Single Instruction Dual-Execution Model Processor Architecture

Taichi Maekawa s1130205

Abstract

I present the architecture and preliminary evaluation results of a novel dual-mode processor architecture which supports queue and stack computation models in a single core. The core is highly adaptable in both functionality and configuration. It is based on a reduced bit produced order queue computation instruction set architecture and functions into Queue or Stack execution models. This is achieved via a so called dynamic switching mechanism implemented in hardware.

The current design focuses on the ability to execute Queue programs and also to support Stack based programs without considerable increase in hardware to the base architecture. The architecture description and design results are presented in a fair amount of detail.

1 Introduction

Embedded processors are used in numerous systems, including laptops, cellular phones, etc. Although these systems differ in terms of their communication and computation requirements, they share the common need for low power and small memory footprint. To support this trend, I present a novel embedded dual-mode architecture, which supports Queue and Stack programming models. The idea is fueled by earlier proposed work about a low power QueueCore architecture [1, 2].

The base QueueCore/PQP processor is based on produced order queue computation model and uses a circular queue-register (QREG) for intermediate data. For example, an add instruction implicitly reads its first operand from the head of the QREG, its second operand from a location explicitly addressed with an offset from the first operand location. The computed result is stored into the QREG at a position pointed at by a special pointer called Queue-tail pointer. A Queue based execution model is analogous to the Java - Stack execution model (Stack-EM), which is widely used in Internet applications. It has operations in its instruction set which implicitly reference an operand queue-register (OPQ), just as a Stack-EM has an operation which implicitly references an operand stack-register (OPS).

In Stack-EM, implicitly referenced operands are retrieved from the top of OPS and results are returned back to the OPS. To this end, operands and results manipulation schemes in both models make the internal instruction processing and hardware modules quite similar in functionality and complexity. Figure 1 illustrates operands manipulation schemes in both Queue (a) and Stack (b) models.

In this research work, I present architecture, design and preliminary evaluation of an embedded dual-mode processor architecture (DPA) [18], which supports Queue and Stack based programs in a single and simple core. This idea is motivated by the aforementioned Queue computing features and also by the urgent need for high hardware usability, compatibility, and fast system design. The proposed DPA architecture uses a single shared instruction set and dynamically switches between Queue-EM and Stack-EM models using a so called Dynamic Switching Mechanism (DSM) [3]. The entities share a common data path and may operate independently but not in parallel. I will show that only a little hardware addition is required to support a Stack-based execution model on the base QueueCore architecture, without considerable performance degradation.

2 DPA execution overview

As mentioned, the DPA processor has two execution modes: (1) Queue-EM and Stack-EM. In Queue-EM, the shared storage unit (SSU) is organized as a circular Queue-register [4, 5]. While in Stack-EM, the SSU is organized as a stack-register unit. A so called Dynamic Switching mechanism switches dynamically between these two modes according to the program being loaded.
The Queue-EM is based on produced order queue computation (QCM), which is the formal definition of a computer that uses a FIFO data structure to store temporary computations. A queue processor is similar to a stack processor in the sense that operations implicitly access the operand queue as stack-based instructions access the operand stack. In the QCM, elements are inserted, or enqueued, through a write pointer named QT that references the rear of the queue. Elements are removed, or dequeued, through a read pointer named QH that references the head of the queue. All operations accessing the operand queue fall into one of three categories: read-write, write-only, and read-only. Read-write operations take some data from the queue, perform some computation, and write the result back to the queue. For example, add, implicitly takes two data from the queue, performs its addition, and writes the result back to the queue. Write-only operations just generate data in the operand queue, like ld operation, that loads a memory location into the queue. Read-only operations, that only take data from the operand queue like st operation, that takes a datum from the operand queue and writes it into a memory location. Depending on the instruction characteristics, the hardware of a queue computer manipulates the read and write pointers QH, and QT. Every time a datum is dequeued the QH pointer is updated to point to the next location QH + 1, and every time a datum is enqueued the QT pointer is updated to point to the following location QT + 1.

2.1 Power effective embedded DPA core

Power dissipation limits have emerged as a first-order design constraint for microprocessors. For high-end applications, increasing micro architecture complexities, clock frequencies, and die areas are pushing the chip-level power consumption literally to the edge. DPA core design approach takes into account performance, power consumption, and dissipation considerations early in the design cycle. It maintains a power-centric focus across all levels of design abstraction. In DPA core, all Queue based instructions are fixed format 16-bit words with minimal decoding effort. As a result, Queue programs have much smaller programs than either RISC or CISC machines. Program sizes for my architecture are found in earlier research work, 60% smaller than programs for RISC (SPARC) codes [10]. 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 specify operands implicitly. This design decision makes instructions independent from the actual number of physical Queue words (Queue-register). Thus, instructions are 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 [6, 7].

In the current version of implementation, I target the processor for small applications where concerns the focus is on the ability to execute codes (Queue and Stack programs) on a processor core with small die size and low power consumption characteristics.

The execution pipeline of the DPA processor operates in five pipeline stages:

Instruction Fetch (FU): The fetch unit fetches instructions from the program memory and inserts them into a fetch buffer. In Queue-EM mode, the FU fetches four instructions per cycle. However, for Stack-EM mode it fetches one instruction per cycle. A special fetch mechanism is used to update the fetch counter according to the mode being processed.

Instruction Decode (DU): Decodes instructions. ANB opcodes and operands. The decode unit (DU) has 4 decode circuits (DC) and 1 Mode-Selector-Register (MS). The latter, is set to “0” or “1” according to the type of execution modes.

Queue-Stack Computing Unit (QSCU): The processor’s computation unit reads information from the DU and uses them to compute each instruction’s source and destination locations in both Queue and Stack execution models.

Instructions Issue (IU): The issue stage issues ready instructions to the execution unit. Memory and register dependencies are checked in this unit/stage. This unit also checks the availability of the source and destination addresses.

Execution Unit (EXE): EXE executes issued instructions and sends the results to the Shared Storage Unit (SSU) or data memory. The EXE consists of four Arithmetic Logical Units (ALU), four Set register units, two Load/Store units and one Branch unit.

Write-Back Unit (SSU): The write back unit writes the result back to the PROG/DATA memory or the SSU.

2.2 Dynamic mode switching mechanism

DPA uses a simple hardware mechanism (DSM) to dynamically switch between execution modes. The block diagram of the DSM is shown in Fig. 2. It consists of a switching circuitry (SW) and a dynamic computation unit [3]. The QSCU calculates the source and destination addresses for instructions in both Queue-EM and Stack-EM modes. The DSM detects the instruction mode by decoding the operand of a special operation,
named “switch” instruction. After it detects the mode, it inserts a mode-bit for all instructions between the current and the next “switch” instruction.

Figure 2: Dynamic Mode-switching mechanism.

2.3 Source and destination addresses Calculation

In Queue-EM execution mode, each instruction needs to know its QH and QT values. The above values are easy to obtain in a serial Queue execution model, since the QH is always used to fetch instructions from the operand Queue and the QT is always used to store the result of the computation into the tail of the operand Queue. However, in parallel execution, the values for QH and QT are not explicitly determined.

Fig. 3 (a) shows the hardware mechanism used for calculating source1 (first operand), source2 (second operand), and destination (result) addresses for current instruction. The computing unit keeps the current value of the QH and QT pointers. Four instructions arrive at this unit in each cycle. The first instruction uses the current QH (QH(n)) and QT (QT(n)) values for source1 and destination addresses respectively. As shown in Fig. 3 (a), the source2 of a given instruction is the first calculated by adding the source1 address to the displacement (OFFSET) (OFFSET(n)).

Fig. 4(a) shows the the pointers – update mechanism of current QH and QT values for next instruction. The number of consumed data (CN) field (8-bit) is added to the current QH value (QH) to find the next QH (NQH) and the number of produced data (PN) field (8-bit) is added to the current QT value (QT) to find the next QT (NQT). The other three instructions source and destination addresses are calculated similarly.

In Stack-EM mode, the execution is based on Stack-EM model. The hardware used for calculating source1, source2, and destination addresses is shown in Fig. 3(b). It is the same hardware used for calculation of operands in Queue-EM mode. The computing unit keeps the current value of the Stack pointer (TOP). One instruction arrives to the QSCU unit in each cycle. The source1 address is popped from the operand stack pointed by the current TOP pointer value (TOP(n-1)). The source2 is calculated by subtracting “1” from current TOP pointer value. The number of consumed data (CN) is subtracted from the current TOP value (TOP(n-1)) and the number of produced data (PN) is added for finding the result address (DEST(n)). Fig. 4(b) shows the hardware mechanism used for calculating the result address for current instruction. The destination address of current instruction points to the next TOP pointer value (NTP), which will be used for the next instruction’s source1 address.

Figures 5 (a) and (b) show two examples of source and destination address calculations for both Queue-EM and Stack-EM execution models respectively. For simplicity, only two instructions “add -1” and “mul -1” are shown in the fetch buffer (FB). The “MS” is the instruction-mode-selector register and is set to “0” which means that the DPA core is in Queue-EM mode. The DU decodes instructions and calculates several fields for each instruction. As shown in the Fig. 5 (a), the fields are: IM (instruction mode), OP (opcode), OPR (operand), CN (consumed number), and PN (produced number). The values of IM, OP, OPR, CN, and PN are: 0, add -1, 1, and 1 for the first instruction “add -1”. The same calculation scheme if performed for the following instructions. Using the mechanism shown in Fig. 3 (a) and Fig. 4 (a), the source1 (SRC1), source2 (SRC2), and destination (DEST) addresses for each instruction are calculated in the QSCU stage. Fig. 5 (b) shows another example illustrating the calculation of SRC1, SRC2, and DEST fields in Stack-EM mode.

3 System architecture design

To make the DPA design easy to debug, modify, and adapt, I decided to use a high-level description, which was also used by other system designers, such as in [13, 14]. The DPA core has been developed in Verilog HDL. After synthesizing the HDL code, the designed processor gives the ability to investigate the actual hardware performance and functional correctness. It also gives the possibility to study the effect of coding style and instruction set architectures over various optimizations. For the DPA processor to be useful for these purposes, I identified the following requirements: (1) High-level description: the format of the DPA description should be easy to understand and modify, (2) Modu-
lar: 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 DPA has been designed with a distributed controller to facilitate debugging and future adaptation for specific application requirements since I 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, I have decided to break up the unstructured control unit to small and 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 DPA 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.

4 Results and discussion

Table 1 shows the hardware configuration parameters of the designed DPA core, each modules and PQP core for the Stratix FPGAs [8, 9]. The complexities of each and whole module are given as the number of logic elements (LEs). When compared to the PQP core, DPA requires only 7.4% extra hardware for speed optimization (SOP) and 9.6% extra hardware for area optimization (AOP). The achievable frequency of the DPA core is 72 for speed and area optimizations respectively.

Table 1: Synthesis results. LEs means logic elements. SOP means speed optimization and AOP means area optimization.

<table>
<thead>
<tr>
<th>Descriptions</th>
<th>LE-SOP</th>
<th>LE-AOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction fetch unit</td>
<td>142</td>
<td>140</td>
</tr>
<tr>
<td>Instruction decode unit</td>
<td>241</td>
<td>226</td>
</tr>
<tr>
<td>DPA computation unit</td>
<td>132</td>
<td>116</td>
</tr>
<tr>
<td>Issue unit</td>
<td>757</td>
<td>680</td>
</tr>
<tr>
<td>Execution unit</td>
<td>13477</td>
<td>9605</td>
</tr>
<tr>
<td>Shared storage unit</td>
<td>9339</td>
<td>7904</td>
</tr>
<tr>
<td>DPA core</td>
<td>28933</td>
<td>22431</td>
</tr>
<tr>
<td>PQP core (base)</td>
<td>26948</td>
<td>20690</td>
</tr>
</tbody>
</table>

Performance of DPA core in terms of speed and power consumption is compared with various synthesized CPU cores as illustrated in Table 2 and Table 3. The source code for PQP and the OpenRisc1200 cores were available. Therefore, I synthesized and evaluated their performance using Altera Quartus II professional edition. The other cores data were obtained from corresponding manufacturers performance reports and published work. LEON2 is a SPARC V8 compliant 32-bit RISC processor [17]. ARM7 is a simple 32-bit RISC core [15, 16].

Figure 3: Address calculation mechanisms for sources and destination: (a) Queue-EM computing circuit and (b) Stack-EM computing circuit.

Figure 4: Address calculation mechanisms for next instruction’s source1 and destination: (a) Queue-EM mode and (b) Stack-EM mode.
The SH2-DSP (SH7616) and SH3-DSP (SH7710, and SH7720) series processor cores are based on a popular Hitachi SuperH (SH) instruction set architecture [11, 12]. The SH has RISC-type instruction sets and 16x32 bit general purpose registers. All instructions have 16-bit fixed length. The SH2-DSP and SH3-DSP are based on 5 stages pipelined architecture, so basic instructions are executed in one clock cycle pitch. Similar to our processor, the SH also has an internal 32-bit architecture for enhanced data processing ability.

Table 2: DPA speed comparisons with various synthesizable CPU cores over speed (SPD) and area (ARA) optimizations. This evaluation was performed under the following constraints: (1) family: Cyclone II and (2) device: EP2C35F672C6. The speed is given in MHz.

<table>
<thead>
<tr>
<th>Cores</th>
<th>Speed (SPD)</th>
<th>Area (ARA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPA</td>
<td>72.43</td>
<td>72.04</td>
</tr>
<tr>
<td>PQP</td>
<td>73.40</td>
<td>73.15</td>
</tr>
<tr>
<td>OpenRisc1200</td>
<td>32.64</td>
<td>32.1</td>
</tr>
<tr>
<td>ARM7</td>
<td>25.2</td>
<td>24.5</td>
</tr>
<tr>
<td>LEON2</td>
<td>27.5</td>
<td>26.7</td>
</tr>
<tr>
<td>MicroBlaze</td>
<td>26.7</td>
<td>26.7</td>
</tr>
</tbody>
</table>

From the results shown in Table 3, the DPA core has higher speed than OpenRisc1200 core, SH7616, ARM7, LEON2 and MicroBlaze cores respectively. When compared with PQP processor, DPA has 1.34% and 1.54% speed decrease for speed and area optimizations respectively. For power consumption comparison, the DPA core consumes less power than LEON2 and SH7616 cores, SH series and OpenRisc1200 cores respectively. However, DPA core consumes more power than the ARM7 core, which also has less area than PQP and DPA for both speed and optimization (not shown in the table). This difference comes mainly from the small hardware configuration parameters of ARM7 when compared to our DPA core parameters.

Table 3: DPA power consumption comparisons with various synthesizable CPU cores. The power consumption is given in mW.

<table>
<thead>
<tr>
<th>Cores</th>
<th>Average Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPA</td>
<td>122.44</td>
</tr>
<tr>
<td>PQP</td>
<td>122.33</td>
</tr>
<tr>
<td>OpenRisc1200</td>
<td>1005</td>
</tr>
<tr>
<td>SH7616</td>
<td>250</td>
</tr>
<tr>
<td>SH7710</td>
<td>500</td>
</tr>
<tr>
<td>SH7720</td>
<td>600</td>
</tr>
<tr>
<td>ARM7</td>
<td>22</td>
</tr>
<tr>
<td>LEON2</td>
<td>458</td>
</tr>
</tbody>
</table>

5 Concluding remarks

This work presents architecture, design and preliminary evaluation results of a novel queue-stack processor architecture targeted for special purpose applications requiring small memory footprints and low power consumption. From our preliminary evaluation results, I found that the DPA processor achieves a speed of about 72 MHz for speed and area optimization. The core was designed without considerable additional hardware when compared with the base PQP core (only about 7.4% more LEs are required). The DPA processor also shows better speed performance for both area and speed optimizations when compared with other well known Synthesizable CPU cores, except for PQP processor, where DPA has 1.34% and 1.54% speed decrease for speed and area optimizations respectively.

DPA has some limitations. First the DPA doesn’t support Java bytecode. Second the DPA was not tested with big benchmarks. Last the DPA doesn’t support interrupt. For these limitation, our future work will focus on optimizations for Java bytecode and interrupt. And the DPA will be tested using big benchmark programs.
References


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


