are calculated similarly as indicated in the lower part of Fig. 2. Notice that the calculation is performed sequentially.
(4) *Barrier:* inserts barrier flag for dependency resolutions.
(5) *Issue:* 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 could be earlier calculated in the QCU stage. However, for timing balance consideration, the $source2$ is calculated in the IS stage. The hardware mechanism used for calculating the second operand ($source2$) address is shown in the right part of Fig. 2.
An instruction is ready to be issued if its data operands and its corresponding functional unit are available.
6) *Execution (EXE)*: the QueueCore's execution unit computes the effective address for loads and stores, and executes ALU instructions. During the EXE stage, the machine also determines whether a conditional branch is taken and, if so, updates the program counter and send the QH and QT of branch instruction to the QCU unit.

3 QueueCore Implementation

To the best of our knowledge, this is the first hardware design of a Queue processor architecture. Therefore, there are no related works in the literature that give design issues or methodology related to hardware Queue processor.

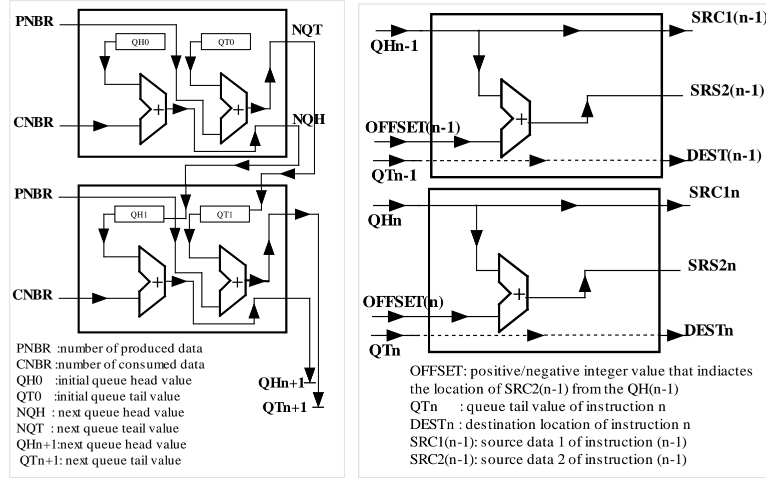

Fig. 2. Address calculation mechanism for *source*1, *destination* (left part of the figure), and source two calculation mechanism (right part of the figure).

To make the QueueCore design easy to debug, modify and adaptable, we decided to use a high-level description, which was also used by other system designers. We have developed the QueueCore system in Verilog HDL.

3.1 QueueCore System Pipeline Control

In many conventional processors, the control unit is centralized and controls all central processing core functions. This scheme introduces pipeline stalls, bubbles, etc. However, especially for pipelined architecture, this control unit is one of the most complex part of the design, even for processors with fixed functionality.

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 as illustrated in Fig. 3. 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 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 neighbours, 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 is available to the next pipeline stage. The next stage will, then, indicate whether it can proceed these data by using the *PROCEED* signal.

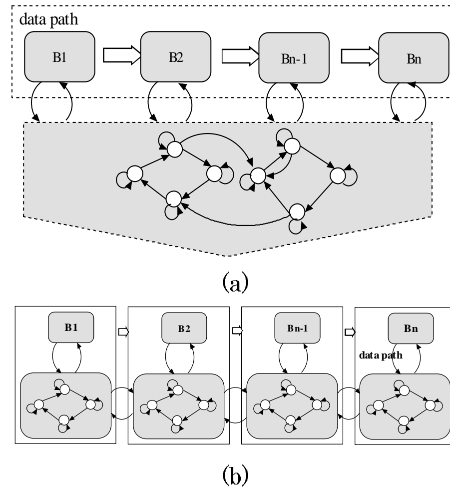

Fig. 3. Control unit: (a) In conventional Processor design, (2) in PQP processor design.

Since all field 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 adjacently stages), computing the next stage is simple. The state transition of a pipeline stage in the QueueCore is illustrated in Fig. 4. This basic state machine is extended to cover the operational requirements of each stage, by dividing the *PROCEED* state into substates as needed. An example is the implementation of the Queue computation stage, where *PROCEED* is divided into substates for reading initial addresses values, calculating next addresses values, and addresses fixup (when needed).

4 QueueCore Validation

Validation loosely refers in this paper to the process of determining that the designed processor is correct. It is performed at multiple abstraction levels by integrating several approaches performed in three phases and based on the same HDL that describe the processor core: (1)Functional verfication,(2)Rapid Prototyping, and (3)Gate Level Simulation. The QueueCore graphical validation tool (GVT) display the validated output ports when the source clock changes. A summary of the memory, QREG, the genral purpose registers (GPR) status and the instructions processing counters can be monitored with the GVT tool.

5 FPGA Synthesis of the QueueCore

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

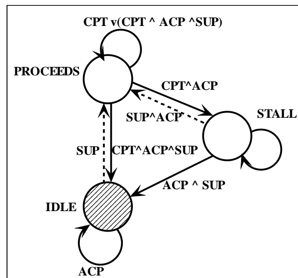

Fig. 4. Finite state machine transition for QueueCore pipeline synchronization. The following conditions are evaluated: next stage can accept data (ACP), previous pipeline stage can supply data (SUP), last cycle of computation (CPT).

was to optimize critical design parts for speed or resource optimizations. The optimizations of the Verilog HDL models for FPGAs can be arranged into two distinctive approaches: (1) Coding style and (2) Usage of special-purpose Devices on FPGA. In this work, our experiments and the results described are based on the Altera Stratix architecture [4]. We selected Stratix FPGA device because it has a 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 [6] for simulation, placement and routing. Simulations were also performed with Cadence Verilog-XL tool [5]. The quality of the final QueueCore hardware and the resource usage of the target FPGA device depend very much on the description style used at a higher level. To account for this, the high-level HDL description has to be adapted to guide the synthesis tool to choose the appropriate implementation method for functionality. The coding style also affects the design of finite-state automata/machine (FSM) that are common is the QueueCore. A poor choice of codes will result in a state machine that uses too much logic, or is too slow, or both. Many advanced techniques have been developed for choosing an optimal state assignment. Typical such approaches use the minimum number of state bits or assume a two-level logic [7].

Fig. 5 compares the encoding tradeoffs for a simple FSM for the Altera STRATIX-EP1S FPGA and Structured ASIC library using three encoding techniques: *gray* code encoding, *binary* encoding, and *onehot* encoding of states.

When ASICs models are the target process, fully encoded representation such as binary or gray code encoding of states lead to space efficient design, wheres the one-hot encoding scheme consumes more resources. This is different for FPGA devices with a large number of flip-flops, where a decoded representation reduces decoding logic at the cost of typically underutilized flip-flops.

Since the resource distribution of the *onehot* encoded FSM maps well to the re-

source distribution of the target architecture, it is a very compact representation.

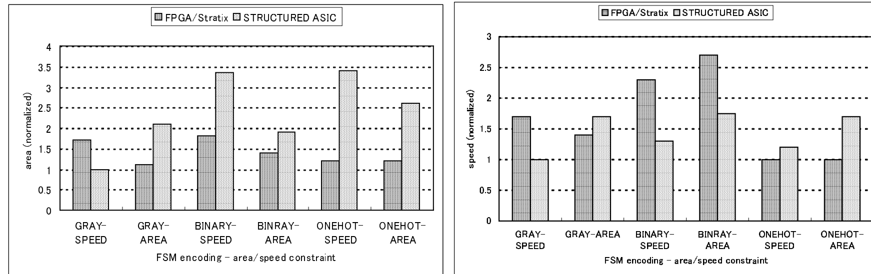

Fig. 5. Resource usage and timing for state machine synthesis using different encoding schemes

6 Modeling Style with Special FPGAs Circuits and Storage Structure

To compare different modeling styles and their implementation in hardware, we have considered the design for an adder module with several different algorithm:
$DEDA$ uses of dedicated logic,
$LGCA$ use only logic element,
$DFTA$ use of default implementation
To compare these design styles, we have compiled these adders with and without target-specific module generators. Resource usage and timing of these implementations of a 32 bit adder are shown in Figures 6. The $DFTA$ implementation was slightly better than $LGCA$ and $DRDA$ version, because the synthesis tool contains a powerfully library of module generators. Implementations not based on the fast arithmetic functionality of the FPGA have almost similar resource usage which is much higher than the technology-specific version, and their performance is slower by a factor of 1.1 to 1.4. Fig. 7 compares two different target

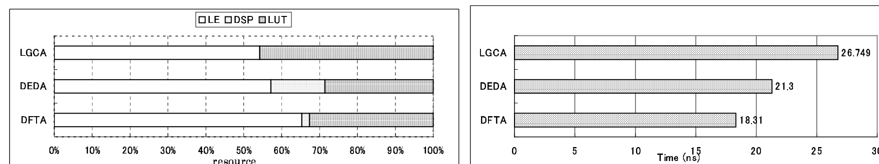

Fig. 6. Resource usage and timing for 32 bit adder.

implantation for 256x33 QREG for various optimizations. Depending on the target implementations device, either logic elements (LEs) or total combinational functions (TC) 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.

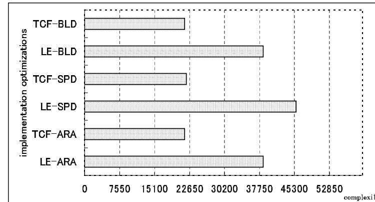

Fig. 7. Resource usage and timing for 256*33 bit QREG unit for different coding and optimization strategies.

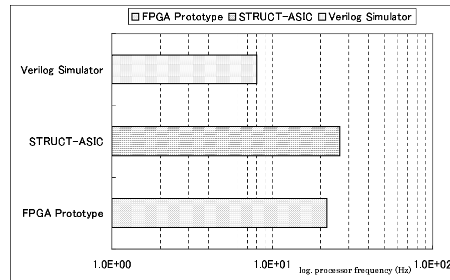

Fig. 8. Achievable frequency is the instruction throughput for hardware implementations of the QueueCore processor. Simulation speeds have been converted to a nominal frequency rating to facilitate comparison.

7 Results and Discussions

Table 1 summarizes the synthesis results of QueueCore for FPGA Stratix and HardCopy devices. The complexity of each processor module as well as the whole processor core are given as the number of logic elements(LEs) for the Stratix FPGA device and as the TCF cell count for the HardCopy device (Stuructured

ASIC). The design was optimised for BLD optimization guided by a properly implemented constraint table. Figure 9 shows the floorplan of the placed and routed QueueCore processor. The pipeline stages have been placed from the top right to the lower left corner of the target Stratix EP1S device. From the above experiment, we found that the processor consumes about 80.4% of the total logical elements of the target device.
Figure 8 shows the achievable throughput which can be obtained from this design on different execution platforms. 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.

Table 1. Queue processor design results: modules complexity as LE (logic elements) and TCF (total combinational functions) when synthesised for FPGA (with Stratix device) and Structured ASIC (HardCopy II) families.

Descriptions	Modules	LE	TCF
instruction fetch unit	IF	633	414
instruction decode unit	ID	2573	1564
queue compute unit	QCU	1949	1304
barrier queue unit	BQU	9450	4348
issue unit	IS	15476	7065
execution unit	EXE	15868	7241
queue-registers unit	QREG	38543	21590
memory access	MEM	5158	3936
control unit	CTR	171	152
Queue processor core	QueueCore	89821	47614

8 Concluding Remarks

A Queue (QueueCore) procerssor for Stratix FPGA was created using modular design structure in high-level prototyping system. The architecture is flexible, with library files controlling key definitions needed by various components. Recompilation can be done to make desired changes to pipeline information, QREG size, memory size and various other architectural features. Each architectural components has been tested through extensive simulation, as well as actual implementation in the Stratix FPGA device.
We also described what constraints contribute to efficient FPGA implementation. This has been demonstrated by target processor core as a single high-density FPGA.

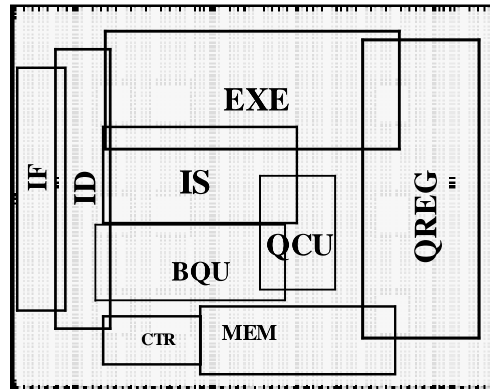

Fig. 9. Floorplan of the placed and routed processor core. The modules of the processor show considerable overlap as logic is mapped according to interconnect requirements.

Evaluation results reveal that queue processor achieves a speed of about 22.5 and 25.5 MHZ for QREG16 (QREG size is 32*16 entries) and QREG264 (QREG size is 32*264 entries) respectively. We also found that the processor consumes about 80.4% of the total logical elements of the Stratix 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 multichip partitioning which results in a loss of resource efficiency. Only, few clearly identified such as the Barrier and the QREG units need to be specially optimised at the HDL source level to achieve efficient resources usage.

References

1. M. Sowa , Ben A. Abderazek, T. Yoshinaga, "Parallel Queue Processor Architecture Based on Produced Order Computation Model", Int. Journal of Supercomputing, HPC, Vol. 32,No. 3, pp. 217-229, June 2005.
2. B. A. Abderazek, M. Arsenji, S. Shigeta, T. Yoshinaga, M. Sowa, Queue Processor for Novel Queue Computing Paradigm Based on Produced Order Scheme, Proc. of HPC, IEEE CS, 0-7695-2138-X/04, pp. 169-177, July 2004
3. M. Sheliga, and E. H. sha,Hardware/Software Co-design With the HMS Framework," Journal of VLSI Signal Processing Systems, Vol. 13, No. 1,pp. 37-56, 1996.
4. D. Lewis et al, The Stratix Logic and Routing Architecture, Proc FPGA-02, pp 12-20, 2002
5. Cadence Design Systems:http://www.cadence.com/
6. Altera Design Software: http://www.altera.com/
7. A.E.A Almaini, et al.: State Assignment of Finite State Machines using a Genetic Algorithm, IEE Proc. on Computers and Digital Techniques, pp. 279-286. 1995.