Journal of Mobile Multimedia, Vol. 3, No. 4 (2007) 347–370

DUAL-EXECUTION MODE PROCESSOR ARCHITECTURE FOR EMBEDDED APPLICATIONS

MD. MUSFIQUZZAMAN AKANDA, **BEN A. ABDERAZEK, and MASAHIRO SOWA
Network Computing Laboratory, Graduate School of Information Systems,
National University of Electro-communications, Tokyo, Japan
*** Adaptive Systems Laboratory, The University of Aizu, Aizu-Wakamatsu, Japan*
akanda@sowa.is.uec.ac.jp

Received June 18, 2007
Revised December 15, 2007

This paper presents a novel embedded 32-bit processor architecture targeted for mobile and embedded applications. The processor supports Queue and Stack based programming models in a single simple core. The design focuses on the ability to efficiently execute Queue programs and also to support Stack programs without a considerable increase in hardware to the base Queue architecture.
A prototype implementation of the processor 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. From the design and evaluation results, the QSP32 core efficiently executes both Queue and Stack based programs and achieves on average about 65MHz speed. In addition, when compared to the base single-mode architecture (PQP), the QSP32 core requires only about 2.54% additional hardware.

Keywords: Dual-Execution Mode; Queue Computation; Dynamic Switching Mechanism; Embedded Core.
Communicated by: I. Ibrahim & L. Yang

1 Introduction

Mobile processors are used in numerous embedded systems, including laptops, personal digital organizers, wearable computers, cellular phones, mobile Internet terminals, digital cameras, digital cam-coders, smart cards, and sensor networks nodes. Although these systems differ in terms of their communication and computation requirements, they share the common need for low power, security and small memory footprint. There are many efforts in architecture design that address these problems. Thus, computer architects are continuously challenged to bring innovations to design microarchitectures, instruction sets, and compilers in order to keep the balance between performance, complexity, and power. Several processors have achieved success as two or four-way Superscalar implementations. However, adding 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.
The desire for a simple, compatible, and fast machine motivated us to look for alternatives. Our research was inspired by several original ideas [28, 29, 30], which proposed using the Queue (first-in-first-out memory) instead of registers (random access register) as intermediate storage for operands. In [31] it is 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.

In [2, 3], we proposed a novel high performance Parallel Queue processor architecture based on produced order Queue instruction set architecture. The key ideas of the Queue computing model are the operands and results manipulation schemes. The Queue execution model stores intermediate results into a circular Queue-register (QREG). A given 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 finally written into the QREG at a position pointed to a Queue-tail pointer. As a result, Queue based instruction set processor architectures have several promising advantages over register-based machines. First, programs written for Queue based computation have higher ILP because they are constructed using the breadth-first algorithm from a directed acyclic graph [10, 28]. Second, instructions are shorter because they do not need to specify operands explicitly [2]. 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 length 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 [10].

Queue execution model (Queue-EM) is analogous to the conventional Stack execution model (Stack-EM), which is suitable for some special embedded applications. It is well known that, by implicitly referring to operands, Stack code is often shorter than general-purpose register code [16]. Moreover, since all working parameters are always present on the Stack, procedure call overhead is minimal, requiring no memory cycles for parameter passing. Another advantage of Stack execution model is that on interrupts there is no need to save registers. Thus interrupt response latencies are low, making Stack architectures ideal for real-time and embedded systems [16]. The Queue-EM has operations in its instruction set which implicitly reference an operand Queue (OPQ), just as a Stack-EM has operation which implicitly reference an operand Stack (OPS). In Stack-EM, implicitly referenced operands are retrieved from the top of OPS and results are returned back to the OPS (refer to Fig. 1). Therefore, operands and results manipulation schemes in both models (Queue-EM and Stack-EM) make the internal instruction processing and hardware modules quite similar in functionality and complexity.

To boost processor hardware usability, compatibility, and reduction of design time, we propose a novel 32-bit dual-mode architecture (QSP32), which efficiently supports Queue and Stack based programs. In addition, the QSP32 core implements *QCaEXT* scheme - a novel technique used to extend immediate values and memory instruction displacements that were otherwise not representable because of bit-width constraints in the original PQP architecture. We will show later that for supporting Stack based programs, only little hardware is added to the base Queue-EM architecture (PQP) without performance degradation.

An important aspect of developing any new architecture is verification which usually requires complicated and lengthy 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 [1, 17, 18]. These software-based simulation approaches are not capable of predicting all micro-architectural issues related to final physical design. Therefore, we

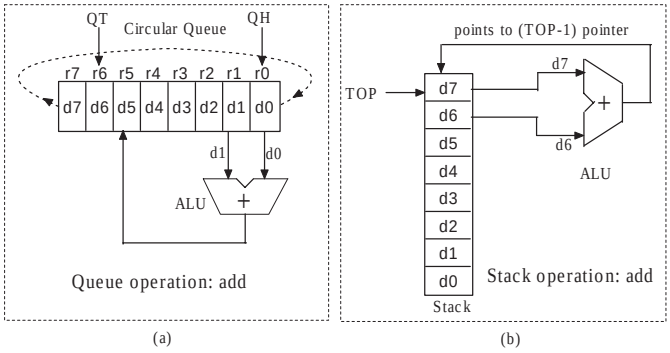


Fig. 1. Operands and results manipulation schemes. (a) Queue (FIFO) computing, (b) Stack (LIFO) Computing.

consider prototyping-based emulation that substitutes real time hardware emulation for slow simulator-based execution. What is important for us is to investigate how to describe the novel QSP32 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 QSP32 core. A prototype implementation is produced by synthesizing the high-level model for the Stratix FPGA device [7, 9]. As a result, we were able to evaluate relative circuit area, speed, and power metrics.

The rest of this paper is structured as follows: Section 2 gives the related work. Section 3 gives the Queue computation model overview. Section 4 gives the embedded application development requirements. In section 5 we give the details of QSP32 system architecture. Section 6 gives the design approach and pipeline control scheme. Section 7 presents the QSP32 evaluation results and discussion. Finally, the last section gives our concluding remarks.

2 Previous Work

During the last years several works have been done in developing different kinds of hybrid processor cores. The ARM926EJ-S [19] is a synthesizable 32bit core which supports hybrid instruction sets. It is an integration of three instruction set architectures: namely the 32bit ARM instruction set, 16bit ThumbR instruction set, and Java bytecode. One part of the Java bytecode is realized in hardware and the remaining complex ones are executed by special software emulation [20]. In [23], the authors implemented a processor core where two virtual processors share one data path. It has two programming models: a Java model and a RISC model. The idea comes from the common instruction (approximately 50%) when compared with Java's core binary instructions and a standard RISC instruction set.

The earliest work about Queue computation idea was proposed in [28]. At the execution stage, each instruction removes the required number of operands from the front of the Queue, performs computation and stores the result back into the operand Queue at a specified offsets from the front of the Queue. A major problem with the above indexed queue architecture is that it requires the relocation of a potentially large number of operands. In addition, the operand Queue within the above architecture is implemented in the main memory.

To overcome the drawbacks of the indexed Queue machine, we proposed in [2, 3, 10] a produced order parallel Queue processor architecture. In this scheme, data produced by instructions are inserted in circular QREG in their executions order and can be reused by other instructions. As a result, performance was highly improved and hardware complexity was decreased. In [3], a new processor architecture based on circular queue register was proposed and designed. There are also a number of microprocessors which implement a Stack directly in hardware [16, 22]. In addition, most of the Stack machines reported previously in the literature have some sort of Stack cache on the processor which serves as the architectural analogue of the register file in conventional register machines. Although the Stack based model has some drawbacks, there are various applications which are based on Stack architecture, such as Java procesors [21, 24].
This research proposes architecture and design evaluation of a novel dual-mode processor (QSP32), which supports both Queue and Stack based programming models in a single and simple core. In addition to the dual-mode execution support, the QSP32 core also implements *QCaEXT* scheme - a novel technique used to extend immediate values and memory instruction displacements that were otherwise not representable because of bit-width constraints in our original Queue (PQP) architecture. We are not aware of any previous research work dealing with both Produced order Queue computation and dual-mode execution support on a single embedded core. To our knowledge, we are the first research group to do so.

3 Queue Computing Overview

The produced order Queue computing model uses a circular QREG (also named operand Queue) instead of random access registers to store intermediate results. A datum is inserted in the QREG in produced order scheme and can be reused. This feature has profound implications in areas of parallel execution, program compactness, hardware simplicity, and high execution speed [2, 3]. This section gives a brief overview of the produced order Queue computation model. We show in Fig. 2 (a) a sample data flow graph for the expressions: $e = ab/c$ and $f = ab(c + d)$. A datum is loaded with load instruction (ld), computed with multiply (*), add (+), and divide (/) instructions. The result is stored in the data memory with store instruction (st). In Fig. 2(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 for m6 node). The instruction sequence for the Queue execution model is correctly generated when we traverse the data flow graph (shown in Fig. 2(a)) from left to right and from the highest to the lowest level. In Fig. 2(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. 2 (c) and the contents of the QREG at each execution stage is shown in Fig. 2 (d). A special register, called queue head pointer (QH), points to the first datum in the QREG. Another pointer, named queue tail pointer (QT), points to the location of the QREG in which the result datum is stored. Immediately after using datum, the QH is incremented so that it points to a datum 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 point to datum a and the QT points to an empty location as shown in Fig. 2 (d) (State 1). The fifth and sixth instructions are also executed in parallel. The *mul* refers a and b then inserts the result $(a * b)$ into the QREG. QH is incremented by

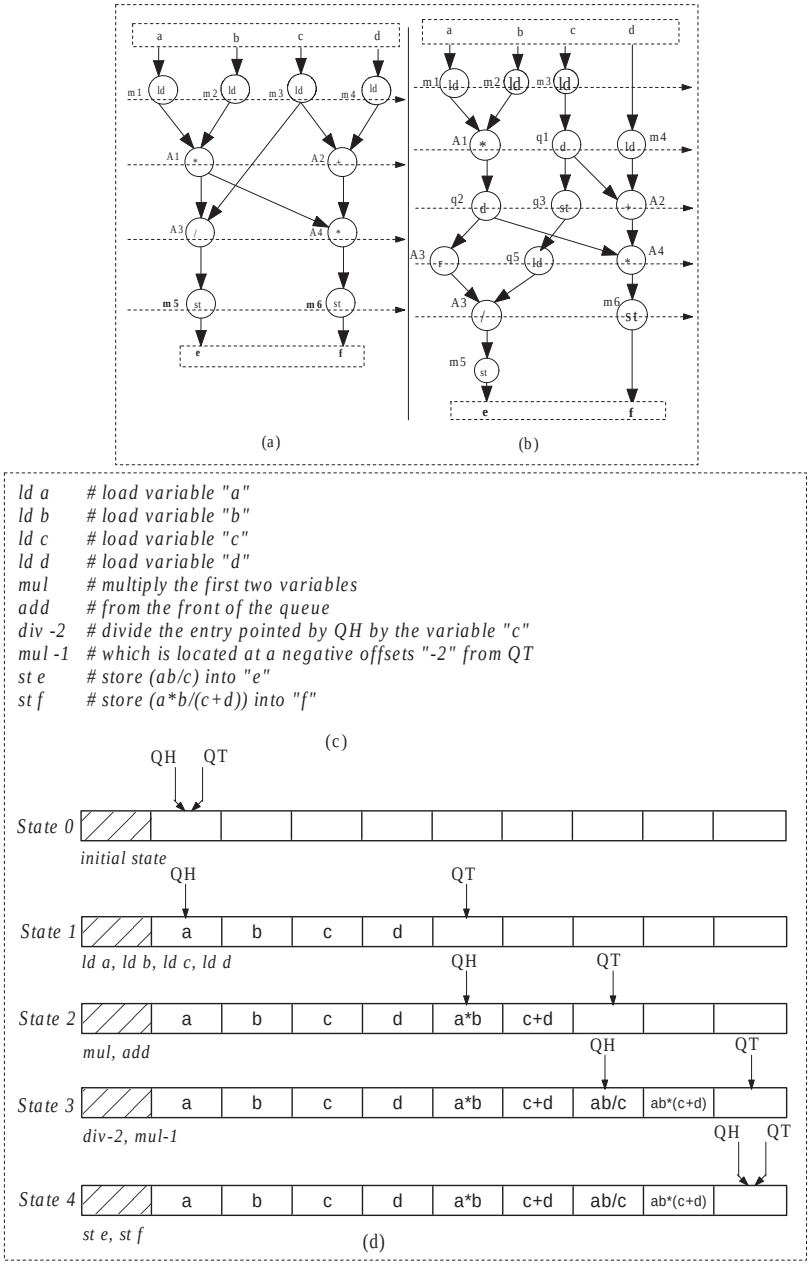


Fig. 2. Sample data flow graph and circular Queue-register contents for the expressions: $e = ab/c$ and $f = ab(c+d)$. (a) Original sample program, (b) Translated (augmented) sample program, (c) Generated instructions sequence, (d) Circular Queue-register content at each execution state.

two and the QT is incremented by one. The *add* refers c and d then inserts $(c + d)$ into the QREG. At this state, the QH and QT are incremented as shown in Fig. 2 (d) (State 2). The seventh instruction (*div*-2) divides the datum pointed to by QH (in this case $a * b$) by the datum located at "-2", negative offset, from QH (in this case c). The QH is incremented 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 $a * b$). After parallel execution of these two instructions, the QREG content becomes as shown in Fig. 2 (d) (State 3). The last two instructions store the result back in the data memory. Since the QREG becomes empty, QH, and QT point to the same empty location (State 4).

4 QSP32 Architecture for Mobile and Embedded Applications

The QSP32 is a dual-mode processor core that offers simple and efficient execution engine for embedded applications. A key design requirement for the QSP32 core was a modular structure to facilitate experiments with a number of extensions for different application areas. Source code modification should be restricted to those modules whose functionality is enhanced. Typically, instruction set extensions involves modification of the instruction decode and execution stage modules. The modular structure provides a flexible framework with well-designed interfaces when changes to the processor architecture are to be made. The QSP32 processor provides a mechanism to add or modify functionality without major redesign efforts. This is achieved with a modular design style with well-designed interfaces between modules and the use of a flexible and extensible pipeline control design. We describe the processor at a high abstraction level. As a result, precise and detailed control on every element is easily achieved without losing the visibility of interrelations with the rest of the elements in the core. Logic synthesis, then, maps high-level operators to module generators, which can select different implementation styles, based on timing, and area optimizations constraints.

4.1 Power Effective Mobile QSP32 Core

Mobile processors share a number of common characteristics. Most importantly, processors that emphasize time-to-market, cost, and low power. Different from desktop processors, just-in-time computing takes precedence over the traditional maximum performance metric. The QSP32 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 QSP32 core, all instructions are fixed format 16-bit words with minimal decoding effort. As a result, Queue programs are much smaller than either RISC or CISC programs. Programs sizes for our architecture are found in our 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 [4, 5].

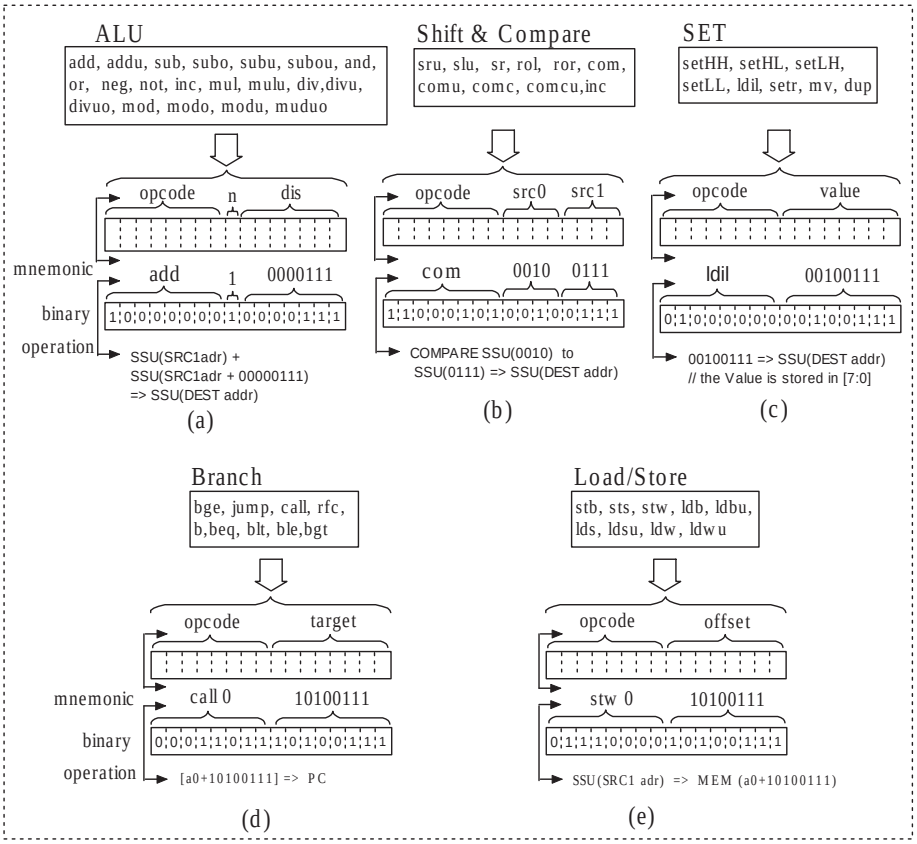
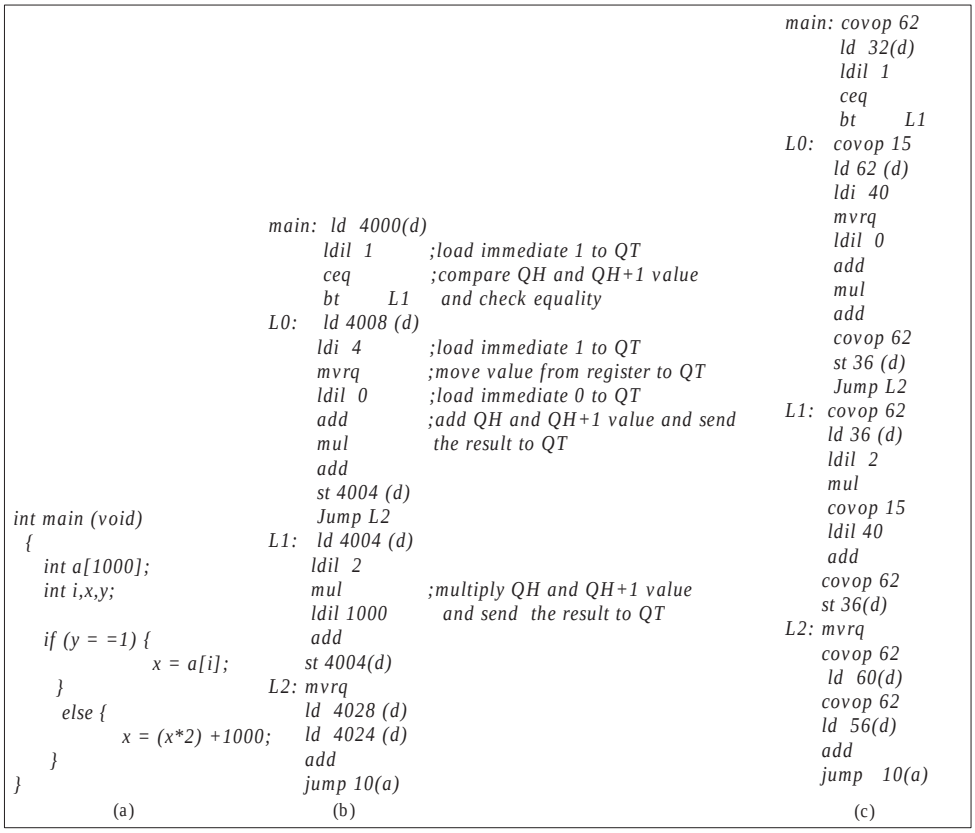


Fig. 3. Instruction format with several examples in Queue-EM mode: (a)*add* instruction, (b)compare instruction (*com*), (c)load immediate (*ldil*) instruction, (d)*call* instruction, and (e)store word (*stw*) instruction.

4.2 Reduced-Bit Instruction Set Design Consideration

The main issue in the design of general or even specific applications is the evaluation of the instruction set architecture. Adapting an instruction set to a particular problem is a difficult task, as many unknown issues have to be explored. Due to the many factors involved in performance optimizations, suggested optimization solutions often minimize only the number of instructions necessary to solve a problem, or at best the number of cycles. Our design approaches are based on software evaluation to establish cycle count, and synthesis to establish speed and chip area. Software evaluation was performed using a cycle accurate instruction set simulator described in the C programming language for Queue based instruction set architecture [2]. For hardware evaluation, we have performed logic synthesis to an FPGA technology for implementation information (discussed later).
The QSP32 processor uses a single shared instruction set. All instructions are 16-bit wide allowing simple instruction fetch, decode, and facilitating pipelining of the processor. The instruction format with several examples is illustrated in Fig. 3(a-e) with an example for each format.

In the current version of our implementation, we target the QSP32 core for medium-scale size applications, where our concerns focus on the ability to execute codes (Queue and Stack based applications) on a processor core with small die size and low power consumption characteristics when compared to other embedded 32-bit architectures. However, the short instructions may limit the memory addressing space as only 8-bits are left for displacement (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, QSP32 core implements *QCaEXT* technique, which uses a special hardware with a "covop" instruction that extends *load* and *store* instructions displacements and also extends immediate values if necessarily. The QSP32/PQP compiler [32] outputs full addresses and full constants and it is the duty of the QSP32 assembler to detects and inserts 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 [32]. Fig. 4 (b) shows an assemby program example after compilation of a sample *C* program shown in Fig. 4 (a). Fig. 4 (c) shows the assembly program output with "covop" instructions. For example, the instruction *ld* 4000(*d*) (in Fig. 4(b), first line) has long displacement value "4000" (in binary 111110100000), which can not fit in one single QSP32 instruction. Therefore, "covop" instruction is inserted to solve this shortage (Fig. 4(c)). For the 6-bit displacement value, the *ld* instruction (Fig. 4(c), second line) takes the lowest 6-bit (100000 (32 in decimal)) from the original displacement value (111110100000) and the remaining bits (111110 (62 in decimal)) of the dispalcement are the operand value of the "covop" instruction (Fig. 4(c), first line).
The "covop" instruction is coalesced with the immediately following instruction to generate a single QSP32 instruction at decode time (discussed later). This technique ensures that coalescing does not introduce pipeline delays or increase cycle time.



```
int main (void)
 {
   int a[1000];
   int i,x,y;

   if (y = =1) {
              x = a[i];
    }
     else {
          x = (x*2) +1000;
    }
}
```

(a)

```
main:  ld  4000(d)
       ldil  1          ;load immediate 1 to QT
       ceq              ;compare QH and QH+1 value
       bt      L1        and check equality
L0:    ld 4008 (d)
       ldi  4           ;load immediate 1 to QT
       mvrq             ;move value from register to QT
       ldil  0          ;load immediate 0 to QT
       add              ;add QH and QH+1 value and send
       mul               the result to QT
       add
       st 4004 (d)
       Jump L2
L1:    ld 4004 (d)
       ldil  2
       mul              ;multiply QH and QH+1 value
       ldil 1000         and send  the result to QT
       add
      st 4004(d)
L2: mvrq
       ld  4028 (d)
       ld  4024 (d)
       add
       jump 10(a)
```

(b)

```
main: covop 62
       ld  32(d)
       ldil  1
       ceq
       bt       L1
L0:   covop 15
      ld 62 (d)
      ldi  40
      mvrq
      ldil  0
      add
      mul
      add
      covop 62
      st 36 (d)
      Jump L2
L1:  covop 62
     ld 36 (d)
     ldil  2
     mul
     covop 15
     ldil 40
     add
    covop 62
    st 36(d)
L2: mvrq
    covop 62
     ld  60(d)
    covop 62
    ld  56(d)
    add
    jump   10(a)
```

(c)

Fig. 4. (a) Sample C program, (b) QSP32/PQP Compiler generated assembly program without "covop" instruction, (c) QSP32/PQP Compiler generated assembly program with "covop" instruction.

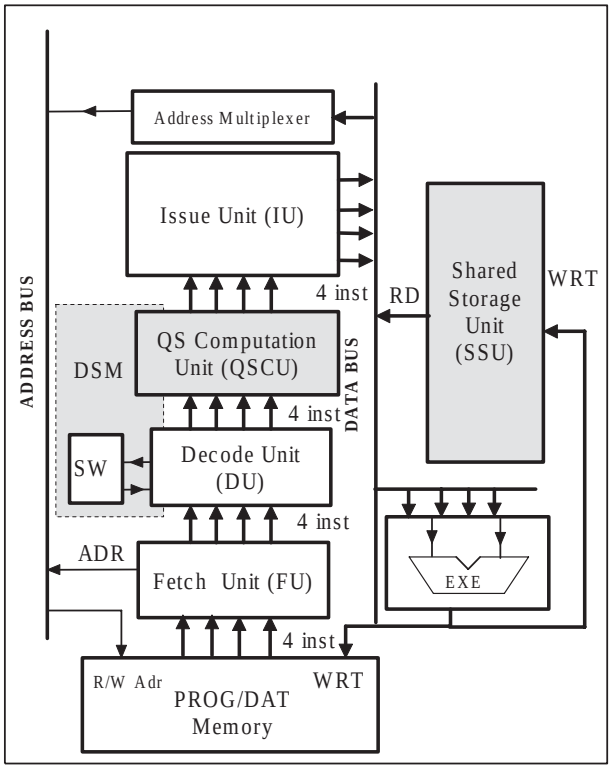


Fig. 5. QSP32 architecture block diagram. During RTL description, the core is broken into small and manageable modules using modular approach structure for easy verification, debugging, and modification.

5 System Architecture

5.1 Pipeline Structure

The block diagram of the proposed architecture is shown in Fig. 5. The execution pipeline operates in five pipeline stages combined with four pipeline buffers to smooth the flow on instructions through the pipeline. Below, we describe the salient characteristics of the core.
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 opcodes and operands. The decode unit (DU) has 4 decode circuits (DCs) and 1 Mode-Selector-Register (MS). The MS 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 computes each instruction's sources and destination locations.
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 sources and destination addresses.
Execution Unit (EXE): The EXE unit executes issued instructions and sends the results to the Shared Storage Unit (SSU) or data memory. It consists of four arithmetic logical units (ALU), four Set-register units, four Load/Store units and one Branch unit.
Write-Back Unit: The write back unit writes the result back to the PROG/DATA memory or the SSU.

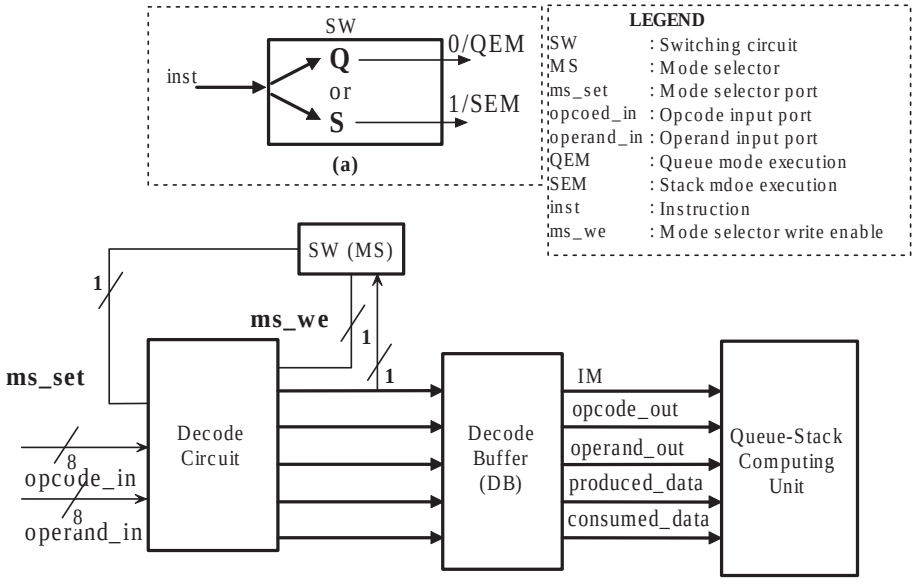


Fig. 6. Mode-Switching mechanism and its interfacing block diagram.

QSP32 uses a simple hardware mechanism (DSM) to dynamically switch between execution

models. The block diagram of the DSM is shown in Fig. 6. It consists of a switching circuitry (SW) and a dynamic computation unit [6]. The DSM detects the execution mode by decoding a special field, named "switch" instruction, within each instruction. After it detects the mode, it inserts a *mode-bit* for all instructions between the current and the next "switch" instruction.

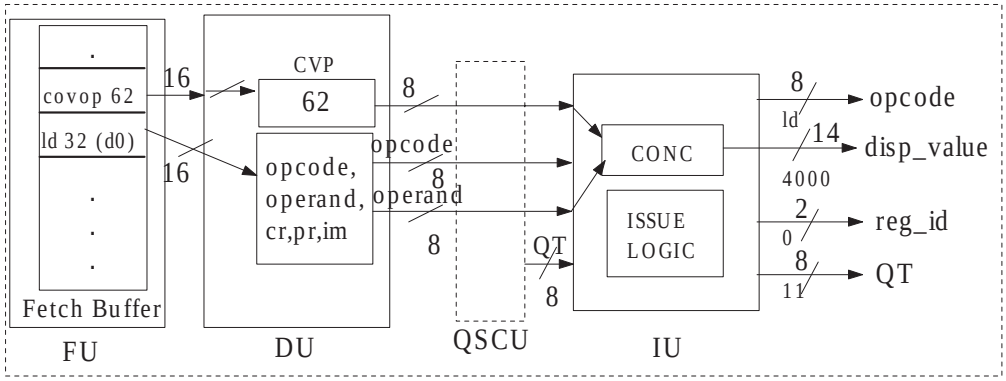


Fig. 7. Address extension mechanism.

As we mentioned, the QSP32 core uses *QCaEXT* scheme for extending memory displacement and immediate values. Fig. 7 shows the main circuit which implements "covop" instruction for extending memory displacement. The mechanism loads the displacement value in the register CVP (62 in this example). The value in the CVP register is then concatenated with the displacement value of the load instruction (32) at the issue stage and saved in a special register (CONC). Finally, the extended memory displacement (4000) is sent to the execution unit for effective address calculation.

5.2 Dynamic Calculation of Source and Destination Addresses

In Queue-EM execution mode, each instruction needs to know its QH and QT values. The above values are easy to obtain in 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 produced order parallel execution, which is supported by the QSP32 processor, the values for QH and QT are calculated at run time.

Fig. 8(a) shows the hardware mechanism used for calculating *source*1 (first operand), *source*2 (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 ($QH(n-1)$ in the figure) and QT ($QT(n-1)$) values for *source*1 and destination addresses respectively. As shown in Fig. 8(a), the *source*2 of a given instruction is the first calculated by adding the *source*1 address to the displacement ($OFFSET(n-1)$).

Fig. 9(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

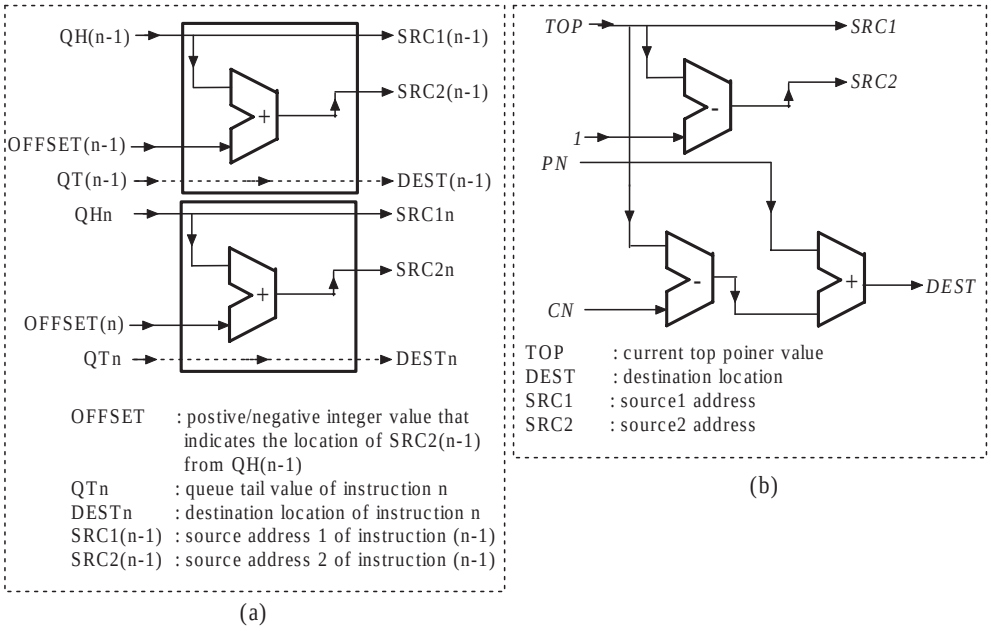


Fig. 8. Address calculation mechanism for sources and destination: (a) Queue-EM computing circuit; (b) Stack-EM computing circuit.

value (QH0) to find the next QH (NQH), and the number of produced data (PN) field (8-bit) is added to the current QT value (QT0) to find the next QT (NQT). The other three instructions sources and destination addresses are calculated similarly. In Stack-EM mode, the execution is based on Stack-EM model. The hardware used for calculating *source*1, *source*2 and destination addresses is shown in Fig. 8(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 each cycle. The *source*1 address is popped from the operand stack pointed by the current TOP pointer value. The *source*2 is calculated by subtracting 1 from current TOP pointer value. The number of consumed data (CN) is subtracted from the current TOP value and the number of produce data (PN) is added for finding the result address (DEST). Fig. 9(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 next instruction's source1 address. Fig. 10 (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) as shown in Fig. 10 (a). The "MS" is the *instruction-mode-selector* register and is set to "0" which means that the QSP32 core is in Queue-EM mode. The DU unit decodes instructions and calculates several fields for each instruction. As shown in Fig. 10 (a), the fields are: IM (instruction mode), OP (opcode), OPR (operand), CN (consumed number), and PN (produced number). For this example the values of IM, OP, OPR, CN, and PN are: $0, add, -1, 1, 1$ for the first instruction ($add\ -1$). The

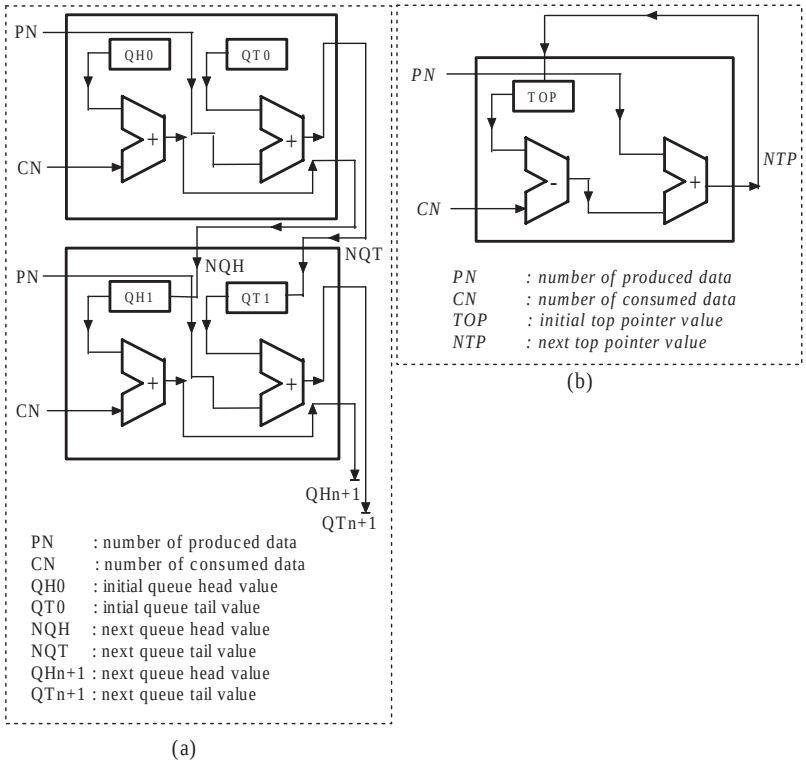


Fig. 9. Address calculation mechanisms for next instruction's source1 and destination: (a) Queue-EM mode; (b) Stack-EM mode.

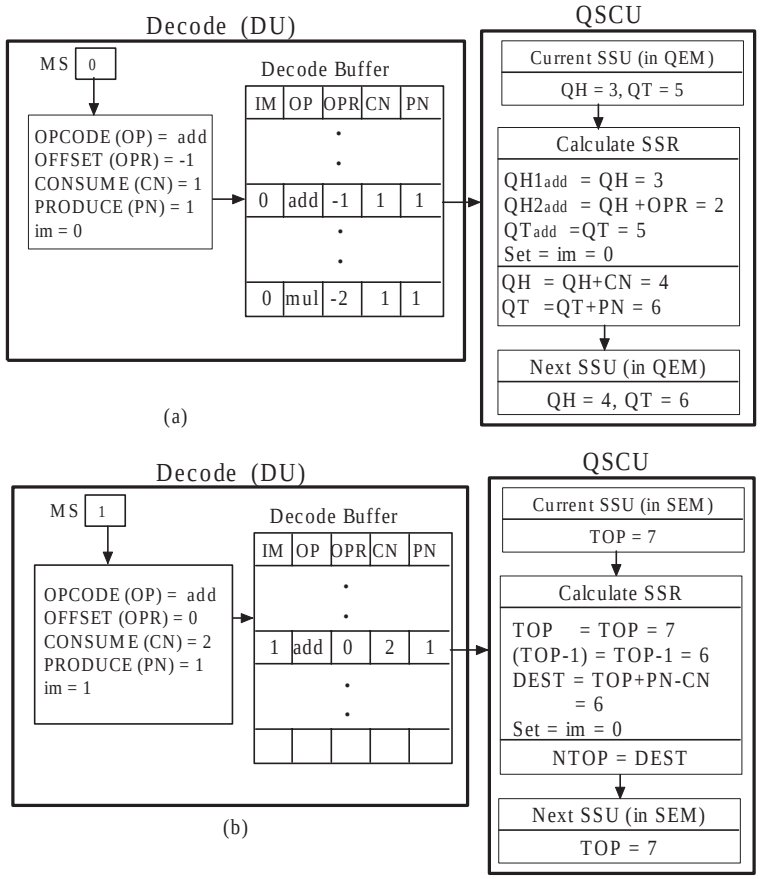


Fig. 10. Addresses calculation example: (a) When QSP32 is in Queue-EM mode; (b)When QSP32 is in Stack-EM mode.

same calculation scheme is performed for the following instructions. Using the mechanism shown in Fig. 8 (a) and Fig. 9 (a), source1 (QH1), source2 (QH2), and destination (QT0) addresses for each instruction are calculated in the QSCU stage. Fig. 10(b) shows another example illustrating the calculation of TOP, TOP-1, and DEST fields in Stack-EM mode.

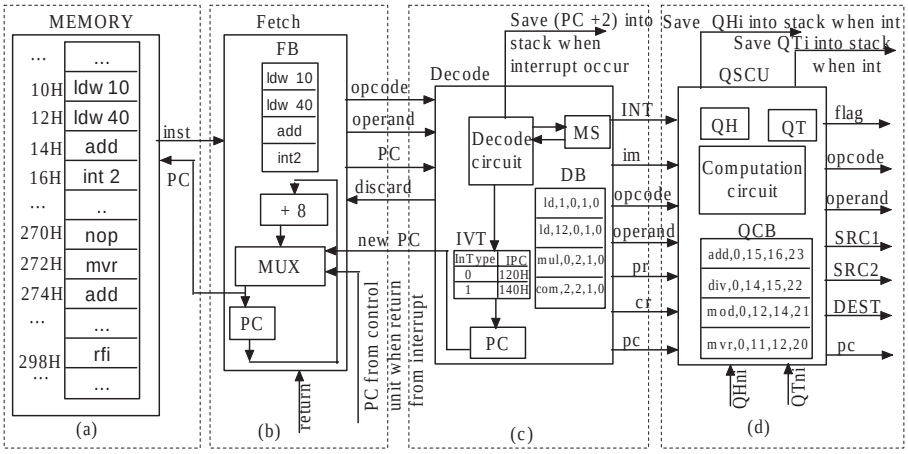


Fig. 11. Components used for Interrupt Handling Mechanism in QSP32.

5.3 Interrupt Handling in QSP32 core

The decode unit (DU) prepares the processor for correct handling of the interrupt. After an interrupt instruction in the DU is detected, the program counter (PC) of the next instruction is saved into the stack. The DU determines the address of the interrupt handler from the interrupt vector table (IVT) and places the value into the PC. At the same time the DU sends the *discard* signal to the fetch unit (FU) and the interrupt (INT) signal to the Queue computation unit (QSCU).

After receiving the *discard* signal and the new value for the PC, the fetch unit stops fetching instructions and resets the fetch buffer. Once finished, the FU resumes the fetching starting from the newest value of PC until the return signal coming from the execution unit is detected.

Program Counter Controller: Fig. 11(b) describes the Program Counter Controller (PCC) mechanism. There is a multiplexer (MUX) that selects the PC. The MUX is controlled by input signals coming from decode and execution units. Normally, MUX is selected by normal fetching mode where PC is always increment by 8 for each cycle. If there is *discard* signal from DU, the PCC is selected by the discard signal and it updates the PC by the new PC. When the PCC gets the return signal with return PC address from the execution unit, it sets the PC as return PC.

Queue Status Controller: Fig. 11(d) shows the block diagram of QSCU when the interrupt occur and Fig. 12 describes the details of Queue Status before and after the interrupt. Fig. 12 (a) shows the present Queue contents with present QH and QT pointer addresses. When the QSCU gets the *INT* signal from the DU, QSCU saves the current Queue status (current QH

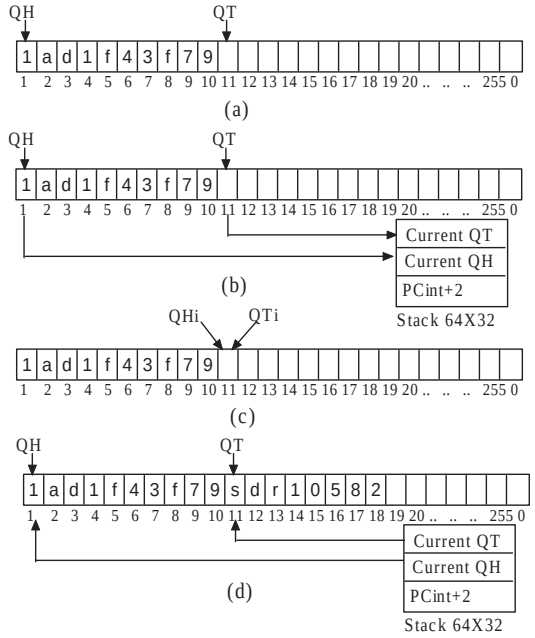


Fig. 12. Queue Status when interrupt occur and return from interrupt. (a) Queue status before interrupt, (b) when interrupt occur, (c) Queue: ready for interrupt handler, (d) Queue: when return from interrupt.

and QT values) into the stack (Fig. 12 (b)) and also changes the status of Queue. Fig. 12(c) shows the new QH (QHi) and QT (QTi) address for the interrupt handler. The QSCU will continue the computation by using the QHi and QTi value until it gets the return signal from Execution unit. When QSCU gets the return signal from EU it will restore the QH and QT address from the stack recovering the normal Queue with correct QH and QT pointer address (Fig. 12(d)).

5.4 Shared Storage Mechanism

We have implemented the *shared storage mechanism* in the shared storage unit (SSU). The shared storage unit (SSU) behaves as a circular QREG in Queue-EM mode and as a Stack data structure in Stack-EM mode. The SSU is an intermediate storage unit for QSP32 architecture. The SSU has 32-bit 256 shared registers (SSR) and behaves like a conventional register file (RF). However, in QEM, the system organizes the SSR access as a first-in-first-out (FIFO) latches, thus accesses concentrate around a small window and the addressing of registers is implicit through the queue head and tail pointers and in QEM, the system organizes the SSR access as a last-in-first-out (LIFO) latches which addressing the register is implicit through the stack pointer. The shared storage mechanism of SSR is controlled by the QSCU. The block diagram of SSU is shown in Fig. 13.

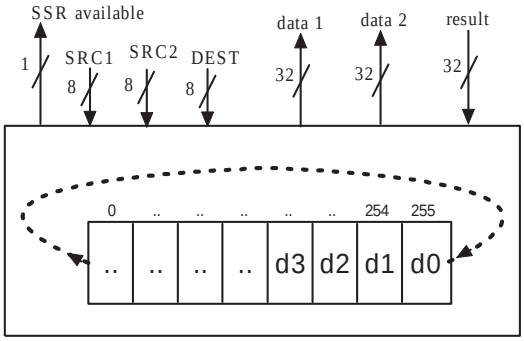


Fig. 13. Block diagram of shared storage unit.

6 QSP32 Core Implementation

6.1 Design Approach

To make the QSP32 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 [13, 14]. We have developed the QSP32 core in Verilog HDL. After synthesizing the HDL code, the designed processor gives us then the ability to investigate 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 QSP32 core to be useful for these purposes, we identified the following requirements: (1) High-level description: the format of the QSP32 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 QSP32 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 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 QSP32 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.

In order to estimate the impact of the description style on the target FPGAs 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 [7] for both speed and area optimizations. We selected Stratix FPGAs 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. Synthesis efficiency is influenced significantly by the match of resource implied by the HDL and resources present in a particular FPGA architecture. When a HDL description implies resources not found in a given FPGA architecture, those elements have to be emulated using other resources at significant cost. Such simulation can be performed automatically by electronic design automation (EDA) simulation tools in some cases, but may require changes in the HDL description in the worst case, counteracting aim of a common HDL source code base. We have used the Altera Quartus II design tools [9] for synthesis, simulation, placement, routing, and vendor supplied software for configurations. Simulations were also performed with Cadence Verilog-XL tool [8]. The synthesis result of the QSP32 processor over speed and area optimizations is shown in Table 2. From the above result, we clearly see how the optimization types affect the number of logic elements (LEs) for each unit and for the whole processor core.

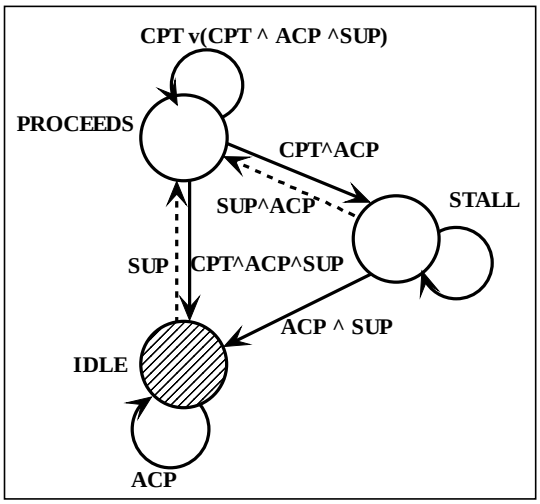


Fig. 14. Finite state machine transition for QSP32 pipeline synchronization. The following conditions are evaluated: next stage can accept data (ACP), previous pipeline stage can supply data (SUP), last cycle of computation (CPT).

6.2 QSP32 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 [15]. 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 indicate whether it can forward the data by using the *PROCEED* signal.

Table 1. QSP32 Processor Hardware Configuration Parameters.

QSP32 core

Items	Configuration	Description
IW	16-bit	Instruction width
FW	8 bytes	Fetch width
DW	8 bytes	Decode width
SI	56	Supported instructions
SSU	256	Shared storage unit size
ALU	4	Arithmetic logical unit
LD/ST	4	Load/Store unit
SET	4	Set unit
BRAN	1	Branch unit
GPR	16	General purpose registers
MEM	2048 word	PROG/DATA memory

Table 2. Synthesis results. LEs means Logic Elements. AOP means Area optimization and SOP means speed optimization.

Synthesis Results

Descriptions	Modules	LE-SOP	LE-AOP
Fetch unit	FU	345	252
Decode unit	DU	1395	1269
QS computation unit	QSCU	489	484
Issue unit	IU	1794	1588
Execution unit	EXE	7543	6213
Shared storage unit	SSU	12084	8120
QSP32 core	QSP32	23650	17926
PQP core (base)	PQP	23065	17556

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 adjacently stages), computing the next stage is simple. The state transitions of a pipeline stage in the QSP32 is illustrated in Fig. 14. 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 sub-states for reading initial addresses values, calculating next addresses values, and addresses fixup (when needed).

7 Results and Discussion

Table 1 shows the hardware configuration parameters of the designed QSP32 core. Table 2 summarizes the synthesis results of the QSP32 for the Stratix FPGAs. The complexities of each module as well as the whole QSP32 core are given as the number of logic elements (LEs). The design was optimized for balanced optimization guided by a properly implemented constraint table. From the prototyping result, the processor consumes 94.389% of the total logical elements of the target Stratix EP1S25F1020 FPGA device. As a result, the QSP32

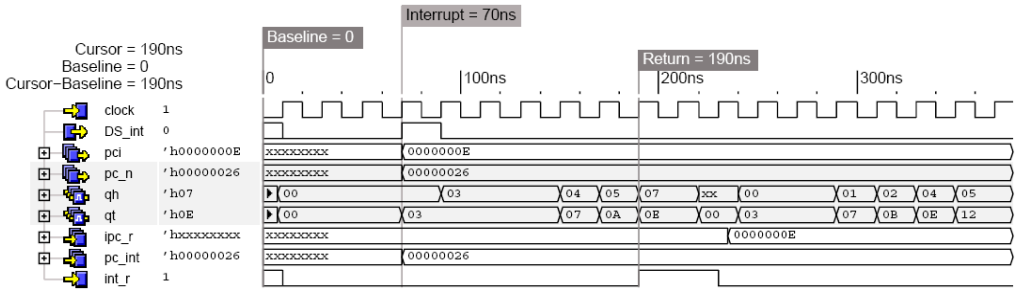


Fig. 15. Simulation result for interrupt controller QSP32 core

processor successfully fits on a single FPGA device, thereby obviating the need to perform multi-chip partitioning which results in a loss of resource efficiency.

Table 2 shows the total number of LEs for the QSP32 and the base architecture (PQP). When compared to the PQP core, QSP32 requires only 2.54% extra hardware for speed optimization (SOP) and 2.11% extra hardware for area optimization (AOP). The achievable frequency of the QSP32 core is 64.8 MHz and 62.31 MHz for speed and area optimizations respectively. The performance can be much more improved by using specific layout generation tools and standard libraries. We have tested the QSP32 core by using different programs. Fig. 15 shows the simulation result of interrupt handling mechanism of QSP32 core. When interrupt signal (DS_int) is active, the interrupt controller saves the processor state in stack (pci, qh, qt in Fig. 15). Fig. 15 shows that at 70ns an interrupt occurs and at that time qh and qt values were 0 and 3 respectively. From that time the controller generates an interrupted pc (pci = E in Fig. 15) and the interrupt handler location (pc_n = 26 in Fig. 15). Then it creates a new Queue from the QT pointer (3 in Fig. 15) of the main Queue and follows the new Queue location for each instruction source and destination address. When the interrupt controller gets the return from interrupt signal (int_r), it pops the saved processor state into the original location.

Table 3. QSP32 Speed and power consumption comparisons with various Synthesizable CPU cores over speed (SOP) and area (AOP) optimizations. This evaluation was performed under the following constraints: (1) Family: STRATIX; (2) Device: EP1S25F1020; (3) Speed: C6. The speed is given in MHz. NA means result not available.

Various Synthesizable CPU cores

Cores	Speed (SOP)	Speed (AOP)	Average Power(mw)
QSP32	64.8	62.31	187.5
PQP	71.5	70.1	187.5
OpenRisc1200	32.64	32.1	1005
SH-2	15.3	14.1	NA
ARM7	25.2	24.5	22
LEON2	27.5	26.7	458
MicroBlaze	26.7	26.7	NA

7.1 QSP32 Speed and Power Consumption Comparison with Synthesizable CPU cores

Performance of QSP32 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 [11, 12]. The SH-2 has RISC-type instruction sets and 16x32bit general purpose registers. All instructions have 16-bit 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 processor, the SH-2 also has an internal 32-bit architecture for enhanced data processing ability. LEON2 is a SPARCV8 compliant 32-bit RISC processor [27]. 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 produced by the manufacturer for hard core[25, 26]. From the results shown in Table 3, the QSP32 processor shows better speed performance for both area and speed optimizations when compared with the other listed processors, except for PQP processor, where QSP32 has 10.34% and 12.50% speed decrease for SOP and AOP optimizations respectively. QSP32 core also shows 144.27% less power consumption when compared with LEON2 and consumes much less power than OpenRisc1200 as shown in Table 3. However, QSP32 core consumes more power than the ARM7 processor, which also has less area than PQP and QSP32 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 QSP32 core parameters.

From the design results and comparison results it becomes clear that QSP32 core required low power consumption when compared with other conventional architectures that is very important for the mobile multimedia system area. On the other hand, the stack computation model is widely used in the internet and mobile system. As a result, the embedded QSP32 will be a good candidate for future mobile multimedia area.

8 Concluding Remarks

In this work we presented a novel dual-mode execution processor targeted for embedded and mobile applications. The QSP32 has a shared instruction set architecture and supports both Queue and Stack execution modes in a simple core. This is achieved dynamically with an *execution-mode-switching* and *sources-results-computing* mechanisms.
Form the evaluation results, we conclude that the proposed processor achieves a speed of about 64.8 and 62.31 MHz for speed and area optimization respectively. The processor consumes 94.389% of the total logic elements (LEs) of a Stratix FPGA 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 eliminating the need to perform multi-chip partitioning which results in a loss of resource efficiency. The core was implemented without considerable additional hardware when compared with the base PQP core (only about 2.54% more LEs are required). The QSP32 processor also shows better speed performance for both area and speed optimizations when compared with other well known Synthesizable CPU cores.

Finally, we conclude that the QSP32 core is expected to be a promising candidate for embedded and mobile multimedia applications requiring tight resource constraints and mutli-execution mode environments.

References

1. G. De Micheli, R. Ernst and W. Wolf, *Readings in Hardware/Software co-design*, Morgan Kaufmann Publishers, ISBN 1-55860-702-1.
2. M. Sowa, B. A. Abderazek and T. Yoshinaga,*Parallel Queue Processor Architecture Based on Produced Order Computation Model*, in: Int. Journal of Supercomputing, HPC, Vol.32, No.3, June 2005, pp.217-229.
3. B. A. Abderazek, T. Yoshinaga, M. Sowa, *High-Level Modeling and FPGA Prototyping of Produced Order Parallel Queue Processor Core*, in: International Journal of supercomputing, Volume 38, Number 1, October 2006, pp. 3-15.
4. P6 Power Data Slides provided by Intel Corp. to Universities.
5. B. Bisshop, T. Killiher, and M. Irwin, *The Design of a Register Renaming Unit*, in: Proceedings of Great Lakes Symposium on VLSI, 1999, pp. 34-37.
6. M. Akanda, Ben A. Abderazek, S. Kawata, and M. Sowa, *An Efficient Dynamic Switching Mechanism (DSM) for Hybrid Processor Architecture*, in: The proceedings of Springer's Lecture Note in Computer Science (LNCS), LNCS 3824, December 6-9, 2005, pp. 77-86.
7. D. Lewis et al, *The Stratix Logic and Routing Architecture*, in: FPGA-02, International Conference on FPGA, 2002, pp 12-20.
8. Cadence Design Systems:http://www.cadence.com/
9. Altera Design Software: http://www.altera.com/
10. B. A. Abderazek, M. Arsenji, S. Shigeta, T. Yoshinaga, M. Sowa, *Queue Processor for Novel Queue Computing Paradigm Based on Produced Order Scheme*, in: Proc. of HPC, IEEE CS, July 2004, pp. 169-177.
11. F. Arahata, O. Nishii, K. Uchiyama, N. Nakagawa.,*Functional verification of the superscalar SH-4 microprocessor*, in: Compcon97, the Proceedings of the International conference Compcon97, Feb 1997, pp. 115-120.
12. SuperH RISC engine SH-1/Sh-2/Sh-DSP Programming Manual: http://www.renesas.com
13. H. Maejima, M. Kinaga and K. Uchiyama, *Design and architecture for Low Power/High Speed RISC Microprocesor:SuperH*, in: IEICE Transaction on Electronics, Vol.E80, No.12, dec. 1997, pp.1539-1549.
14. H. Takahashi, S. Abiko and S. Mizushima, *A 100 MIPS High Speed and Low Power Digital Signal Processor*, in: IEICE Transaction on Electronics, Vol.E80-C, No.12, 1997, pp.1546-1552.
15. R. Lysecky and F. Vahid, *A Study of the Speedups and Competitiveness of FPGA Soft Processor Cores using Dynamic Hardware/Software Partitioning*, in: Design Automation and Test in Europe (DATE'05),Munich, Germany, Vo.1, March 2005, pp. 18-23.
16. J. P. Koopman, *Stack Computers: the new wave*, Ellis Horwood Limited, 1989.
17. M. Sheliga and E. H. Sha, *Hardware/Software Co-design With the HMS Framework*, in: Journal of VLSI Signal Processing Systems, Vol. 13, No.1, 1996, pp. 37-56.
18. K. Kim, H. Y. Kim and T. G. Kim, *Top-down Retargetable Framework with Token-level Design for Accelerating Simulation Time of Processor Architecture*, in: IEICE Trans. Fundamentals of Electronics, Communications and Computer Sciences, Vol. E86-A, No. 12, Dec. 2003, pp.3089-3098.
19. http://www.arm.com/products/CPUs/ARM926EJ-S.html
20. JazelleTM-ARM Architecture Extensions for Java Applications, White Paper, http:// www.arm.com
21. Advancel Logic Corporation, Tiny2J Microprocessor Core for Javacard Applications, http://www.advancel.com
22. http://ultratechnology.com/
23. Oyvind Strgm, Einar J. Aas., *An Implementation of an Embedded Microprocessor Core with support for Executing Byte Compiled Java Code*, in: Proceedings of the Euromicro Symposium on Digital Systems Design, 2001, pp. 396-399.

24. Harlan McGhan and Mike O'Connor, *PicoJava: A direct execution engine for Java bytecode*, in: Computer 31(10), October 1998, pp. 22-30.
25. ARM7DMI Data Sheet, Advanced RISC Machines Ltd, 1994.
26. ARM Architecture Reference Manual, Advanced RISC Machines Ltd., September 2001.
27. Gaisler Research Laboratory. LEON2 XST User's Manual, 1.0.22 edition, May 2004.
28. B. R. Preiss, V. C. Hamacher, *Data Flow on Queue Machine*, in: ISCA 1985, 12th International Symposium on Computer Architecture, Boston, August 1985, pp. 342-351.
29. M. Fernandes, J. Llosa, N. Topham, *Using Queues for Register File Organization in VLIW*, Technical Report ECS-CSG-29-97, University of Edinburgh, Department of Computer Science, 1997.
30. L. S. Heath, S. V. Pemmaraju, A. N. Trenk, *Stack and Queue Layouts of Directed Acyclic Graphs: Part I*, in: SIAM Journal of Computing, Vol 23, No. 4, 1996, pp.1510-1539.
31. H. Schmit, B. Levine, B. Ylvisaker, *Queue Machines: Hardware Compilation in Hardware*, in: FCCM'02, 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2002, pp. 152-161.
32. A. Canedo, *Code Generation Algorithms for Consumed and Produced Order Queue Machines*, Master Thesis, Graduate School of Information Systems, University of Electro-Communications, September 2006.