Advanced Optimization and Design Issues of a 32-bit Embedded Processor Based on Produced Order Queue Computation Model

Hiroki Hoshino, Ben A. Abderazek, Kenichi Kuroda
The University of Aizu,
School of Computer Science and Engineering,
Adaptive System Laboratory,
Aizu-Wakamatsu 965-8580, Fukushima, Japan.

Abstract

Queue computing based programs are generated using a so called level order traversal that exposes all available parallelism in the programs. All instructions within the same level are data independent from each other and are safely to be executed in parallel. This property is leveraged by the compiler generating queue programs with high amounts of grouped independent instructions. Thus, the hardware invests little efforts to find parallelism. In this paper, we present various optimization and design issues of a synthesizable queue processor architecture targeted for embedded applications. A prototype implementation is produced by synthesizing the high-level model for a target FPGA device.

1 Introduction

Instruction level parallelism (ILP) is the key to improve the performance of modern general or special purpose architectures. ILP allows the instructions of a sequential program to be executed in parallel on multiple data paths and functional units. Data and control independent instructions determine the groups of instructions that can be issued together while keeping the program correctness. Two approaches have been followed: superscalar machines using its hardware to schedule the program at run-time, and VLIW machines relying entirely on the compiler to schedule the program. For both technologies, the compiler algorithms are critical to facilitate the schedule. Sophisticated compiler scheduling algorithms are used to expose ILP on sequential code. This techniques concentrate mainly on loops where programs spend most of their running time.

However, these aggressive optimizations have the side effect of increasing the register pressure.

Traditional high performance processors have a fixed number of architectural registers that are not enough for the current compiler technology [1]. Hardware designers and compiler writers have proposed ideas to handle the problem of high ILP while keeping register requirements low to avoid spilling registers to memory. A hardware/compiler technique to alleviate register pressure is to provide more registers than allowed by the instruction encoding. In [2] the usage of queue register files has been proposed to store the live variables in a software pipelined loop schedule while minimizing the pressure on the architectural registers. The work in [3] proposes the use of register windows to give the illusion of a large register file without affecting the instruction set bits.

An alternative to hide the registers from the instruction set encoding is by using a queue machine. We proposed a produced order parallel Queue processor (PQP) [4, 5, 6]. 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).

Queue processor has several promising advantages over register-based machines. First, Queue programs have higher instruction level parallelism because they are constructed using breadth-first algorithm [4]. Second, Queue based instructions are shorter because they do not need to specify operands explicitly. That is, data is 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, Queue based instructions are free from false dependencies. This eliminates the need for register renaming [4].

In this paper, we present the design and evaluation results of a 32-bit embedded Queue processor with several hardware optimization techniques. The presented QueueCore processor implements all hardware features found in our earlier non-optimised version (PQP core). In addition, it supports queue-register control instruction and uses a novel technique (QCaEXT) to extend immediate values and memory instruction offsets that were otherwise not representable because of bit-width constraints in the earlier version. 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.

2 System architecture overview

The QueueCore processor implements a producer order instruction set architecture. Each instruction can encode at most two operands that specify the location in the queue register file from where to read the operands. The processor determines for each instruction the physical location of the operands by adding the offset reference in the instruction to the current position of QH pointer. A special unit called Queue Computation Unit is in charge of finding the physical location of source operands and destination within the queue register file allowing parallel execution of instructions. Every instruction of the QueueCore is 16-bit wide. For cases when there are insufficient bits to express large constants, memory offsets, or offset references, a covop instruction is inserted. This special instruction extends the operand field of the following instruction by concatenating it to its operand.

2.1 Offset references

We classify instructions that read and write into the queue as offsetted instructions. For example, a binary operation whose both operands are not in QH is a 2-offset instruction. A binary operation with one operand taken from QH and the other from another location is a 1-offset instruction. An operation whose operands are taken from QH is a 0-offset instruction. Unary instructions can be 1-offset or 0-offset since they have only one operand. From our earlier study about the offsetted instruction distribution for SPEC benchmark programs, we found that 0-offset and 1-offset instructions account for about 96% of the total instructions. 2-offset instructions appear less than 7%. From this evidence and to accomplish our goal of a compact instruction set and simple hardware we designed a 1-offset constrained instruction set where instructions are allowed to have at most one offset reference. The QueueCore compiler [7] deals with 2-offset instructions by inserting a "duplicate" instruction to copy and move one of the operands to QH. This method converts 2-offset instructions to 1-offset instructions at the cost of one duplicate instruction. More details are published in [7]. This method is very effective to produce a Code-size aware compiler.

2.2 Instruction set architecture

All instructions are 16-bit wide, allowing simple instructions fetch and decode stages and facilitating pipelining of the processor. In the current version of our implementation, we target the QueueCore processor 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, QueueCore processor 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 [7] outputs full addresses and full constants and it is the duty of the QueueCore 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 [7].

2.3 Grouped instruction level parallelism

In addition to generate the instruction sequence for queue programs, the level-order traversal exposes all available parallelism in the programs. All instructions within the same level are data independent from each other and are safely to be executed in parallel. This property is leveraged by the compiler generating queue programs with high amounts of grouped independent instructions. Thus, the hardware invests little efforts to find parallelism and therefore a smaller instruction window than conventional superscalar processors. Figure 1 shows a simple DAG expression and both RISC and QueueCore program. It is clear to notice that the first four instructions in Figure 1(c) are independent. Thus, they can be grouped in one group and executed in parallel. We define this feature as Grouped – instruction – level – parallelism (GILP).
The last four instructions in Figure 1 (c) can be also grouped and executed in parallel. Thus, the QueueCore needs a small instruction window when compared with conventional architecture. Figure 2 shows the compile-time exposed Instruction Level Parallelism. From this experiment, we found that for various benchmark programs the average ILP is around 3. Therefore, we decided that the issue width of the QueueCore should be set to 4 in order to prepare for those dynamic extracted ILP.

2.4 Instruction pipeline structure

The execution pipeline operates in six stages combined with five pipeline-buffers to smooth the flow of instructions through the pipeline. Data dependencies between instructions are automatically handled by hardware interlocks. Below we describe the salient characteristics of the QueueCore core.

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.

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. 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.
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 QCU unit keeps track of the current value of the QH and QT pointers.

Barrier: inserts barrier flags for dependency resolutions.

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. An instruction is ready to be issued if its data and its corresponding functional unit are available. The processor reads the operands from the QREG in the second half of the IS stage and execution begins in the execution stage (stage 6).

Execution (EXE): The macro data flow execution core consists of 4 integer ALU units, 2 floating-point units, 1 branch unit, 1 multiply unit, 4 set-units, and 2 load/store units. The load and store units share a 16-entry address window (AW), while the integer units and the branch unit share a 16-entry integer window (IW). The floating-point accelerator (FPA) has its own 16-entries floating point window (FW). The load/store units have their own address generation logic. Stores are executed to memory in-order.

To execute instructions in parallel, the QueueCore 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. To calculate the source1 address of a given instruction, the number of consumed data (CNBR) field is added to the current queue head value (QHn). The destination address on the next instruction (INSTn+1) is calculated by adding the PNB field to the current queue tail value (QTn). 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.

2.5 Displacement extension mechanism

For cases when there are insufficient bits to express displacement (large constants, memory offsets, or offset references) a covop instruction is inserted. The QueueCore use this special instruction and dynamically extends the operand field of the following instruction by concatenating it to its operand. The hardware mechanism consists of a convey-

Figure 3. Displacement extension hardware mechanism.

register and a concatenation unit. Figure 3 shows an example of displacement extension for a ld instruction. The mechanism inserts the displacement value embedded within the covop instruction in the covop-register (110000001 in this example). The value in the covop register is then concatenated with the displacement value of the load instruction (0100). Finally, the concatenation unit generates the extended offset value (110000010100) for the above instruction.

2.6 FPA organization

The QueueCore floating-point accelerator is a pipelined structure and implements a subset of the IEEE-754 single precision floating-point standard. 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 QueueCore 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 towards positive infinity, round towards negative infinity, and round towards zero. The area efficient FALU hardware is shown in Figure 4.

3 Design 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 and QueueCore compiler [7]. The embedded applications are selected from MediaBench [12] and MiBench [13] suites. The selected benchmarks include two video compressing 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 [14] 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 QueueCore compiler. The table shows that the binaries for the QueueCore processor are about 70% smaller than the binaries for MIPS and about 50% smaller than ARM [15]. Compared to dual-instruction set embedded RISC processors, MIPS16 [16] and Thumb [17], QueueCore binaries are about 20% and 40% denser, respectively. When compared to the CISC architecture, Pentium processor [18], QueueCore binaries are about 14% denser. Execution time for a serial queue machine can be estimated at compile-time by counting the number of cycles required to execute all instructions in the program. Although this measurement does not reflect any run-time properties other than the number of instructions in the binary, it gives an approximation on how the queue computation model can exploit the parallelism found in programs. As the compiler schedules the program in level-order manner, it exposes the critical path of the basic blocks and groups all independent instructions in execution levels. These execution levels can be executed concurrently in a parallel queue architecture. Therefore, the execution time for a parallel queue machine is given by the number of levels in the program. Table 2 shows the execution time in cycles for serial (PQP-S) and parallel (QueueCore) 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.

### Table 1. Normalized code sizes for various benchmark programs over different target architectures.

<table>
<thead>
<tr>
<th>Bench.</th>
<th>MIPS16</th>
<th>ARM</th>
<th>Thumb</th>
<th>x86</th>
<th>QC</th>
</tr>
</thead>
<tbody>
<tr>
<td>H.263</td>
<td>58.00</td>
<td>83.66</td>
<td>80.35</td>
<td>57.20</td>
<td>41.34</td>
</tr>
<tr>
<td>MPEG2</td>
<td>53.09</td>
<td>78.40</td>
<td>69.99</td>
<td>53.22</td>
<td>36.75</td>
</tr>
<tr>
<td>Susan</td>
<td>47.34</td>
<td>80.48</td>
<td>77.54</td>
<td>46.66</td>
<td>35.12</td>
</tr>
<tr>
<td>AES</td>
<td>51.27</td>
<td>86.67</td>
<td>69.59</td>
<td>44.72</td>
<td>34.93</td>
</tr>
<tr>
<td>Blowfish</td>
<td>54.59</td>
<td>86.38</td>
<td>82.76</td>
<td>57.45</td>
<td>45.49</td>
</tr>
<tr>
<td>FFT</td>
<td>58.09</td>
<td>100.74</td>
<td>92.54</td>
<td>46.27</td>
<td>36.77</td>
</tr>
<tr>
<td>Average</td>
<td>53.73</td>
<td>86.05</td>
<td>78.79</td>
<td>50.9</td>
<td>36.77</td>
</tr>
</tbody>
</table>

### Table 2. Execution time and speedup results.

<table>
<thead>
<tr>
<th>Bench.</th>
<th>PQP-S</th>
<th>QueueCore</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>H.263</td>
<td>25980</td>
<td>11777</td>
<td>2.21</td>
</tr>
<tr>
<td>MPEG2</td>
<td>22690</td>
<td>10412</td>
<td>2.18</td>
</tr>
<tr>
<td>Susan</td>
<td>11321</td>
<td>7613</td>
<td>1.49</td>
</tr>
<tr>
<td>AES</td>
<td>5132</td>
<td>1438</td>
<td>3.57</td>
</tr>
<tr>
<td>Blowfish</td>
<td>5377</td>
<td>3044</td>
<td>1.77</td>
</tr>
<tr>
<td>FFT</td>
<td>9127</td>
<td>5234</td>
<td>1.74</td>
</tr>
</tbody>
</table>
Table 3. Speed and power consumption comparisons for various Synthesizable CPU cores over speed (SPD) and area (ARA) optimizations. This evaluation was performed under the following constraints: (1) Family: Stratix; (2) Device: EP1S25F1020; (3) Speed: 25.5 MHz. The speed is given in MHz.

<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>QueueCore</td>
<td>25.5</td>
<td>24.2</td>
<td>90</td>
</tr>
</tbody>
</table>

configuration is 44.4 ns, that was 22.5 MHz of clock frequency. For QueueCore 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.

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 QueueCore processor, all instructions designed are fixed format 16-bit words with minimal decoding effort. As a result, the QueueCore 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 to 70% smaller than programs for conventional architectures. The importance of the system memory size translates to an emphasis on code size since data is dictated by application. Larger memories mean more power, and optimization power is often critical in embedded applications. In addition, instructions of QueueCore processor specify operands implicitly. This design decision makes instructions independent from 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 [9, 8].

Performance of QueueCore in terms of speed and power consumption is compared with various synthesizable CPU cores as illustrated in Table 3. The SH-2 is a popular Hitachi SuperH based instruction set architecture [10, 11]. The SH-2 has RISC-type instruction sets and 16x32 bit general purpose registers. All instructions have 16-bits fixed length.

The SH-2 is based on 5 stages pipelined architecture, so basic instructions are executed in one clock cycle pitch. Similar to our QueueCore core, the SH-2 also has an internal 32-bit architecture for enhanced data processing ability. LEON2 is a SPARCv8 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 [19].

From the result shown in Table 3, the QueueCore processor shows better speed performance for both area and speed optimizations when compared with SH-2, PQP and ARM7 (hard core) processors. The QueueCore 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). QueueCore core also shows 25% less power consumption when compared with PQP and consumes less power than LEON2 and MicroBlaze processors. However, QueueCore core consumes more power than ARM7 processor, which also has less area than PQP and QueueCore for both speed and optimization (not shown in the table). This difference comes from the small hardware configuration parameters of ARM7 when compared to our QueueCore core parameters.

4 Conclusions and future work

The QueueCore processor implements a producer order queue computation model instruction set where operands can be read from a distinct location of the head of the queue specified as an offset reference in the instruction. This architectural modification enables the processor and the compiler to find and execute common subexpressions. We presented in this paper advanced optimization issues for a 32-bit produced order queue processor targeted for embedded applications. Hardware design evaluation results reveal that the QueueCore processor achieves a speed of about 25.5 MHz. In addition, the QueueCore processor shows better speed performance for both area and speed optimizations when compared with SH-2, PQP and ARM7 (hard core) processors. On average the QueueCore has about 40.87% higher speed than SH-2 processor. QueueCore also shows 25% less power consumption when compared with our earlier non-optimized PQP processor, and consumes less power than SH-2, LEON2 and MicroBlaze cores.

The QueueCore feature low power, low complexity and offers an attractive option in the design of ubiquitous systems requiring low power consumption and small memory footprints. Future work will investigate research about the QueueCore usage within ubiquitous wireless monitor-
ing system for elderly monitoring, where power and performance are real design challenges.

References


[9] P6 Power Data Slides, Provided by Intel Corporation to Universities.


