This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TCAD.2014.2363388

Soft-Core Dataflow Processor Architecture Optimised for Radar Signal Processing

R. Broich*† and H. Grobler*
*Department of Electrical, Electronic and Computer Engineering, University of Pretoria
†DPSS Radar and Electronic Warfare, CSIR, South Africa
Emails: RBroich@csir.co.za, Hans.Grobler@up.ac.za

***Abstract*—Current radar signal processors lack either performance or flexibility. Custom soft-core processors exhibit potential in high-performance signal processing applications, yet remain relatively unexplored in research literature. In this paper, we use an iterative design methodology to propose a novel soft-core streaming processor architecture. The datapaths of this architecture are arranged in a circular pattern, with multiple operands simultaneously flowing between switching multiplexers and functional units each cycle. By explicitly specifying instruction level parallelism and software pipelining, applications can fully exploit the available computational resources. The proposed architecture exceeds the clock cycle performance of a commercial high-end DSP processor by an average factor of 14 over a range of typical operating parameters in a radar signal processor application.**

***Index Terms*—Soft-core DSP, Transport-Based Processor, Signal Processing Architecture, Radar Signal Processor, Soft-core Processor, Processor Design Methodology, Streaming Architecture, Circular Dataflow.**

I. Introduction

In modern radar systems the architecture of the radar signal processor (RSP) is one of the most important design choices. The amount of useful information that can be extracted from the radar echoes is highly dependent on the computational performance that the RSP can deliver. Current RSPs lack either performance or flexibility in terms of ease of modification and large design time overheads. Combinations of processors and field-programmable gate arrays (FPGAs) are typically hard-wired together into a precisely timed and pipelined solution to achieve a desired level of functionality and performance. To address this gap between performance and flexibility, a custom processor architecture is proposed.

This paper is organised as follows: Current RSP processing technologies are compared in Section II, emphasising the need for a programmable radar processing architecture. The computational characteristics and requirements of radar algorithms are identified in Section III and used to derive the conceptual architecture in Section IV. The optimisation process is described in Section V, with the final architecture being presented with an example in Sections VI and VII. The FPGA implementation and the final performance results are then discussed in Section VIII. Finally, Section IX summarises the characteristics of the proposed architecture and concludes this paper.

II. Current RSP Processing Technologies

Early digital radar systems relied heavily on custom ASIC implementations to achieve the required performance, as other technologies were simply not available. More recent systems typically use combinations of DSPs, PCs and reconfigurable logic as processing technologies, increasing flexibility and minimising non-recurring engineering costs.

The majority of RSPs rely on FPGAs for both analogue converter interfaces as well as processing tasks, often implementing the entire RSP on a single FPGA [1]. RSP operations such as I/Q demodulation, filtering, channel equalisation and pulse compression are usually implemented as a fixed pipeline of streaming operations in the traditional hardware description language (HDL) design flow [2]–[6].

Although these designs are parametrised and thus configurable to a limited extent, they are not programmable. With evolving requirements in the constantly changing radar processing field, the traditional HDL based design flow lacks flexibility in terms of ease of modification and design time overhead. Some notable attempts have been made to improve the overall design process of FPGA systems; software abstraction layers [7], library based tool chains [8], rapid implementation tool-suites for translating high-level algorithms into HDL [9], [10] and high-level synthesis tools [11]. Regardless, there seems to be a struggle to balance speed, flexibility and ease of implementation.

The embedded processor approach shifts the design methodology to a sequential execution paradigm, while still providing tight coupling to the FPGA resources. This embedded software approach offers substantial flexibility advantages over other HDL approaches, including ease of use, quick design changes and easy debugging. However, embedded hard-core (e.g. ARM, PowerPC) or generic soft-core (e.g. NIOS II, MicroBlaze) processors are limited in computational performance, mostly serving simple control, configuration, interface or supervisory roles in RSPs.

Custom soft-core processors have high performance potential in the DSP domain, especially with vectorisation techniques [12]–[14]. However, among the vast amount of implementation details relating to RSPs, custom soft-core processors remain largely unexplored in the radar and high performance computing domain.

III. Computational Characteristics

The first step to any architectural design is to identify the computational characteristics and prominent operations of the target applications. Typical radar signal processing algorithms were previously discussed and broken down into common digital signal processing operations [15], [16]. Fig. 1 summarises the performance requirements that were identified for various implementation alternatives.

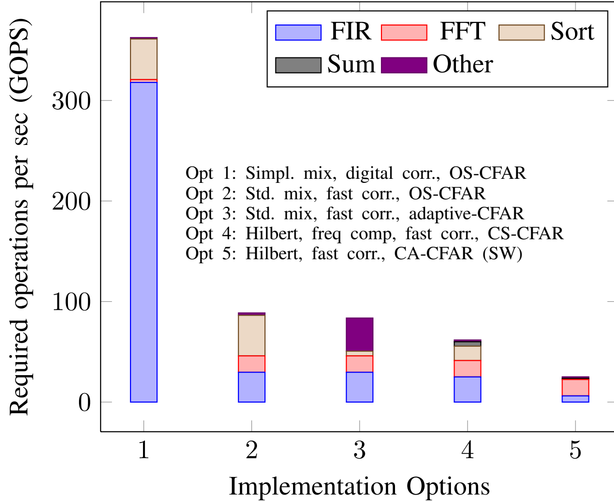


Fig. 1: Computational requirements of a radar signal processor

Based on the selected implementation option, the computational requirements range between 25 billion operations per second (GOPS) and 363 GOPS, comprising mostly of finite impulse response (FIR), fast Fourier transform (FFT) and sorting operations. Other operations that are common in the radar signal processing field are convolutions, vector operations, block summations, matrix multiplications and basic arithmetic instructions.

In addition to ensuring that the architecture is optimally suited for handling the prominent operations, it is also important to define the computational characteristics that dominate this processing field. The signal and dataflow characteristics of the radar signal processing algorithms are highly regular with a linear data independent processing chain. The following list summarises the most important computational characteristics of radar signal processing:

- High performance: 350+ GOPS
- Mostly FIR and FFT operations
- Low latency
- Tight coupling to analogue converters and data processor
- Data independent processing chain
- Deterministic and regular dataflow
- Minimal branching, no interrupts
- Large dynamic data range
- Alternating horizontal and vertical memory accesses

In the general purpose and DSP processing paradigm, the challenges of attaining significant utilisation of the raw computational resources have been overcome with various instruction set optimisations and micro-architectural techniques. In the streaming processing paradigm however, some of these techniques are actually detrimental to the application performance. Task-, data- and instruction-level parallelism is inadequately captured in most high-level programming languages, and further obscured during compilation into a sequential instruction stream. As a result, the hardware-based dynamic scheduling mechanisms cannot extract sufficient parallelism from the instruction stream, and the low-level computational resources remain underutilised.

The regular instruction stream and data access patterns of most stream processing applications enable static scheduling with large degrees of parallelism, provided that the programmer/compiler has explicit control over the low-level processing resources. The general purpose processing optimisations and techniques inherently limit this low-level control. For this reason an architecture with much finer control over each low-level computational resource is proposed. The following list summarises some of the desirable and undesirable features of a radar signal processor.

- ✓ Deep Pipelines
- ✓ Vectorisation and SIMD
- ✓ Multiple cores
- ✓ Instruction-level parallelism (VLIW, EPIC)
- ✓ Hardware looping mechanism
- ✓ Dedicated address generation unit
- ✓ Deterministic dual-ported memory accesses
- × Hardware scheduling
- × Memory caching
- × Data bus arbitration
- × Register renaming and rotating register files
- × Out-of-order execution
- × Interrupts
- × Branch prediction and speculation buffers
- × MMU, virtual memory
- × Central register file
- × High-level abstraction

These characteristics differ substantially from those of current processing architectures, which typically focus on higher levels of abstraction and task-level parallelism.

IV. Conceptual Architecture

A new processing architecture which is well suited to these computational characteristics was designed from first principles. This novel template architecture is shown in Fig. 2. Rather than an instruction word controlling the execution units, the instruction word defines how data is routed between the lower level functional units. The switching matrix consists of multiple simple multiplexers, each controlled independently by a unique slice of the program word.

Data values circulate in a counter-clockwise direction, passing from a register through a functional unit back into a selected register to form a software pipeline. For functional units that are not clocked (e.g. integer adder), each data value completes one revolution per clock cycle. Clocked functional units can make use of deep pipelines and produce a new output value each clock cycle. It is thus possible to assign any functional unit output to each register every clock cycle.

V. Optimisation Process

To optimise this template architecture for the radar application, the fine grained details and trade-offs between number

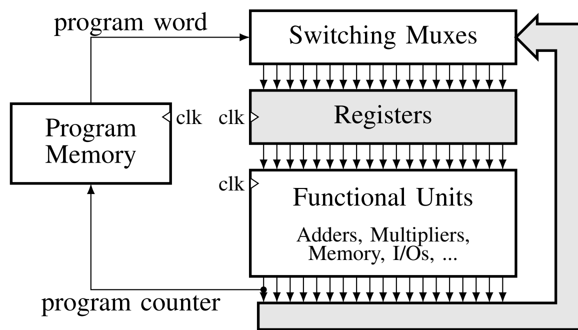


Fig. 2: Basic Processor Architecture

and type of functional units as well as register connections are investigated. The refinement process consists of alternating processor definition, practical or theoretical implementation, and application profiling stages. The application profiling stage includes functional as well as performance verification, but also takes other factors such as resource usage and power consumption into consideration. If the designer is not satisfied with any of these factors, the architecture is modified and the process repeats.

This detailed optimisation process involves small incremental architectural changes for each design iteration - a task of which many aspects can be automated. A software development environment was designed to automate this architectural design space exploration phase, bearing a close resemblance to the ASIP design methodology [17], [18]. The design flow of this software-based design approach is shown in Fig. 3. The shaded components are integrated and generated by the software development environment.

The architecture is defined by an `*.ARCH` file, which forms the foundation of the entire software development environment. This architecture description consists of a list of the various functional units, registers and their interconnections. Based on the `*.ARCH` file, the VHDL source files for the processor core implementation, a graphical depiction of the processor architecture, and all the required development tools such as code editor, assembler, linker, cycle accurate emulator / simulator, debugger and programmer are generated.

Together with the board specific HDL-based hardware abstraction layer (HAL) files, the generated VHDL design files are then synthesised using the vendor specific FPGA tools (in this case Xilinx ISE, but similarly on Altera Quartus II). Timing results, functional accuracy, resource usage, profiling and performance data are then analysed and used by the designer to further refine the architecture through the architecture description file. The generated `*.BIT` file can also be loaded onto the development board for practical verification.

The left side of the design flow in Fig. 3 is more concerned with the application development and simulation aspect of the processing architecture. The code editor provides syntax highlighting and dynamic code completion mechanisms for the custom `FLOW` language based source files. Functional verification and performance profiling are important aspects of the simulator during algorithm development. The development environment enables efficient graphical design feedback for debugging and optimisation of the architecture, automating many of the tedious and error-prone tasks that are usually involved in the optimisation process.

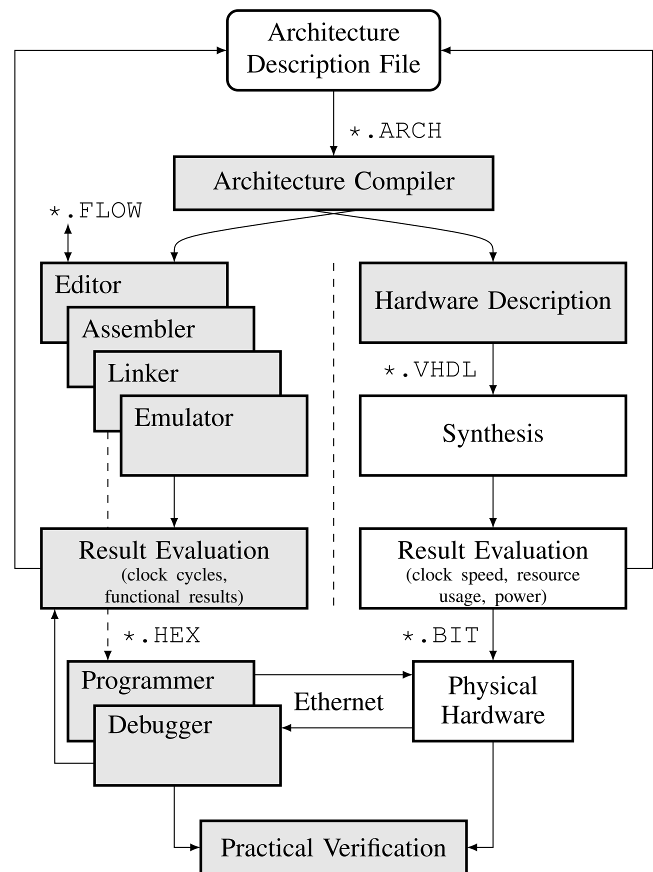


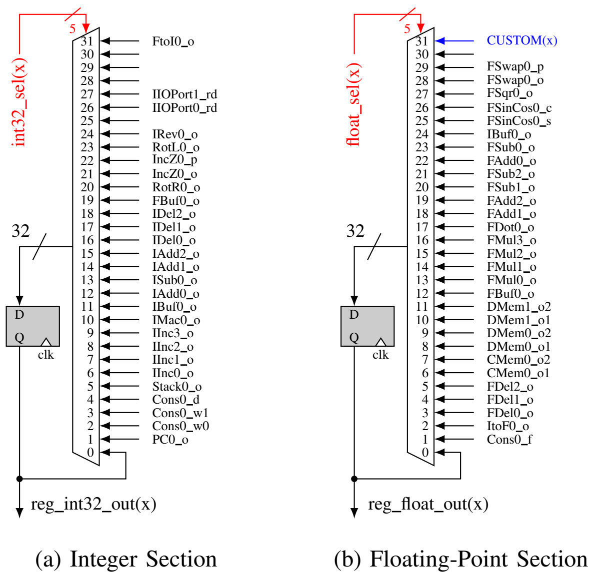
Fig. 3: Software-based Architecture Design Flow

VI. Final Architecture

The results of this optimisation process are presented in this section. In the final architecture, registers and functional units are 32 bits wide and divided into integer as well as floating-point sections to limit multiplexer sizes. The integer registers are used for memory address generation and program flow (e.g. branching and loop control), while the floating-point registers are used for data processing. Fig. 4 depicts the switching matrix and register architectures of the integer and floating-point sections. In both cases, the register output is fed back into the first multiplexer input. When the select signal is zero, the register is thus assigned to itself and remains unchanged.

The input ports on the right of both Fig. 4a and Fig. 4b are the functional unit outputs. In this implementation, there are 32 multiplexer inputs (31 functional unit outputs) making the select signal for each multiplexer $log_2(N) = 5$ bits wide.

The program memory width is determined by the number of registers and the multiplexer select width, and can become rather large. For the 92 registers (82 actual, 10 expandable) used in this implementation, 460 bits are required for the multiplexer select signals. Additionally a 32 bit constant, a 4 bit condition code and 12 flags form the 512 bit wide program word as shown in Fig. 5.



(a) Integer Section

(b) Floating-Point Section

Fig. 4: Switching Matrix and Register Architecture

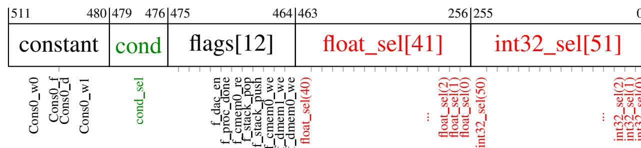


Fig. 5: Program Word Format

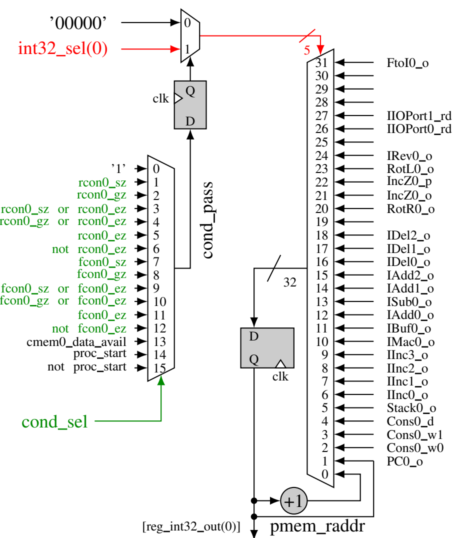


Fig. 6: Program Counter Architecture

The constant from the program word is also routed to the multiplexer inputs, making it possible to assign a fixed value to any register. On the integer section, the constant can be split into two sign extended 16 bit constants or kept in its 32 bit form.

The first register, the program counter, deviates slightly from Fig. 4. It uses the conditional code from the program word (`cond_sel`) to determine whether the program counter is allowed to change. If the condition check fails, or if no condition is selected, the multiplexer selects input port 0. Unlike the other registers however, input port 0 is not directly connected to the register output, but instead increased by one. Thus, if the condition passes the new value is assigned, else the program counter is increased and the program execution continues normally. Fig. 6 depicts the program counter architecture.

Note that the program counter can still be assigned to itself, thus repeatedly executing the current instruction for looping purposes. The loop is terminated and instruction flow continues normally once the conditional code check fails. The condition code can be any external flag (e.g. Ethernet packet received, synchronisation signals) or the result of a comparison operation via the integer- (`rcon`) or floating-point comparators (`fcon`).

The final architecture is shown in Fig. 7 and Fig. 8 for both the integer and floating-point sections. Multiple functional units can share the same physical input register (e.g. `IAdd0_a` and `ISub0_a` refer to the same register) to reduce silicon or FPGA resource utilisation when these functional units are not used in parallel.

The majority of functional units on the integer section perform a standard arithmetic operation such as increment, add, subtract, multiply-add, or a bitwise operation. The add and increment functional units are instantiated multiple times to handle the simultaneous generation of read addresses, write addresses, offsets, write inhibit signals and the various loop counters.

The memory write inhibit signal is generated by a counter and a comparator. The inhibit signal allows the write enable signal to be asserted a fixed number of clock cycles after the inner loop instruction commences, aligning to the latency through the various functional units in the software pipeline. A loop prologue is thus avoided, greatly simplifying the control flow.

Other interesting functional units include a stack, increment-compare-and-zero (`IncZ`), variable delay, I/O ports, integer to floating-point conversion and debugging registers.

The stack unit is a simple clocked last-in-first-out (LIFO) buffer, with a calling hierarchy depth of 16. It only has a single output, `Stack0_o`, which represents the current value on top of the stack. When the "pop" flag from the program word is asserted, the current value on the top of the stack is "popped" off, and the next value appears on the output (or zero if the stack is empty). The "push" flag increases the current program counter value and adds it to the top of the stack. A function call thus requires the function address to be assigned to the PC while asserting the "push" flag. Returning from the function is achieved by assigning the stack output to the PC and asserting the "pop" flag.

The increment-compare-and-zero (`IncZ`) functional unit is one that is surprisingly not featured on modern instruction sets. It is however a very useful functional unit in the address generation and control-flow paradigm. Under normal operating conditions the output `IncZ0_o` is assigned to input `IncZ0_a`, which forms a continuous counter that resets to zero when the value `IncZ0_b` is reached. When the output `IncZ0_p` is assigned to the `IncZ0_c` input, an up-counter

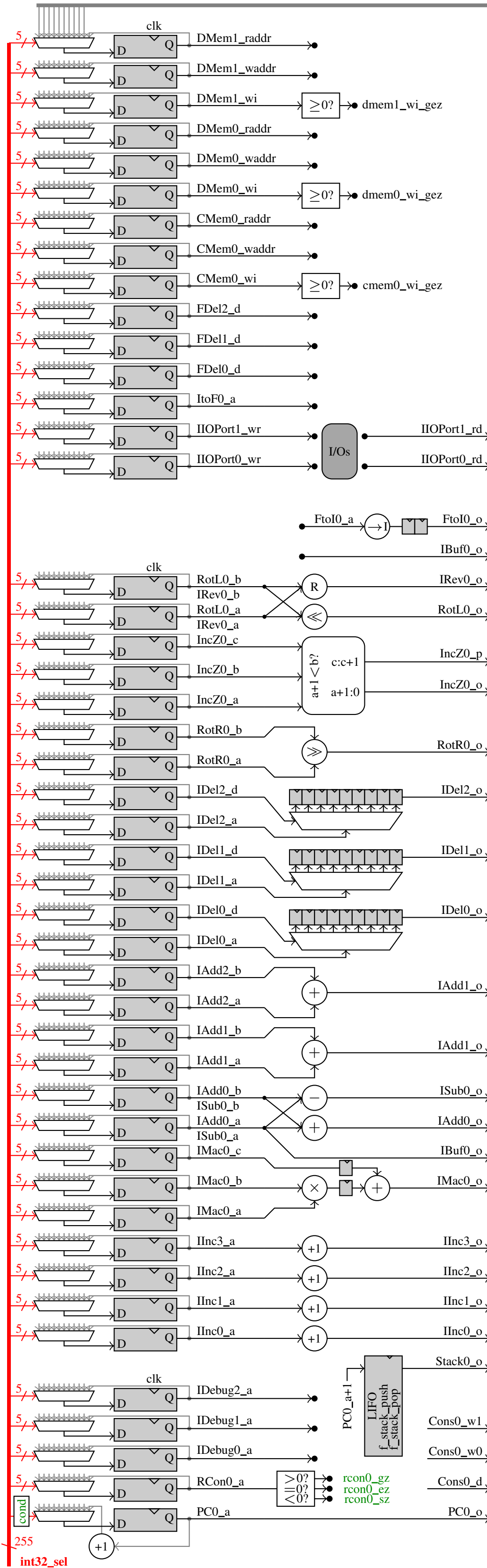


Fig. 7: Processor Architecture: Integer Section

counting the number of overflows on the `IncZ0_a` side is achieved. This instruction can thus be used to transpose matrices of arbitrary dimension, for FFT address calculation purposes, as a circular address buffer, or simply as a counter and comparator.

The integer buffer unit is used for temporary variable storage (e.g. parameters in function call) or delaying a result by a single clock cycle for alignment purposes. When more clock cycles of delay are required, the variable delay operation provides a tapped delay register, capable of selecting between 1 and 32 clock cycles of latency. This operation is needed for synchronisation and alignment purposes when the processing latency needs to be matched to the address generation latency or vice versa.

An I/O port interface also resides on the integer section, and similar to all other functional units, can be assigned and read every clock cycle. The `IIOPort` functional unit can thus provide a full duplex high bandwidth interface to peripherals, coprocessors or general purpose I/O pins.

The integer and floating-point debug registers are routed to a logic analyser (such as integrated Xilinx ChipScope ILA or an external logic analyser port) to provide a clock-by-clock snapshot of the internal debug register values. These snapshots can be loaded into the development environment for exact comparisons between runtime and simulated results.

On the floating-point section in Fig. 8, functional units have longer critical paths and thus higher latency (e.g. 2 clock cycles for multiply or add) than those of the integer section (0 or 1 clock cycle). Multipliers, adders and subtracters are instantiated numerous times to cater for the concurrent arithmetic requirements of the different applications. All functional units operate on real numbers, as the software pipelining mechanism can join these primitives into complex operations with the same latency and throughput as a dedicated circuit.

The buffer (`FBuf`), delay (`FDel`), comparator (`FCon`), and debug (`FDebug`) functional units are similar to their counterparts of the integer section. Sine, cosine and a square root functional units are used in various algorithms and thus also added into the datapath. Conversion functions (`ItoF` and `FtoI`) as well as direct pass-through registers between integer and floating-point sections are also provided.

Since most algorithms exhibit alternating horizontal and vertical data dependencies, the processing chain typically involves reading a complex-valued data stream from memory, performing some mathematical operations, and writing back the processed data. Two 64-bit data memories are thus mapped directly into the datapath as functional units along with a coefficient memory unit. Memory architectures such as external QDR memory, SRAM or internal FPGA Block RAM exhibit deterministic latency and are thus well suited for this purpose. DDR or NAND memory devices pose a problem when used as data memory in the processing loop, as the row activation times vary and additional operations such as refreshing or block erasing need to be performed.

Other interesting functional units on the floating-point section include the `FSwap` and the `FDot` operations. The `FSwap` functional unit takes two input values, sorts them and outputs the larger value to the `_p` port and the smaller value to

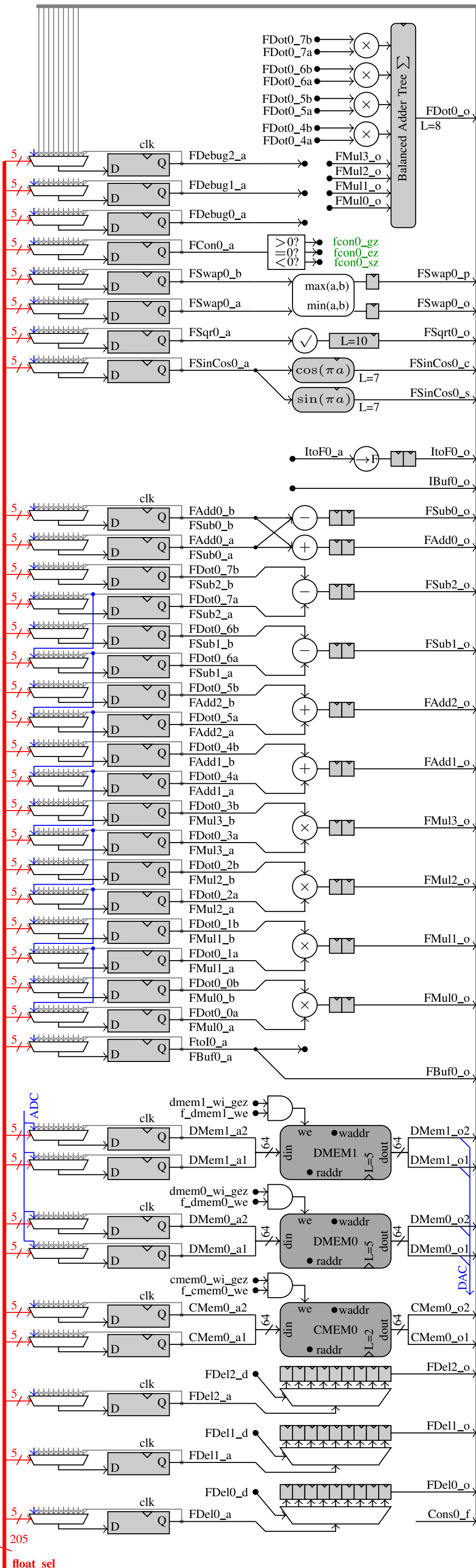


Fig. 8: Processor Architecture: Floating-Point Section

the _o port. This operation is extremely useful for algorithms requiring data comparisons such as sorting networks, maximum value identification, and various CFAR algorithms. The floating-point dot product (`FDot`) has 16 inputs and a throughput of 1 result every clock cycle. Internally the outputs of all 8 multipliers are connected to a balanced adder tree consisting of 7 adders (3 levels deep). The `FDot` functional unit can thus be used for FIR filters, matrix multiplications, correlations, convolutions, windowing and any other sum-of-products operations.

The independent select signals for each multiplexer provide direct control over horizontal as well as vertical instruction-level parallelism in both the data- and control-path of the proposed architecture. This features some resemblance to very large instruction word (VLIW) and transport triggered architectures (TTA) [19]–[21]. A simplification of the TTA architecture is the synchronous transfer architecture (STA) [22], [23], which removes the register file, trigger-ports and queues from the critical path of the TTA architecture, using synchronous communication between modules (somewhat resembling [24]). The assembly instruction thus contains transfer, opcode and explicit trigger signals for each functional module.

The proposed architecture could thus be seen as a further simplification of the STA architecture, in which the instruction word is only used to specify the transfer routing for each register, and not for functional unit control or triggering. The proposed architecture thus allows even finer grained control and can use every computational resource simultaneously, rather than using multiple functional modules simultaneously. Additionally, the proposed architecture completely removes the register file, and provides various architectural optimisations for loop control and streaming applications. This makes the architecture ideal for creating deep software pipelines for a variety of applications and algorithms.

VII. Complex Modulus Example

The complex modulus (magnitude) operation is well suited for the illustration of this software pipelining mechanism. The signal flow graph of the magnitude operation is shown below in Fig. 9.

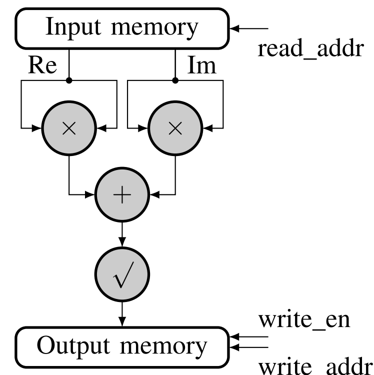


Fig. 9: Signal Flow Graph for the Magnitude Operation

The signal flow graph is almost directly translated to the dataflow routing on the proposed architecture. Each line in the

FLOW language determines a connection between a functional unit output and a functional unit input as shown in the listing below:

```
FMul0_a = DMem0_o    ; RE*RE
FMul0_b = DMem0_o
FMul1_a = DMem0_p    ; IM*IM
FMul1_b = DMem0_p
FAdd0_a = FMul0_o
FAdd0_b = FMul1_o    ; RE*RE + IM*IM
FSqr0_a = FAdd0_o    ; SQRT(RE*RE + IM*IM)
DMem0_a = FSqr0_o
||   ; next instruction delimiter
```

The *FLOW* language maps the multiplexer routing on the proposed architecture to a human readable representation, to some extent resembling the assembly language of traditional instruction set architectures. Each assignment line in the source code thus simply determines the constant on the select signal of the related multiplexer. Every move operation in this listing occurs in parallel, making the process a software pipeline capable of producing a new output every clock cycle.

The integer section is responsible for updating the memory read address to supply a constant stream of input values. Similarly, the loop counter, write address and write enable signals need to be updated accordingly. Using the write inhibit counter, the write enable signal is asserted when the first output is available from the square root operation after the processing and memory fetch latency.

Provided that there are sufficient functional units to do the required operation in a software pipelined stream, only a single iteration over the data values is required. For more complex calculations, the processing chain can be split into stages, writing the temporary results to memory before reading them in the next stage.

VIII. Results

The final architecture was implemented on a Xilinx Virtex 5 SX95T FPGA for verification purposes. Fig. 10 shows the top level of the firmware instantiating the soft-processor core.

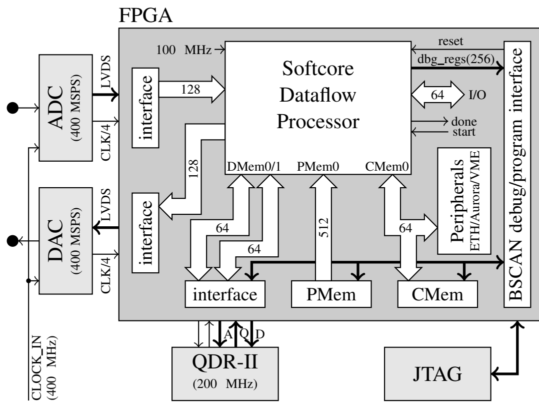


Fig. 10: Hardware and Firmware Interface: Top Level

A conservative clock frequency of 100 MHz was initially chosen as a proof of concept to avoid timing closure problems, but later synthesis results revealed that substantially higher clock frequencies are achievable. A full custom ASIC implementation of the proposed architecture is expected to achieve clock frequencies in the GHz range, matching or even exceeding those of commercial DSP and CPU architectures [25], [26].

The cycle count equations for the FIR and FFT operations are shown in Eq. 1 and Eq. 2 respectively, where N is the number of samples, L is the filter length and P is the number of FFTs to perform consecutively in memory. The cycle count for any operation that is implementable as a stream with the available functional units, is simply N with a few extra clock cycles for memory read and functional unit latency.

$$\mathrm{FIR_{cc}} = 5 + \mathrm{ceil}(\frac{L}{8})\left(N + 27 - 4\mathrm{ceil}(\frac{L}{8})\right) \quad (1)$$

$$\mathrm{FFT_{cc}} = \left(\frac{PN}{2} + 21\right)\log_2(N) + 8 \quad (2)$$

The FFT clock cycle results of the proposed architecture are compared against the Texas Instruments' C66x architecture and the Xilinx CORE Generator FFT IP core for point sizes ranging between 8 and 16384 in Fig. 11. Note that these results were obtained by averaging the clock cycle results of over 40 repeated runs, and using the optimised floating-point FFT operation from the Texas Instruments DSPLIB. The proposed architecture and the FFT core required the identical number of clock cycles for each repeated run (Eq. 2 for the proposed architecture), while the C66x clock cycle count achieved a relative standard deviation of less than 6 percent over the entire range.

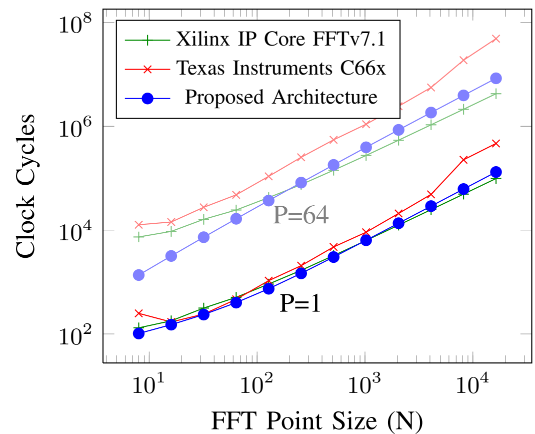


Fig. 11: FFT clock cycle comparison N=8 to 16384

The clock cycle performance of the C66x and FFT core is similar to the proposed architecture for point sizes smaller than 4096 and single FFT operations. As the point sizes and the number of FFT operations increase, the C66x cache performance is no longer optimal and the performance gap between the architectures increases. For large sizes and multiple consecutive FFTs, the pipelined streaming architecture of the FFT IP core outperforms both sequential processors.

To evaluate the radar performance results, the entire radar signal processing chain was implemented in the *FLOW* language, and run on the hardware platform in Fig. 10. Similarly,

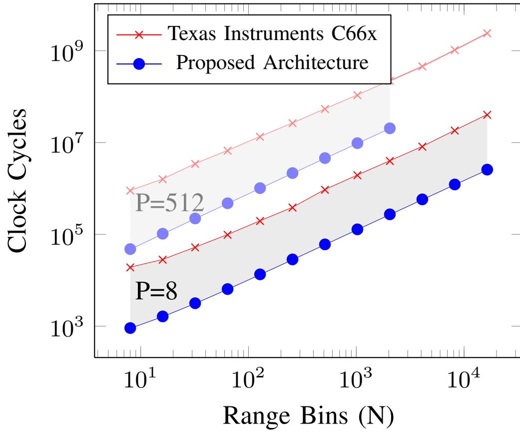


Fig. 12: RSP clock cycle comparison N=8 to 16384

the identical chain was implemented on the Texas Instruments C66x architecture using only library calls to the DSPLIB (except for the complex modulus operation, which was not available in the library). The radar clock cycle performance comparison chart is shown in Fig. 12.

Note that the last 3 data points of the proposed architecture were not computable due to limited external memory on the development board. There is an almost constant offset between the C66x results and the proposed architecture results on the log scale (shaded regions), a difference of more than an order in magnitude. The C66x implementation requires between 10.8 and 20.9 times (average factor of 13.9) the number of clock cycles compared to the equivalent implementation on the proposed architecture for typical radar operating parameters (N=8 to 16384, P=8 to 512).

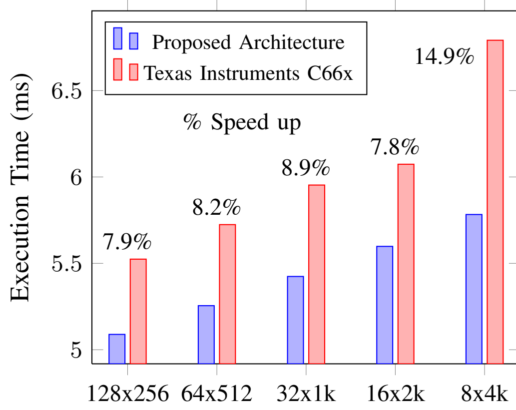


Fig. 13: RSP execution time comparison

A performance comparison based on execution time is biased towards the DSP architecture, which features a clock speed of 1200 MHz compared to the proposed architecture which runs between 100 and 160 MHz on the selected FPGA. The performance of the two architectures is compared in Fig. 13 for a few arbitrary selected dimensions. Note how the performance of the C66x architecture decreases when the point sizes becomes extremely small. Similarly, the performance declines as the point sizes become excessively large and no longer fit into cache memory. When comparing the total execution time performance across the entire range of typical operating parameters (N=8 to 16384, P=8 to 512), the proposed architecture outperforms the C66x DSP architecture by an average of 15.8 percent, even with the limited clock frequency of 100 MHz.

IX. Conclusion

The main focus of this research was to design a processing architecture that is optimised for radar signal processing applications. Constructs of both sequential processors and dataflow machines were merged into a tightly coupled solution, capable of fully exploiting each of the underlying processing resources concurrently. This novel soft-core processing architecture features an excellent match to the core computational requirements of the RSP. The software-based development environment enables quick algorithmic changes and instant compile times during field tests, greatly improving the ease of use compared to the complex FPGA design flow. The proposed architecture outperforms a high-end commercial DSP architecture in both number of clock cycles and processing time, despite containing fewer arithmetic resources and being limited by the restricted clock frequencies achievable in the FPGA technology. Table I summarises the characteristics of this processing architecture.

TABLE I: Architectural characteristics summary

Number of Cores	1 (can be increased)
Clock Frequency	100 MHz (ASIC estimated: 1.0 - 2.7 GHz)
Streaming Performance	High (data independent processing)
Burst Processing Perf.	Medium to High (software pipelines)
General Purpose Perf.	Low to Medium (no hardware scheduling)
Interrupt Support	Limited (impractical context backup)
Latency	Low (deterministic and real-time)
Interface Capabilities	Excellent (direct streaming interfaces to peripherals and external systems)
Architectural Efficiency	Excellent (extremely low overhead, high ALU utilisation)
Performance Scalability	Excellent (add/remove functional units)
Power Consumption	Low to Average (minimal control overheads)
Code Compatibility	None (no backwards compatibility)
Code Density	High (horizontal and vertical)
Resource Usage	Average (medium to high on an FPGA)
Ease of Use	Good (software based: function calls to optimised DSP routines, no compiler yet)

The proposed architecture is well suited for applications requiring a programmable front-end or streaming processor with a high computational throughput and a low power consumption. Although the Pulse-Doppler radar processor was the main focus of this investigation, the architecture is equally applicable to other radar classes (e.g. SAR, STAP) as well as sonar processors. A custom ASIC implementation of the proposed architecture would thus be well suited for integration

into the transmit/receive (TR-) modules of active electronically scanned array (AESA) and multiple-input multiple-output (MIMO) radar systems, enabling instant front-end processing mode changes for various operational requirements (e.g. communications, radar, electronic warfare techniques and jamming modes).

References

[1] S. Lal, R. Muscedere, and S. Chowdhury, "An FPGA-Based Signal Processing System for a 77 GHz MEMS Tri-Mode Automotive Radar," in *IEEE Int. Symp. Rapid System Prototyping*, May 2011, pp. 2–8.

[2] R. Stapleton, K. Merranko, C. Parris, and J. Alter, "The Use of Field Programmable Gate Arrays in High Performance Radar Signal Processing Applications," in *IEEE Int. Radar Conf.*, 2000, pp. 850–855.

[3] H. Nicolaisen, T. Holmboe, K. Hoel, and S. Kristoffersen, "High Resolution Range-Doppler Radar Demonstrator Based on a Commercially Available FPGA Card," in *Int. Conf. Radar*, Sept. 2008, pp. 676–681.

[4] Z. Ali, A. Arshad, and U. Razzaq, "An FPGA based semi-parallel architecture for higher order Moving Target Indication (MTI) processing," in *IEEE Int. Symp. Rapid System Prototyping (RSP)*, June 2010, pp. 1–7.

[5] C. Neri, G. Baccarelli, S. Bertazzoni, F. Pollastrone, and M. Salmeri, "Parallel hardware implementation of RADAR electronics equipment for a LASER inspection system," *IEEE Trans. Nuclear Science*, vol. 52, no. 6, pp. 2741–2748, Dec. 2005.

[6] M. Pfitzner, F. Cholewa, P. Pirsch, and H. Blume, "FPGA based architecture for real-time SAR processing with integrated motion compensation," in *Asia-Pacific Conf. Synthetic Aperture Radar (APSAR)*, Sept. 2013, pp. 521–524.

[7] J. Greco, G. Cieslewski, A. Jacobs, I. Troxel, and A. George, "Hardware/software Interface for High-performance Space Computing with FPGA Coprocessors," in *IEEE Aerospace Conf.*, Jan. 2006, pp. 1–10.

[8] Y. He, C. Le, J. Zheng, K. Nguyen, and D. Bekker, "ISAAC - A Case of Highly-Reusable, Highly-Capable Computing and Control Platform for Radar Applications," in *IEEE Radar Conf.*, May 2009, pp. 1–4.

[9] J. McAllister, R. Woods, S. Fischaber, and E. Malins, "Rapid implementation and optimisation of DSP systems on FPGA-centric heterogeneous platforms," *J. Syst. Architect.*, vol. 53, no. 8, pp. 511–523, 2007.

[10] O. Cret, K. Pusztai, C. Vancea, and B. Szente, "CREC: A Novel Reconfigurable Computing Design Methodology," in *Int. Proc. Parallel Distr. Processing*, Apr. 2003, pp. 8–16.

[11] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-Level Synthesis for FPGAs: From Prototyping to Deployment," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 30, no. 4, pp. 473–491, Apr. 2011.

[12] P. Yiannacouras, J. Steffan, and J. Rose, "Portable, Flexible, and Scalable Soft Vector Processors," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 20, no. 8, pp. 1429–1442, Aug. 2008.

[13] J. Yu, C. Eagleston, C. H.-Y. Chou, M. Perreault, and G. Lemieux, "Vector Processing as a Soft Processor Accelerator," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 2, no. 2, pp. 12:1–12:34, June 2009.

[14] P. Wang, J. McAllister, and Y. Wu, "Soft-core Stream Processing on FPGA: An FFT case study," in *IEEE Int. Conf. Acoustics, Speech and Signal Processing (ICASSP)*, May 2013, pp. 2756–2760.

[15] R. Broich and H. Grobler, "Analysis of the Computational Requirements of a Pulse-Doppler Radar Signal Processor," in *IEEE Radar Conf.*, May 2012, pp. 835–840.

[16] R. Broich, "A Soft-Core Processor Architecture Optimised for Radar Signal Processing Applications," Master's thesis, University of Pretoria, Dec. 2013.

[17] A. Hoffmann, T. Kogel, A. Nohl, G. Braun, O. Schliebusch, O. Wahlen, A. Wieferink, and H. Meyr, "A Novel Methodology for the Design of Application-Specific Instruction-Set Processors (ASIPs) Using a Machine Description Language," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 20, no. 11, pp. 1338–1354, 2001.

[18] V. Kathail, S. Aditya, R. Schreiber, B. Rau, D. Cronquist, and M. Sivaraman, "PICO: Automatically Designing Custom Computers," *Computer*, vol. 35, no. 9, pp. 39–47, 2002.

[19] H. Corporaal, "A different approach to high performance computing," in *Int. Conf. High-Performance Computing*, Dec. 1997, pp. 22–27.

[20] *MAXQ Family Users Guide*, 6th ed., Maxim Integrated, Sept. 2008.

[21] Y. He, D. She, B. Mesman, and H. Corporaal, "MOVE-Pro: a Low Power and High Code Density TTA Architecture," in *Int. Conf. Embedded Computer Systems*, July 2011, pp. 294–301.

[22] G. Cichon, P. Robelly, H. Seidel, T. Limberg, and G. Fettweis, "SAMIRA: A SIMD-DSP architecture targeted to the Matlab source language," in *Proc. Global Signal Processing Expo & Conf.*, July 2004.

[23] G. Cichon, P. Robelly, H. Seidel, M. Bronzel, and G. Fettweis, "Synchronous Transfer Architecture (STA)," in *Proc. Int. Workshop on Systems, Architectures, Modeling & Simulation (SAMOS04)*, July 2004, pp. 126–130.

[24] D. Cronquist, C. Fisher, M. Figueroa, P. Franklin, and C. Ebeling, "Architecture Design of Reconfigurable Pipelined Datapaths," in *Conf. Advanced Research in VLSI*, Mar. 1999, pp. 23–40.

[25] L. Noury, S. Dupuis, and N. Fel, "A Reference Low-Complexity Structured ASIC," in *IEEE Int. Sym. on Circuits & Systems*, May 2012, pp. 2709–2712.

[26] I. Kuon and J. Rose, "Measuring the Gap Between FPGAs and ASICs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 26, no. 2, pp. 203–215, Feb. 2007.

René Broich received his BEng, BEng(Hons) and MEng degrees in electronic engineering from the University of Pretoria, South Africa, in 2010, 2011 and 2014 respectively, all with distinction. He is currently working as a digital design engineer in the Radar and Electronic Warfare department at the Council for Scientific and Industrial Research (CSIR), South Africa. His research interests include application specific architectures, high performance computing, real-time embedded systems, as well as field-programmable gate array, digital signal processor and microprocessor architectures.

Hans Grobler received the BE degree in electronic engineering from Stellenbosch University, South Africa, in 1995. In 2001 he received the ME degree in electronic engineering whilst working as digital systems engineer and software team leader on the first miniaturized satellite (SUNSAT) designed and manufactured in South Africa. He completed his BSc(Hons) and MSc degrees in computer science at the University of Pretoria, South Africa, in 2005 and 2006 respectively. He is currently senior lecturer in the Department of Electrical, Electronic and Computer Engineering at the University of Pretoria. His research interests include real-time embedded systems, hardware and software parallel processing, robotics, computer vision and artificial intelligence. He is a member of the IEEE and the ACM.