RECONFIGURABLE RADIO WITH FPGA-BASED APPLICATION-SPECIFIC PROCESSORS

Rob Jackson (Altera European Technology Centre; High Wycombe; UK; rjackson@altera.com) Sambuddhi Hettiaratchi (European Technology Centre; High Wycombe; UK: shettiar@altera.com); Mike Fitton (European Technology Centre; High Wycombe; UK; mfitton@altera.com); Steven Perry (European Technology Centre; High Wycombe; UK; sperry@altera.com)

ABSTRACT

FPGAs are a viable solution for the computationally intensive signal processing requirements in software-defined radio (SDR). The conventional approach to partially or fully reconfigure the device to meet different requirements results in an associated delay and/or over-provision of resources required. We demonstrate a new approach based on application-specific integrated processors (ASIPs) where the FPGA is configured to provide a number of functional units controlled by a simple processor and associated program. Reconfiguration then merely requires modification of the program and may be performed quickly and simply.

1. INTRODUCTION

FPGAs are widely used for computationally intensive signal processing applications. We describe a novel approach to implementing algorithms on FPGAs based on ASIPs.

An ASIP combines the flexibility of a software approach with the efficiency and performance of dedicated hardware. It is a processor that has been specialised to perform certain tasks or classes of tasks efficiently and with a required level of performance. An ASIP may be derived from a general purpose processor by varying the number and type of function units (arithmetic, logical, load/store, register-file), introducing new types of application-specific function unit (multiply-add, permute, address generation), and by changing the internal topology of processor—i.e., the pattern of interconnection between function units.

Digital signal processors and network processors can be viewed as ASIPs targeting specific classes of application. In general, the more specialised the processor, the greater the efficiency and less the applicability.

There are a number of advantages to implementing an ASIP using an FPGA. While a processor implemented on a FPGA may be larger and slower than one implemented as an ASIC, they use the same process technology. FPGAs are typically fabricated using a more advanced process than is available to the majority of ASIC applications. An FPGA processor implementation is also able to benefit from improvements in process technology as they are introduced with new generations of FPGA devices. FPGAs have become large enough to contain a complete system on a single device, hence the name "system on a programmable chip" (SOPC). The single-chip design allows tight integration between multiple processors, memories, and other system components [1].

We believe building ASIPs using FPGAs has further advantages. Modern FPGAs have all the building blocks necessary to build a processor: digital signal processing (DSP) blocks (multiply-accumulators), small and large RAMs, and fast arithmetic and logic. The FPGA fabric supports variation of the number, type, and topology of the function units and creation of new types of function unit. Highly specialised FPGA-based ASIPs can be quickly and cheaply produced and offer very high levels of efficiency.

Furthermore, for computationally intensive tasks that require a hardware solution to meet performance requirements, the ASIP-on-FPGA model has a number of benefits over traditional design flows using hardware description languages like VHDL and Verilog.

In contrast to the traditional hardware design flows, the ASIP approach naturally separates behaviour (algorithm) from implementation (architecture). For example, sharing (or reuse) of hardware blocks is much more naturally expressed in terms of a processor. In a hardware description language, multiplexers and control signals must be written explicitly, whereas they are implicit in a given ASIP architecture.

The assembler and compiler tools available for an ASIP manage pipeline delays, schedule operations, and automatically generate a control program. These tasks are performed manually when writing a hardware description.

The ASIP program is explicitly stated using a sequential programming model with instruction-level parallelism. A hardware description implicitly encodes the program in the behaviour of a number of parallel processes.

Control flow is expressed as a series of arithmetic and logical operations applied to a set of variables. In an ASIP this may be implemented using a register file and an arithmetic logic unit (ALU) with each compare-and-branch

operation performed sequentially. A hardware implementation may include a finite state machine, multiple registers, and comparators, and perform multiple comparisons in a single step (an N-way branch). However, in many computationally intensive applications, control flow represents a small fraction of the total number of operations and is not performance critical. The smaller, denser ASIP may require more cycles but may also operate faster. The ASIP may also be quickly and easily reprogrammed by changing the contents of the program memory. A hardware approach will require the FPGA to be reconfigured to make even trivial changes to its behaviour.

As a result of a higher degree of algorithm-implementation separation, the ASIP approach is more amenable to design automation and design space exploration than traditional hardware design flows. Moreover, this technique permits an easier migration from programmable logic to ASICs without compromising its ability to be reconfigured.

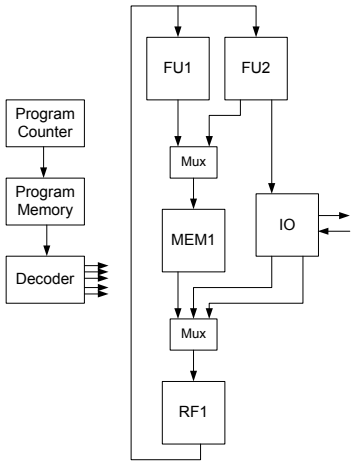
2. CONSTRUCTING AN ASIP

2.1 Analysis and Design

The first step in constructing an ASIP is to analyse the application that is to be executed. A simple analysis can yield characteristics such as the amount of memory required and the number and type of operations. Given a real-time constraint such as throughput or latency, we can translate algorithm requirements into resource requirements: i.e., a simple implementation of a 15-tap finite impulse response (FIR) filter requires 15 multiply-add operations, so if given a throughput of 1,000,000 samples per second, we require resources to provide 15 MMACS (million multiply-accumulated operations per second).

By comparing the algorithmic resource requirements against the features and performance of various function units, we can begin to define an ASIP architecture.

Figure 1. Architecture of an ASIP



Notes:

1. FU = function unit
2. Mux – multiplexer

The availability, classes, and performance of function units depend on the FPGA device targeted. For instance, using the DSP blocks on Altera's largest Stratix® II device, we could construct 384 18-bit multiply-accumulate function units with appropriate rounding and saturation, and each performing 370 MMACS [2]. A straightforward 32-bit add/subtract unit implemented using logic cells can perform 350 million add/subs per second.

Memory requirements are often more important than computational requirements. The ASIP architecture must include sufficient memory units to satisfy the bandwidth requirements of the algorithm. Modern high-density FPGAs typically contain a range of memory resources which can be combined or configured to provide different sized memories.

Designing the ASIP architecture is an iterative process whereby the processor architecture and the algorithm are progressively refined. The cost of candidate processor architecture may be estimated relatively easily by counting the resources used. The performance of the algorithm can be assessed by a more detailed analysis of implementation of the key sections of the application (such as innermost loop bodies).

Reaching the optimal implementation using this approach may, in general, require a number of iterations. However, we find in practise that an acceptable architecture can be designed for many applications relatively quickly.

For applications that have been implemented in software or in previous hardware generations, modern FPGAs often provide vastly more computational power than is required. We can, therefore, apply standard architectural templates to generate ASIP architectures: for DSP-like algorithms, an architecture template with N memories feeding M multipliers and accumulators, and for control applications, a register file feeding an ALU.

2.2 Architecture Definition

Typically a microprocessor is defined in terms of a non-application-specific instruction set. This instruction set is somewhat abstract, often relying on a set of logical register names. An instruction-set-based processor will require logic to:

- Detect hazards: for example, when a register is read but its value has not been completely computed
- Forward results: either by directly moving a computed value from the output of one function unit to the input of another while bypassing the register file, or by

detecting which physical location contains the current value of a logical register name

In an FPGA, many function units are pipelined and so the number of hazard conditions and the number of forwarding paths will be large.

When a function unit can receive its arguments from a number of sources, a multiplexer must be included to select between them. The multiplexer must be constructed out of logic cells, thereby adding to the resources needed for the processor and potentially slowing its execution. A four-to-one multiplexer implemented in an Altera® Stratix FPGA will consume twice as many resources as an add/sub unit.

When constructing an ASIP, we include no hazard detection hardware. We require the assembler/compiler to statically schedule instructions so that no hazards occur. Moving this analysis to compile time means the processors generated are smaller and can run faster.

However, this form of static scheduling may lead to inefficient programs. Some hazards are data dependent either due to control flow or through address generation and cannot be determined until the program executes. A static schedule must be pessimistic to ensure correctness. For most applications, fortunately, the inefficiency we encounter in the program is more than compensated by obtaining a faster and smaller processor. In DSP algorithms, there is often little data-dependent execution, and memory accesses are regular and analyzable, so the potential benefits of run-time hazard detection are minimal.

When constructing an FPGA-targeted ASIP, we only include those routing paths which are required by the application. The need for multiplexers is removed or reduced, resulting in a smaller and faster processor. The resulting ASIP architecture has much in common with a transport-triggered architecture (TTA) and the finite state machine plus datapath (FSM-D) architecture generated by many high-level behavioural synthesis tools [3][4].

Figure 1 shows the general form of an FPGA-based ASIP. Pipelined memory and counter programs supply the machine with an encoded instruction word. The memory program is typically included within the processor and exploits the dual-port facilities of the memories to allow external sources to load program code.

The encoded instruction word feeds a decode block that decodes the data to provide a set of control signals for the processor. Control signals include immediate values such as literals, register file read and write addresses, function unit enable and operation select signals, and multiplexer operand-select codes.

The processing core includes a set of function units and the multiplexers that route data between them. The function units include memories, registers, basic arithmetic and logic units, and multiply-add blocks. These blocks may exploit specific features of the FPGA device or may rely on standard libraries such as the library of parameterized modules (LPM) [5]. In addition, custom application-specific units may be included.

Function units implementing bus-masters, slaves, general purpose I/O (GPIOs), and streaming point-to-point protocols provide I/O functionality.

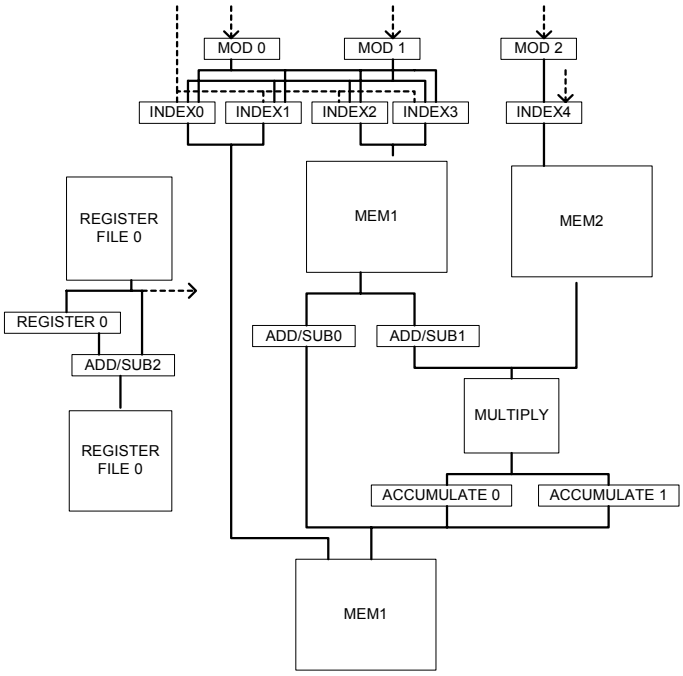


Figure 2. Combined FFT/FIR ASIP Architecture

2.3 Program

Each ASIP defines its own custom assembly language. For each processor we also construct a custom assembler which translates an executable specification of the algorithm into the correct control signals. The assembler schedules operations and loops to avoid hazards and minimise execution time.

3. RECONFIGURABLE RADIO

SDR implements algorithms in a software form to improve portability, lifetime costs, and retargetability. However, achieving cost and performance requirements necessitates the use of application-specific hardware. Fast Fourier transform (FFT) and FIR filters are representative of the typical algorithms used in SDR. Therefore, we use FFT and FIR implementations to demonstrate the effectiveness of the ASIP on FPGA methodology.

3.1 Fast Fourier Transform

The details of the FFT algorithm and its implementation are well understood and widely known [6][7], so we will not describe them in detail here. The N-point FFT can be implemented as a series of $\log_2(N)$ passes across the source data set. Each pass requires N/2 multiplies. Therefore, a 1024-point FFT requires 5120 multiply operations.

Our FFT ASIP is primarily composed of a dual-ported data memory and a single-ported coefficient memory with associated address generation units. A single 16 x 16 multiplier and a 16-bit arithmetic block are included. We include an accumulator and single-ported register file (memory) to act as a simple general purpose controller.

3.2 Finite Impulse Response

The FIR filter is the building block of many DSP applications. We chose the simplest FIR structure that requires a single ported data and coefficient, a multiplier, along with an accumulator.

3.3 Combined FFT/FIR

The architecture of the processor we have specified is shown in Figure 2. In effect, we have built a simple digital signal processor capable of performing FIR and FFT. The instruction set exposes all the registers and potential hazards. The FFT and FIR algorithms are expressed in terms of the operations the machine performs and the tools automatically schedule each operation to avoid pipelined hazards.

The final ASIP can be considered as either an application-specific VLIW digital signal processor or a hardware block implementing the FFT algorithm using a microcoded FSM.

The hardware is simple and small (322 Logic Elements (LEs), one 16 x 16 multiplier) and fast (230 MHz in a Stratix -5 part). A 1024-point radix-2 complex FFT takes 21850 cycles and executes in approximately 95 µs. Moreover, the FFT/FIR ASIP takes less than 5 percent of the available LEs and DSP blocks on the smallest Altera Stratix device.

In comparison, a complex radix-2 FFT implemented on a C62x DSP device from Texas Instruments would take 20840 cycles at 300 MHz (69.5 µs) to compute a 1024 point FFT [8]. Although the DSP solution is faster than the specific ASIP implementation reported here, a DSP solution requires an entire digital signal processor, while the ASIP on FPGA solution requires less than five percent of a Stratix device.

Another alternative solution would be to use an IP Core optimised for a particular family of FPGAs. The Radix-4 FFT IP core from Xilinx, for example, takes 1869 logic slices (a Xilinx logic slice contains four 4-input look-up tables (LUTs) compared to an Altera LE containing one 4-input LUT) and takes 4145 clock cycles at 100 MHz (41.45 µs) [9]. A radix-4 FFT implemented using our ASIP on FPGA methodology takes about 6368 cycles at 256Hz (under 25 µs) and 600 LEs.

4. CONCLUSION

We have described a methodology for constructing customised application-specific processors targeting an FPGA. We believe that this architectural style leads to efficient algorithm implementations in modern FPGA devices, with a FFT/FIR processor as an example. FPGA-implemented ASIPs are a good target for SDR as they give the performance of an FPGA but can be re-programmed without having to complete a hardware change cycle with its associated risks. They also remain reprogrammable if the FPGA design is converted to a structured ASIC like Hardcopy®.

5. REFERENCES

[1] Altera Corporation, *NIOS II Processor Reference Handbook*, http://www.altera.com/literature/lit-nio2.jsp, 2004.

[2] Altera Corporation, *Stratix II Device Handbook*, http://www.altera.com/literature/lit-stx2.jsp, 2004.

[3] H. Corporaal, *Microprocessor Architectures from VLIW to TTA*, Wiley, Chichster, 1998.

[4] D. Gajski, N. Dutt, and A. Wu, *High-Level Synthesis*, Kluwer, Boston, 1992.

[5] Electronic Industry Alliance, "EIA/IS-103-A Library of Parameterized Modules." http://www.ediforg/lpmweb, 1999.

[6] P. Duhamel and M. Vetterli. "Fast Fourier Transforms: A Tutorial Review." *Signal Processing,* Vol. 19, pp 259-299, 1990.

[7] G.D. Gergkand "A Guided Tour of the Fast Fourier Transform." *IEEE Spectrum,* Vol. 6, pp 41-52, July 1969.

[8] C62x DSP Benchmarks, Texas Instruments, Inc., latest version.

[9] LogiCore High Performance 1024-Point Complex FFT/IFFT, Xilinx, Inc., July 2000.