High Performance Stream Processing on FPGA

John McAllister

Abstract Field Programmable Gate Array (FPGA) have plentiful computational, communication and member bandwidth resources which may be combined into high-performance, low-cost accelerators for computationally demanding operations. However, deriving efficient accelerators currently requires manual register transfer level design—a highly time-consuming and unproductive process. Software-programmable processors are a promising way to alleviate this design burden but are unable to support performance and cost comparable to hand-crafted custom circuits. A novel type of processor is described which overcomes this shortcoming for streaming operations. It employs a fine-grained processor with very high levels of customisability and advanced program control and memory addressing capabilities in very large-scale custom multicore networks to enable accelerators whose performance and cost match those of hand-crafted custom circuits and well beyond comparable soft processors.

1 Introduction

Field Programmable Gate Array (FPGA) technologies have long been recognised for their ability to enable very high-performance realisations of computationally demanding, highly parallel operations beyond the capability of other embedded processing technologies. Recent generations of FPGA have seen a rapid increase in this computational capacity and the emergence of System-on-Chip SoC-FPGA, incorporating heterogeneous multicore processors alongside FPGA programmable fabric. A key motivation for these hybrid architectures is the ability of FPGA to host performance-critical operations, offloaded from processors, as application-specific *accelerators* with any combination of high-performance, low cost or high energy efficiency.

J. McAllister (✉)
Institute of Electronics, Communications and Information Technology (ECIT), Queen's University Belfast, Belfast, UK
e-mail: jp.mcallister@ieee.org

S. S. Bhattacharyya et al. (eds.), *Handbook of Signal Processing Systems*,
https://doi.org/10.1007/978-3-319-91734-4_13

The resources available with which accelerators may be built are enormous: the designer has, every second, access to trillions of multiply accumulate operations via on-chip DSP units [3, 30] and memory locations in Block RAM (BRAM) [3, 31], alongside the computationally powerful and highly flexible Look-Up Table (LUT) FPGA programmable logic [17]. For instance, the Virtex$^{\text{®}}$-7 family of Xilinx FPGAs offers up to 7×10^{12} multiply-accumulate (MAC) operations per second and 40×10^{12} bits/s memory access rates.

To combine these resources into accelerators of highest performance or lowest cost, though, requires manual design of custom circuit architectures at Register Transfer Level (RTL) in a hardware design language. This is a low level of design abstraction which imposes a heavy design burden, significantly more complicated than describing behaviour in a software programming language. Hence, for many years designers have sought a way to realise accelerators more rapidly without suffering critical performance or cost bottlenecks. Software-programmable 'soft' processors are one way to do so, but at present adopting such an approach demands substantial compromise on performance and cost. Soft processors allow their architecture to be tuned before synthesis to improve the performance and cost of the final result. Soft general-purpose processors such as MicroBlaze [32] and Nios-II [2] are performance-limited and a series of approaches attempt to resolve this issue. One approach uses soft vector coprocessors [9, 24, 33, 34] employing either assembly-level [34] or mixed C-macro and inline assembly programming. These enable performance increases by orders of magnitude beyond Nios-II and MIPS [34], but performance and cost still lag custom circuits. An alternative approach is to redesign the architecture of the central processor architecture for performance/cost benefit, and approach adopted in the iDEA [8] processor. Multicore architectures incorporating up to 16 [12, 22, 25] or even 100 processors in [12] have also been proposed.

However, the cost of enabling software programmability in all of these approaches is a reduction in performance or efficiency in the resulting accelerators, relative to custom circuit solutions. The result is that the performance of these architectures is only marginally beyond that of software-programmable devices and there is no evidence these are competitive with custom circuits. It appears that if FPGA soft processors are to be a viable alternative to custom accelerators then performance and cost must improve radically.

2 The FPGA-Based Processing Element (FPE)

A unique, lean soft processor—the *FPGA Processing Element* (FPE)—is proposed to resolve this deficiency. The architecture of the FPE is shown in Fig. 1. It contains only the minimum set of resources required for programmability: the instructions pointed to by the *Program Counter* (PC) are loaded from *Program Memory* (PM) and decoded by the *Instruction Decoder* (ID). Data operands are read either from *Register File* (RF), or in the case of immediate data *Immediate Memory* (IMM) and

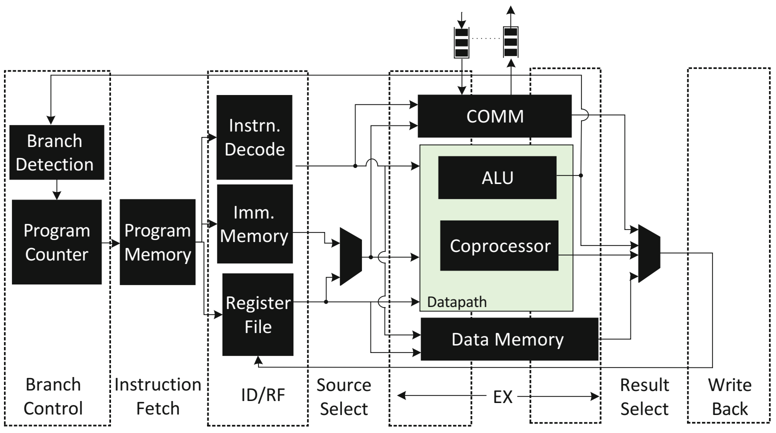

Fig. 1 The FPGA processing element

Table 1 FPE parameters and instructions

(a) FPE configuration parameters

Parameter	Meaning	Values
DataWidth	Data wordsize	16/32 bits
DataType	Type of data	Real/complex
ALUWidth	No. DSP48e slices	1–4
PMDepth	PM Capacity	Unlimited
PMWidth	PM Wordsize	Unlimited
DMDepth	DM/RF Capacity	Unlimited
RFDepth	No. RF locations	Unlimited
TxCOMM	No. Tx ports	≤1024
RxCOMM	No. Rx ports	≤1024
IMMDepth	IMM Capacity	Unlimited

(b) FPE instruction set

Instruction	Function
LOOP	Loop
BEQ/BGT/BLT	Branching
GET/PUT	FIFO get/put
NOP	No operation
MUL/**ADD**/**SUB**	Multiply/add/subtract
MULADD(FWD)	Multiply-add
MULSUB(FWD)	Multiply-subtract
COPROC	Coprocessor access
LD/**ST**	Load/store
LDIMM/STIMM	IMM load/store

processed by the ALU (implemented using a Xilinx DSP48e). In addition, a *Data Memory* (DM) is used for bulk data storage and a *Communication Adapter* (COMM) performs on/off-FPE communications.

The FPE is soft and hence *configurable* to allow its architecture to be customised pre-synthesis in terms of the aspects listed in Table 1(a). Beyond these, custom coprocessors can also be integrated alongside the ALU to accelerate specific custom instructions. Of course, the FPE is also *programmable*, with an instruction set described in Table 1(b).

When implemented on Xilinx Virtex 5 VLX110T FPGA, a 16 bit Real FPE costs 90 LUTs, 1 DSP48e and enables 483×10^6 multiply-add operations per second. This represents around 18% of the resource of a conventional MicroBlaze processor, whilst increasing performance by a factor 2.8.

The FPE's low cost allows it to be combined in very large numbers on a single FPGA, to realise operations via multicore architectures, with communication between FPEs via point-to-point queues. Hence the FPE may be viewed as a

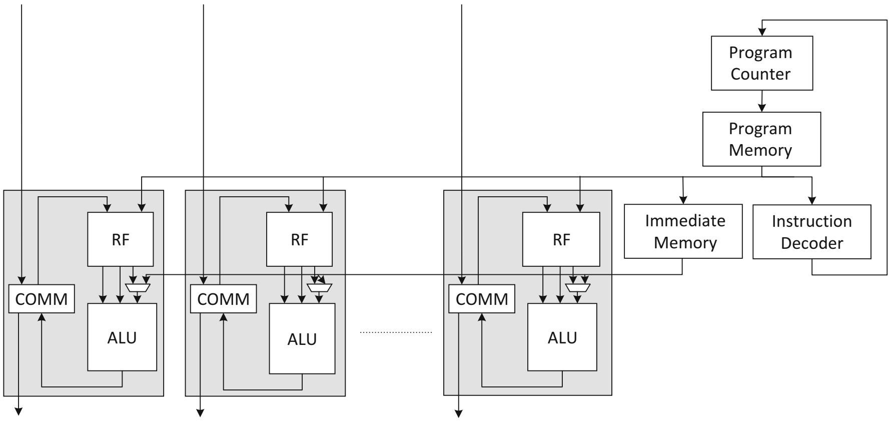

Fig. 2 SIMD processor architecture

fundamental building block for realising computationally demanding operations on FPGA.

To do so efficiently, the FPE should be able to exploit all the different types of parallelism in a program or application. Task parallelism is exploited in the multicore architectures proposed, but using these to realise data parallel operation is less than efficient, due to the duplication of control logic and data and memory resources. In this case each FPE will contain the same instructions in their PM, access RF in the same orders and execute the same programs. There is considerable overhead incurred when control resource is duplicated for each FPE. To avoid this occurring, the FPE is further extended into a configurable SIMD processor component, as illustrated in Fig. 2.

The width of the SIMD is configurable via a new parameter, `SIMDways`, which dictates the number of datapath lanes. All of the FPE instructions (except BEQ, BGT and BLT) can be used as SIMD instructions.

3 Case Study: Sphere Decoding for MIMO Communications

To illustrate the use of FPE-based multicores for FPGA accelerators, a case study—Sphere Decoding (SD) for Multiple-Input, Multiple-Output (MIMO) communications systems—is used. MIMO systems employ multiple transmit and multiple receive channels [26] to enable data rates of unprecedented capacity, prompting their adoption in standards such as 802.11n [14]. An M-element array of transmit antennas emit a vector $\mathbf{s} \in \mathbb{C}^M$ of QAM-modulated symbols. The vector of symbols $\mathbf{y} \in \mathbb{C}^N$ received at an N-element array of antennas is related to $\mathbf{s}$ by:

$$\mathbf{y} = \mathbf{Hs} + \mathbf{v}, \tag{1}$$

where $\mathbf{H} \in \mathbb{C}^{N \times M}$ represents the MIMO channel, used typically as a parallel set of flat-fading subchannels via Orthogonal Frequency Division Multiplexing (OFDM)

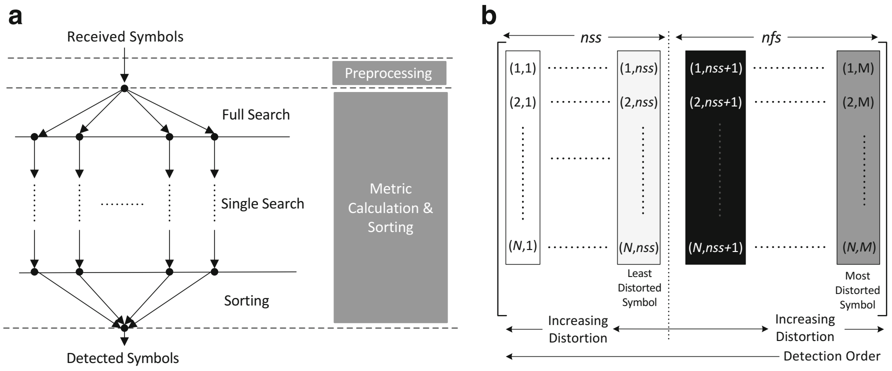

Fig. 3 FSD algorithm components. (**a**) FSD Tree Structure. (**b**) General Form of $\mathbf{H}^{\dagger}$

(108 in the case of 802.11n) and $\mathbf{v} \in \mathbb{C}^N$ additive noise. Sphere Decoding (SD) is used to derive an estimate $\hat{\mathbf{s}}$ of $\mathbf{s}$. It offers near that of the ideal ML detector, with significantly reduced complexity [20, 23]. The Fixed-Complexity SD (FSD) has a particularly low complexity, two-stage deterministic process which makes it ideal for efficient realisation via an FPGA accelerator [5]. FSD realises a two-stage detection process illustrated in Fig. 3a.

Algorithm 1 SQRD for FSD

```
    input  : $\mathbf{H}, M$
    output: $\mathbf{Q}, \mathbf{R}$, order
 1  Phase 1: Initialization
 2  $\mathbf{Q} = \mathbf{H}, \mathbf{R} = \mathbf{0}_M$,
 3  order $= [1, \cdots, M]$,
    $nfs = \lceil \sqrt{M} - 1 \rceil$
 4  for $i \leftarrow 1$ to $M$ do
 5      $\mathbf{norm}_i = \|\mathbf{q}_i\|^2$
 6  end
 7  Phase 2: SQRD ordering
 8  for $i \leftarrow 1$ to $M$ do
 9      $k = min(nfs + 1, M - i + 1)$
10      $k_i = \underset{j=i,\cdots,M}{\overset{k}{\arg\min}}\, \mathbf{norm}_j$
11      Exchange columns $i$ and $k_i$ in $\mathbf{R}$, order, norm and $\mathbf{Q}$
12      $r_{i,i} = \sqrt{\mathbf{norm}_i}$
13      $\mathbf{q}_i = \mathbf{q}_i / r_{i,i}$
14      for $l \leftarrow i + 1$ to $M$ do
15          $r_{i,l} = \mathbf{q}_i^H \cdot \mathbf{q}_l$
16          $\mathbf{q}_l = \mathbf{q}_l - r_{i,l} \cdot \mathbf{q}_i$
17          $\mathbf{norm}_l = \mathbf{norm}_l - r_{i,l}^2$
18      end
19  end
```

Pre-Processing (PP) orders the symbols of $\mathbf{y}$ according to the perceived distortion experienced by each. This is achieved by reordering the columns of $\mathbf{H}$ to give $\mathbf{H}^{\dagger}$ (the general form of which is illustrated in Fig. 3b). Practically, this is achieved via an iterative Sorted QR Decomposition (SQRD) algorithm, described in Algorithm 1 [11].

SQRD-based PP ordering for FSD transforms the input channel matrix $\mathbf{H}$ to the product of a unitary matrix $\mathbf{Q}$ and an upper-triangular $\mathbf{R}$ via QR decomposition, whilst deriving **order**, the order of detection of the received symbols during MCS. It operates in two phases, as described in Algorithm 1. In Phase 1 $\mathbf{Q}$, $\mathbf{R}$, **order**, **norm** and nfs are initialized as shown in lines 2–5 of Algorithm 1, where $\mathbf{q}_i$ is the ith column of $\mathbf{Q}$. Phase 2 comprises M iterations, in each of which the kth lowest entry in **norm** is identified (lines 9 and 10) before the corresponding column of $\mathbf{R}$ and elements in **order** and **norm** are permuted with the ith (line 11) and orthogonalized (line 12–18). The resulting $\mathbf{Q}$, $\mathbf{R}$, and **order** are used for Metric Calculation and Sorting (MCS) as defined in (3) and (4).

Metric Calculation and Sorting uses an M-level decode tree to perform a Euclidean distance based statistical estimation of $\mathbf{s}$. Groups of M symbols undergo detection via a tree-search structure illustrated in Fig. 3a.

The number of nodes at each tree level is given by $\mathbf{n}_S = (n_1, n_2, \ldots, n_M)^T$. The first *nfs* levels process the symbols from the worst distorted paths by *Full Search* (FS) enumeration of all elements of the search space. This results in P child nodes at level $i+1$ per node at level i, where P is the number of QAM constellation points. For full diversity, *nfs* is given by

$$nfs = \lceil \sqrt{M} - 1 \rceil. \tag{2}$$

The remaining *nss* ($nss = M - nfs$) levels undergo *Single Search* (SS) where only a single candidate detected symbol is maintained between layers. At each MCS tree level, (3) and (4) are performed.

$$\tilde{s}_i = \hat{s}_{ZF,i} - \sum_{j=i+1}^{M_t} \frac{r_{ij}}{r_{ii}} \left(\hat{s}_{ZF,j} - \hat{s}_j\right) \tag{3}$$

$$d_i = \sum_{j=i}^{M_t} r_{ij}^2 \left\| \hat{s}_{ZF,j} - \hat{s}_j \right\|^2, D_i = d_i + D_{i+1} \tag{4}$$

In (3) and (4), r_{ij} refers to an entry in $\mathbf{R}$, derived by QR decomposition of $\mathbf{H}$ during PP, $\hat{s}_{ZF}$ is the center of the FSD sphere and $\tilde{s}_j$ is the jth detected data, which is sliced to $\hat{s}_j$ in subsequent iterations of the detection process [13]. Since D_{i+1} can be considered as the Accumulated Partial Euclidean Distance (APED) at level $j = i + 1$ of the MCS tree and d_i as the PED in level i, the APED can be obtained by recursively applying (4) from level $i = M$ to $i = 1$. The resulting candidate symbols are sorted based on their Euclidean distance measurements, and the final result produced post-sorting.

This behaviour is duplicated across all OFDM subcarriers, of which there are 108 in 4×4 16-QAM 802.11n MIMO. For real-time processing this behaviour is repeated independently for all 108 subcarriers and must occur within 4 μs and at a rate of 480 Mbps for real-time performance. These are challenging requirements which has seen detection using custom circuit accelerators become a well-studied real-time implementation problem [4, 7, 15, 16, 21, 27]. It is notable that none of these uses software-programmable accelerator components. This section considers the use of the FPE to realise such a solution.

4 FPE-Based Pre-processing Using SQRD

The SQRD preprocessing technique is low-complexity relative to other, ideal preprocessing approaches. It is also numerically stable and lends itself well to fixed-point implementation, hence making it suitable for realisation on FPGA, as a result of its reliance on QRD. However, there are two major issues that must be resolved to enable FPE-based SQRD PP for 4×4 802.11n. It computational complexity remains high as outlined in Table 2; given the capabilities of a single FPE, it appears that a large-scale multi-FPE architecture is required to enable SQRD for 4×4 802.11n. Its reliance on square root and division operations also present a challenge, since these operations are not native to the DSP48e components used as the datapaths for the FPE and will have low performance when realised thereon [19].

To avoid this performance bottleneck, datapath coprocessors are considered to enable real-time division and square-root operations.

4.1 FPE Coprocessors for Arithmetic Acceleration

Non-restoring 16-bit division [19] requires 312 cycles when implemented using only the DSP48e in an *16R* FPE. This equates to approximately 1.2×10^6 div/s (divisions per second). Hence, around 100 FPEs would be required to realise the 120×10^6 divisions required per second (MDiv/s) for 4×4 SQRD for 802.11n. The high resource cost this would entail can be alleviated by adding radix-2 or radix-4 non-restoring division coprocessors [19] alongside the DSP48e in the FPE ALU (Fig. 4).

The performance, cost and efficiency (in terms of throughput per LUT, or TP/LUT) of the programmed FPE when division is realised using a programmed approach and the DSP48e only, (*FPE-P*) and when radix-2 or radix-4 coprocessors

Table 2 4×4 SQRD operational complexity

Operation	$+/-$	$\times$	$\div$	$\surd$
op/second ($\times 10^9$)	3.24	12.72	0.12	0.12

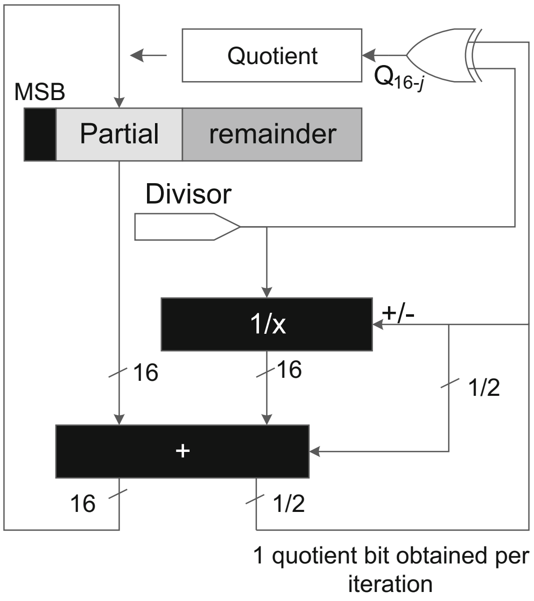

Fig. 4 FPE division coprocessor

Table 3 SQRD division implementations

Solution	Resource			Throughput
	FPEs	DSP48es	LUTs	(MDiv/s)
FPE-P	100	100	13,600	120
FPE-R_2	5	5	900	120
FPE-R_4	4	4	944	144

are added alongside the DSP48e (*FPE-R_2*, *FPE-R_4* respectively) on Virtex 5 FPGA is described in Table 3. The *FPE-R_2* and *FPE-R_4* solutions both increase throughput, by factors of 8.9 and 13.3 respectively and hence increase hardware efficiency by respective factors of 9.4 and 10.7 as compared to *FPE-P*. Since 4×4 802.11n MIMO requires 120 MDiv/s for SQRD-based preprocessing, the implied cost and performance metrics of each option are summarised in Table 3. According to these estimates, *FPE-R_2* represents the lowest cost real-time solution, enabling a 93.4% reduction in resource cost relative to *FPE-P*. This approach is adopted in the FPE-based SQRD implementation.

To realise the 120×10^6 square root operations required per second (MSQRT/s), performance and cost estimates for software-based execution on the FPE using the pencil-and-paper method [19] (*FPE-P*), or by adding a CORDIC coprocessor [28] (*FPE-C*) are compared in Table 4(a). The coprocessor-based *FPE-C* solution at once increases throughput and efficiency by factors of 23 and 10 respectively as compared to *FPE-P*, implying the resources required to realise real-time square-root for SQRD-based detection of 4×4 802.11n MIMO can be estimated as in Table 4(b). As this shows, *FPE-C* enables real-time performance using only 11% of the resource required by *FPE-P*, and is adopted for realising FPE-based square root operations.

Table 4 FPE square root options

(a) 16-Bit PSQRT, CSQRT			(b) 802.11n SQRD		
	FPE-P	*FPE-C*		*FPE-P*	*FPE-C*
PM/RF locations	29/14	8/1	FPEs	63	3
LUTs	142	330	LUTs	8946	990
DSP48es	1	0	DSP48es	63	3
Clock (MHz)	367.7	350	T (MSQRT/s)	121.6	130.8
Latency (cycles)	191	8			
T (MSQRT/s)	1.93	43.6			
T/LUT ($\times 10^{-3}$)	13.6	132.1			

4.2 SQRD Using FPGA

Integrating these components into a coherent processing architecture to perform SQRD, and replicating that behaviour to provide PP for the 108 subcarriers of 802.11n MIMO is a large scale accelerator design challenge. Figure 5 describes the SQRD algorithm as a, iterative four-task (T_1, $T_{2.1}$–$T_{2.3}$) process. The first task, T_1, conducts channel norm ordering, and computes the diagonal elements of **R** (lines 11–13 in Algorithm 1). This is followed by $T_{2.1}$–$T_{2.3}$, which are independent and permute and update **Q**, **R** and **norm** respectively (lines 14–18 in Algorithm 1).

This process is realised using a 4-FPE Multiple Instruction, Multiple Data (MIMD) architecture, shown in Fig. 6, is used. All FPEs employ 16-bit datapaths and are otherwise configured as described in Table 5(a). FPE_1–FPE_3 permute **Q**, **R** and **norm** and iteratively update ($T_{2.1}$–$T_{2.3}$ in Fig. 5). FPE_4 calculates the diagonal elements of **R** (T_1). The SQRD process executes in three phases. Initially, **H** and the calculation of **norm** are distributed amongst the FPEs, with the separate parts of **norm** gathered by FPE_4 to undergo ordering, division and square root. The results are distributed to the outer FPEs for permutation and update of **Q**, **R** and **norm**. Inter-FPE communication occurs via point-to-point FIFO links, chosen due to their relatively low cost on FPGA and implicit ability to synchronize the multi-FPE architecture in a data-driven manner whilst avoiding data access conflicts.

The performance and cost of the 4-FPE grouping is given in Table 5(b). According to these metrics, the throughput of each 4-FPE group is sufficient to support SQRD-based PP of 3 802.11n subcarriers. To process all 108 subcarriers, the architecture is replicated 36 times, as shown in Fig. 6. The mapping of subcarriers to groups is as described in Fig. 6.

On Xilinx Virtex 5 VSX240T FPGA, the cost and performance of this architecture is described in Table 5(b). As this describes, 32.5 MSQRD/s are achieved, in excess of the 30 MSQRD/s required for 4×4 802.11n MIMO.

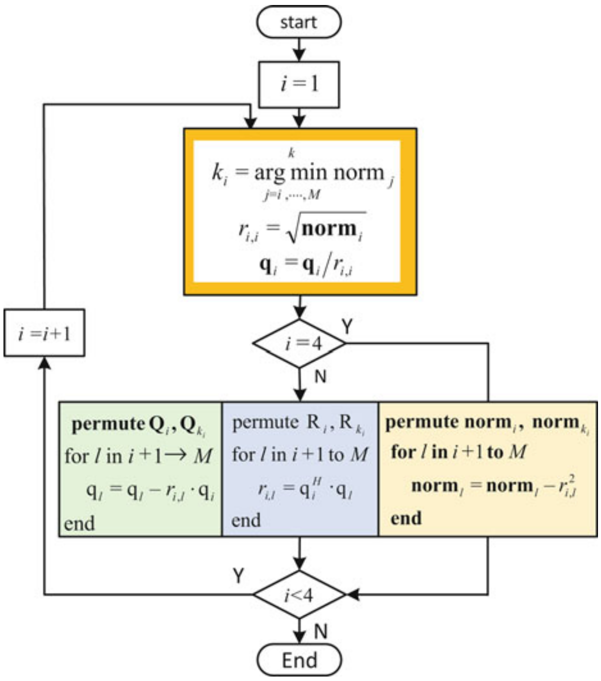
start
$i = 1$
$k_i = \arg\min_{j=i,\dots,M} \text{norm}_j$
$r_{i,i} = \sqrt{\text{norm}_i}$
$\mathbf{q}_i = \mathbf{q}_i / r_{i,i}$
$i = 4$
Y
N
$i = i+1$
permute $\mathbf{Q}_i, \mathbf{Q}_{k_i}$
for l in $i+1 \rightarrow M$
$\mathbf{q}_l = \mathbf{q}_l - r_{i,l} \cdot \mathbf{q}_i$
end
permute $\mathrm{R}_i, \mathrm{R}_{k_i}$
for l in $i+1$ to M
$r_{i,l} = \mathbf{q}_i^H \cdot \mathbf{q}_l$
end
permute norm_i, norm_{k_i}
for l in $i+1$ to M
$\text{norm}_l = \text{norm}_l - r_{i,l}^2$
end
$i<4$
Y
N
End

Fig. 5 4 × 4 SQRD

5 FSD Tree-Search for 802.11n

Computing MCS for FSD in 4 × 4 16 QAM 802.11n is even more computationally demanding than SQRD-based preprocessing. The operational complexity is described in Table 6(a). When a single 4 × 4 16-QAM FSD MCS is implemented on a *16R* FPE, the performance and cost are as reported as *16R-MCS* in Table 6(b).

To scale this performance to support all 108 subcarriers for 4 × 4 16-QAM 802.11n MIMO, a large-scale architecture is required. Two important observations of the application's behaviour help guide the choice of multiprocessing architecture:

1. THE FSD MCS tree exhibits strong SIMD-like behaviour, where each branch (Fig. 3a), performs an identical sequence of operations on data-parallel samples.
2. The number of FPEs required to implement MCS for all 108 OFDM subcarriers on a single, very wide SIMD processor implies limitations on the achievable clock rate as a result of high signal fan-outs to broadcast instructions from a central PM to a very large number of ALUs, restricting performance [10]. Hence, a collection of smaller SIMDs is used.

As described in Table 6(b), the cost of *16R-MCS* as compared to the basic 16-bit FPE described in Sect. 2 (from 90 LUTs to 2530 approximately) is significantly higher. This large increase is due to the large PM required to house the 4591

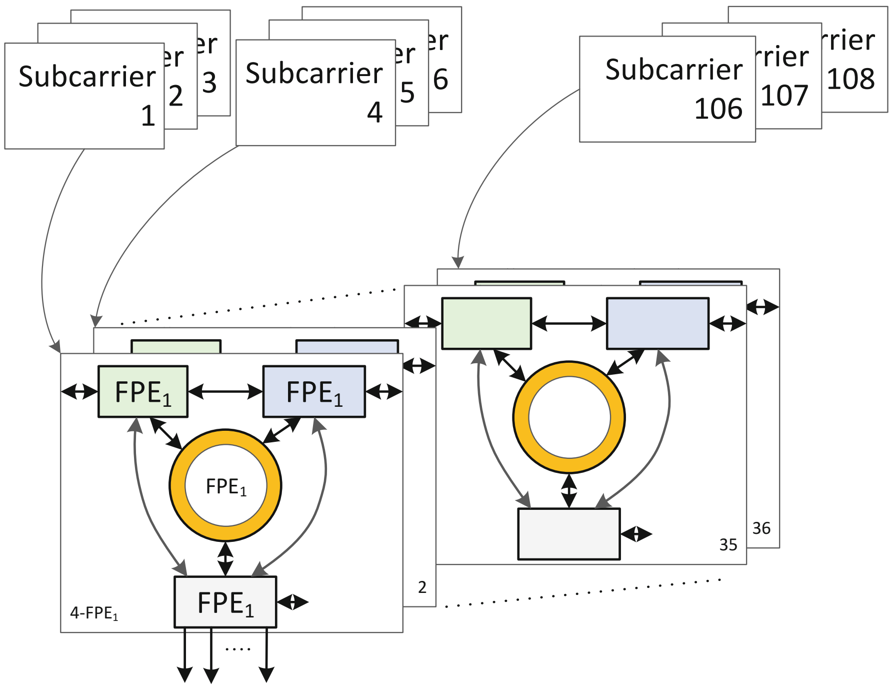

Fig. 6 4 × 4 SQRD mapping

Table 5 4-FPE-based SQRD

(a) FPE configuration		(b) FPE-SQRD metrics		
Parameter	Value		4-FPE$_1$	FPE-SQRD
PMDepth	350	LUTs	2109	70,560
RFDepth	32	DSP48es	4	144
IMMDepth	32	Clock (MHz)	315	265
DMDepth	64	T (MSQRD/s)	1.07	32.5
TxComm	32	Latency (μS)	0.9	1.1
RxComm	32			

instructions. A significant factor in this large number of instructions are the comparison operations required for slicing (Eq. (3)) and sorting the PED metrics, which require branch instructions, which have associated NOP operations due to the deep FPE pipeline and the lack of forwarding logic [10]. These represent wasted cycles and dramatically increase cost and reduce throughput—branch and NOP instructions represent 50.7% of the total number of instructions. Optimising the FPE to reduce the impact of these branch instructions could have a significant impact on the MCS cost/performance.

Table 6 802.11n MCS complexity

(a) Operational complexity		(b) MCS implementation options		
Operation	op/s ($\times 10^9$)		*16R-MCS*	16R
+/−	32.37	LUTs	2520	805
×	19.20	DSP48es	1	0
		Clock (MHz)	367.7	350
		L (Cycles)	3281	1420
		T (MOP/s)	1.9	4.5

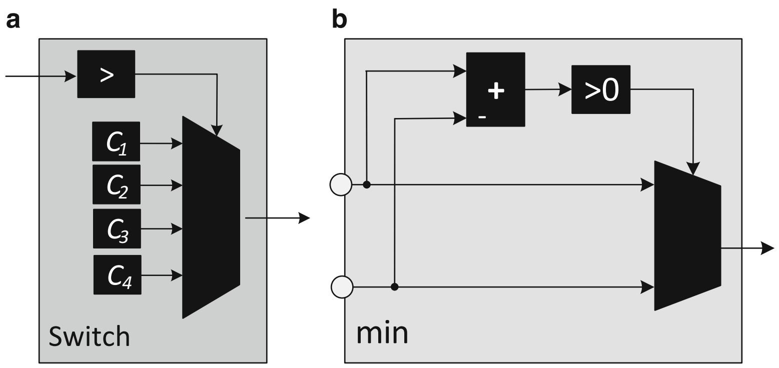

Fig. 7 (**a**) Switch coprocessor (**b**) Min coprocessor

5.1 FPE Coprocessors for Data Dependent Operations

Employing ALU coprocessors can significantly reduce these penalties. A switch coprocessor compares the input to each of four constants, determined pre-synthesis (a logical depiction of behaviour is shown in Fig. 7a), selecting the closest. This increases the efficiency of slicing by comparing an input operand to one of a number of pre-defined values. Similarly, a MIN coprocessor (Fig. 7b) can be used to accelerate sorting.

Each of these coprocessors occupy around 20 LUTs, but their ability to eliminate wasted instructions can significantly reduce the PM size. This can enable significant reductions in overall cost and increases in performance as described in column 3 of Table 6(b). Including these components results in a 68% reduction in resource cost and a factor 2.3 increase in throughput. The resulting component is capable of realising FSD MCS for a single 802.11n subcarrier in real-time, providing a good foundation unit for implementing MCS for all 108 subcarriers.

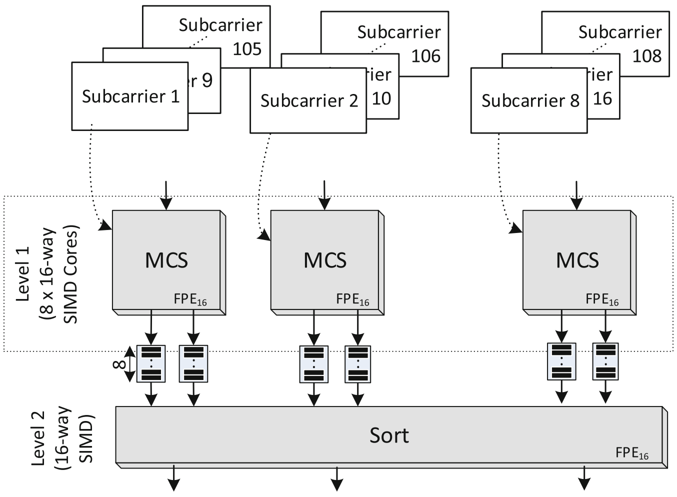

Fig. 8 802.11n OFDM MCS-SIMD mapping

5.2 *SIMD Implementation of 802.11n FSD MCS*

To scale the FPE to realise all 108 subcarriers, a range of architectures may be used. The data-parallel operation of the subcarriers suggests that a very wide single SIMD could be used, providing the most efficient realisation from the perspective of PM and control logic cost. However, as the width of an FPE SIMD unit increases beyond 16 lanes, the instruction broadcast from the single central PM limits the speedup which may be obtained by constraining the clock frequency. Hence, 16-way SIMDs are employed and FSD MCS for all 108 802.11n subcarriers is implemented on a dual-layer network of such processors, as illustrated in Fig. 8.

Level 1 consists of eight SIMDs. The 802.11n subcarriers are clustered into eight groups $\{G_i = \{j : (j-1) \mod 8 = i\}_{j=1}^{108}\}_{i=0}^{7}$, where j is the set of subcarriers processed by FPE i. The 16 branches of the MCS tree for each subcarrier are processed in parallel across the 16 ways of the Level 1 SIMD onto which they have been mapped. Sorting for the subcarriers implemented in each Level 1 SIMD is performed by adjacent pairs of ways in the Level 2 SIMD—hence given the 8 Level 1 SIMDs, the Level 2 SIMD is composed of 16 ways.

Each FPE is configured to exploit 16-bit real-valued arithmetic [6]. All processors exploit `PMDepth` = 128, `RFDepth` = 32 and `DMDepth` = 0, and communication between the two levels exploit 8-element FIFO queues. The Level 1 SIMDs incorporate SWITCH coprocessors to accelerate the slicing operation, whilst the Level 2 SIMDs support the MIN ALU extension to accelerate the sort operation.

The program flow for each Level 1 SIMD is as illustrated in Fig. 9a. Each FPE performs a single branch of the MCS tree, with the empty parts of the program flow—representing NOP instructions—used to properly synchronise movement of data into and out of memory.

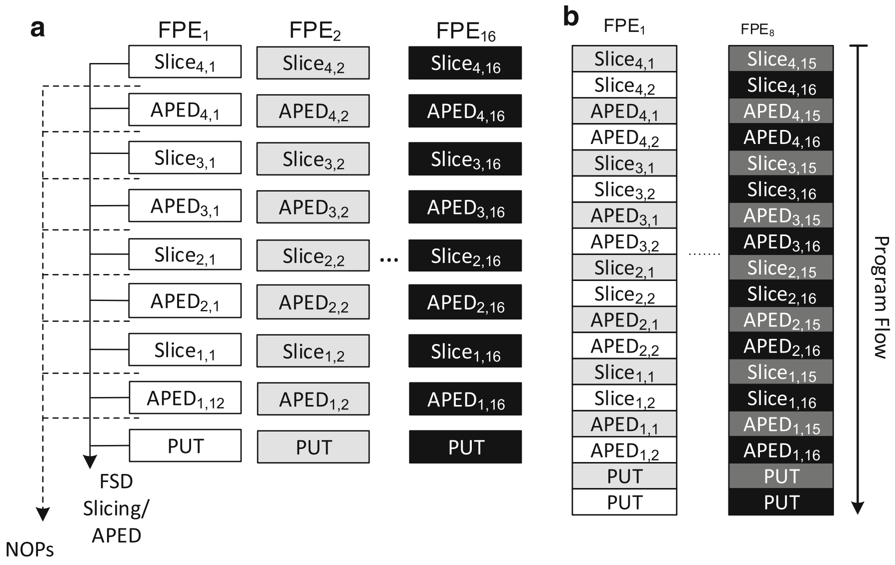

Fig. 9 FPE branch interleaving. (**a**) Original FSD threads. (**b**) Interleaved threads

Table 7 4 × 4 16-QAM FSD using FPE

	FPE-MCS	FPE-FSD
LUT	16,601	96,115
DSP48e	144	408
Clock (MHz)	296	189
T (Mbps)	502.5	483
L (μS)	0.9	2.3

The NOP cycles represent 29% of the total instruction count but since they represent ALU idle cycles they should preferably be eliminated. To do so, NOP cycles in one branch can be occupied by the useful, independent instructions from another, i.e. the branches may be interleaved as illustrated in Fig. 9.This interleaving occupies wasted NOP cycles, to the extent that when two branches are interleaved the proportion of wasted cycles is reduced to 4%.

On Xilinx Virtex 5 VSX240T FPGA, this multi-SIMD architecture enables FSD-MCS for 802.11n as reported Table 7. As this shows, it comfortably exceeds the real-time performance criteria of 802.11n.

Together with the results of the SQRD preprocessing accelerator, these MCS metrics show that the FPE can support accelerators for applications with demanding real-time requirements. By using massively parallel networks of simple processors (>140 in this case), FPGA can support real-time behaviour and can enable solutions with resource cost comparable to custom circuits. When the PP and MCS are combined to create a full FSD detector (*FPE-FSD* in Table 7) the resulting architecture is the only software-defined FPGA structure to enable real-time performance for 4 × 4 16 QAM 802.11n.

6 Stream Processing for FPGA Accelerators

The FPE is a load-store structure, supporting only register-register and immediate instructions. All non-constant operands and results access the ALU via Register File (RF). Consider the effect of this approach for a 256-point FFT (FFT_{256}) realised using two FPE configurations: an 8-way FPE SIMD (FPE_8) or a MIMD multi-FPE composed of 8 SISD FPEs (8-FPE). The FFT mappings and itemized ALU, communication (IPC), memory (MEM) and NOP instructions for each are shown in Fig. 10.

Figure 10 shows that the efficiency of each of these programs is low—only 52.5% and 31.8% of the respective cycles in 8-FPE_1 and FPE_8 are used for ALU instructions. The resulting effect on accelerator performance and cost is clear from Table 8, which compares 8-FPE_1 with the Xilinx Core Generator FFT [29] component. The FPE is not competitive with the custom circuit Xilinx FFT, which exhibits twice the performance at a fraction of the LUT cost.

These results follow from the restriction to register-register instructions. Each FFT_{256} stage consume 512 complex words. Since RF is the most resource-costly element of the FPE, buffering this volume of data requires BRAM Data Memory (DM); in order for these operands to be processed and results stored, a large number of loads (stores) are required between BRAM and RF, increasing PM cost. Given the simplicity of the FFT butterfly operation, the overhead imposed by these is significant. This is combined with the effect of the FPE's requirement to be standalone: since it must handle its own communication, further cycles are consumed transferring incoming and outgoing data between DM and COMM, reducing program efficiency still further. Finally, each of these transfers induces a latency between source and destination—as Fig. 11 illustrates, each FPE

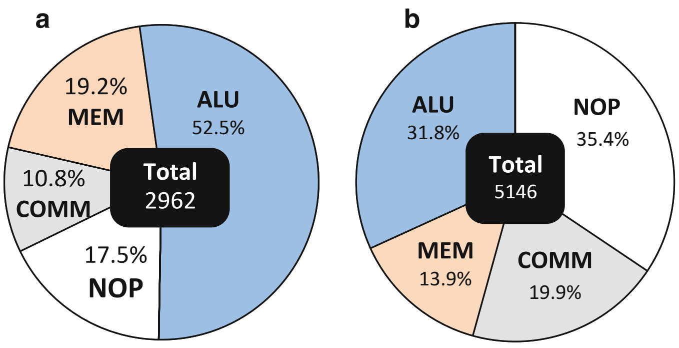

Fig. 10 FFT_{256}: FPE-based 256 Point FFT. (**a**) 8-FPE_1. (**b**) FPE_8

Table 8 256-Point FFT performance/cost comparison

	Cost		T	T/LUT
	LUTs	DSP48e	(MSamples/s)	($\times 10^3$)
8-FPE_1	2296	8	30.5	13.3
Xilinx	621	6	61.9	99.7

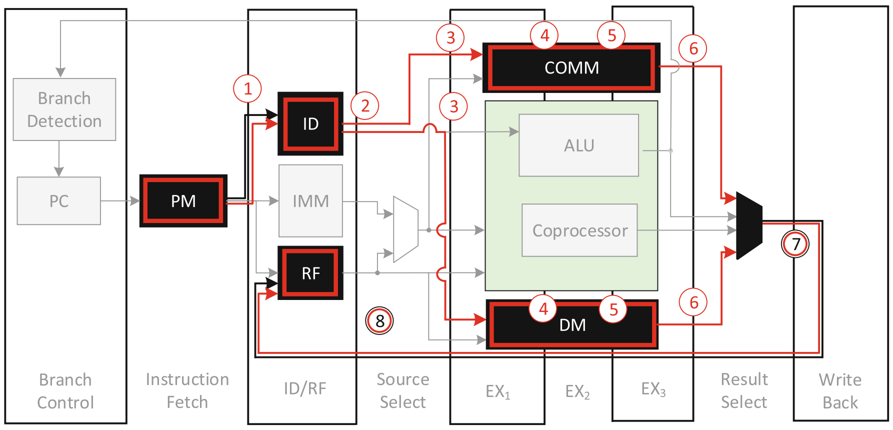

Fig. 11 Load-store paths in the FPE

DM-RF (black) and COMM-RF (red) transfer takes eight cycles, imposing the need for **NOP**s.

These factors combine to severely limit the efficiency of the FPE for applications such as FFT. Mitigating the effect of these overheads requires two features:

- Direct instruction access to any combination of RF, DM and COMM for either instruction source or destination.
- In cases where local buffering is not required, data streaming through the PE should be enabled, reducing load/store and communication cycle overhead.

6.1 Streaming Processing Elements

To support these features, a streaming FPE (sFPE) is proposed. The sFPE is still standalone, software-programmable and lean, but supports a processing approach—streaming—which diverges from the load-store FPE approach. Streaming means that focus is placed on ensuring that data can stream into and out of operation sources and destinations and through the ALU without the need for load and store cycles. This streaming takes two forms:

- Internal: between RF, DM, COMM and IMM without load-store cycles.
- External: from input FIFOs to output FIFOs via only ALU.

The architecture of a SISD sFPE$_1$ is illustrated in Fig. 12. There are three main architectural features of note.

- An entire pipeline stage is dedicated to instruction decode (ID)
- A *FlexData* data manager has been added which allows zero-latency access to any data source or sink.

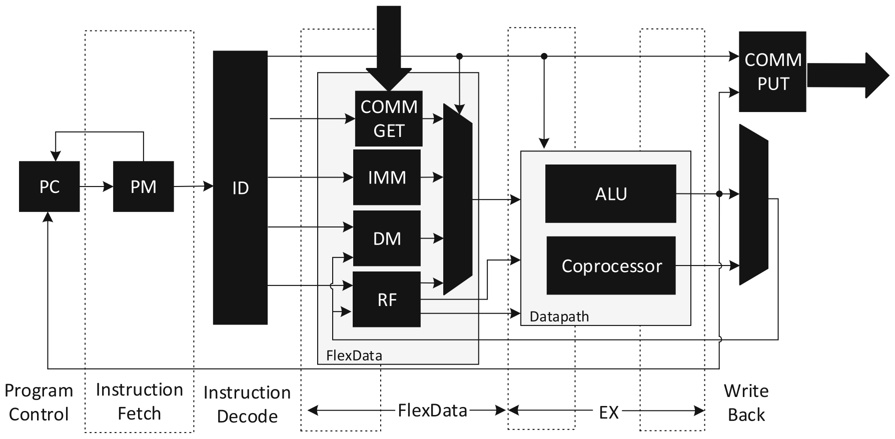

Fig. 12 SISD sFPE architecture

- Off-FPE communication has been decoupled into read (COMMGET) and write (COMMPUT) components

In the sFPE, ID and FlexData are assigned entire pipeline stages. The ID determines the source or destination of any instruction operand or result, with all of the potential sources or destinations of data incorporated in FlexData to allow each to be addressed with equal latency; this flat memory architecture is unique to the sFPE. This approach removes the load/store overhead of accessing, for example, data memory or off-FPE communication; all data operands and results may be sourced/produced to any of IMM, RF, DM or COMM with identical pipeline control and without the need for explicit load and store cycles or instructions for DM or COMM.

To allow unbuffered streaming from input FIFOs or output FIFOs via ALU, simultaneous read/write to external FIFOs is required, with direct access to ALU in both directions. Decoupling the off-FPE communication components into COMMGET and COMMPUT allow each to be accessed with zero-latency, from a single instruction—note that these both reside in the same pipeline stage and hence conform to the regular dataflow pipeline maintained across the remainder of FlexData. In addition, since all of COMMGET, COMMPUT, DM, RF and IMM access distinct memory resources (with separate memory banks employed within the sFPE and a FIFO employed per off-sFPE communication channel) there is no memory bandwidth bottleneck resulting from decoupling these accesses in this way—all could be accessed simultaneously if needed.

Table 9 ALU operand/destination instruction coding

Op	Source/sink	x
Rx	RF	Register location
&x	DM	DM address
^x	COMMGET/COMMPUT	IPC channel no.
x	IMM	Constant value

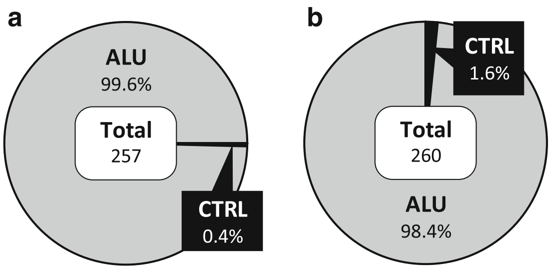

Fig. 13 FFT_{256}: sFPE implementations. (**a**) $8\text{-}sFPE_1$. (**b**) $sFPE_8$

6.2 *Instruction Coding*

To support the increase level of specialisation of the operands in each instruction, however, operand addressing needs to become more complicated. Generally, sFPE ALU instructions take the form:

`INSTR dest, opA, opB, opC`

where `INSTR` is the instruction class, `dest` identifies the result destination and `opA, opB, opC` identify the source operands. The possible encodings of each of `dest, opA,`
`opB, opC` and the destination are described in Table 9.

This encoding allows any of RF, DM, COMMGET and COMMPUT to be addressed directly from the absolute addresses quoted in the sFPE instruction. Constant operands are hard-coded into the instruction and IMM locations allocated by the assembler.

This architecture and data access strategy can lead to sFPE programs which are substantially more efficient that their FPE counterparts. Using the sFPE, the number of instructions needed for FFT_{256} in both the 8-sFPE and $sFPE_8$ variants are described in Fig. 13.

In MIMD 8-sFPE form, the total number of instructions required is 257, a decrease of around 91%. In addition, the efficiency of this realisation is now 99.6%, with only a single non-ALU instruction required for control. Similarly, $sFPE_8$ requires 95.9% fewer instructions and operates with an efficiency of 98.4%. Given these metrics it is reasonable to anticipate increases in throughput for 8-sFPE and $sFPE_8$ by factors of 20 and 30.

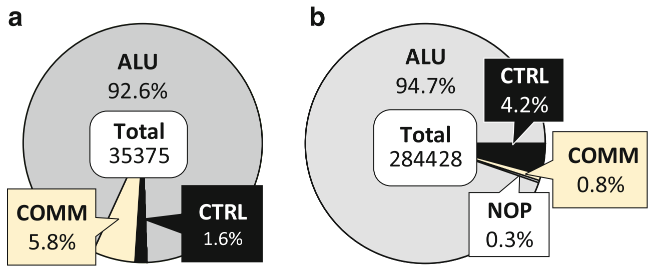

Fig. 14 Itemised sFPE matrix multiplication and ME operations. (**a**) Matrix multiplication. (**b**) Motion estimation

7 Streaming Block Processing

In many operations, however, addressing modes other than the simple direct approach used in the FPE are vital. An itemized instruction breakdown for multiplication of two 32×32 matrices and Full-Search ME (FS-ME) with a 16×16 macroblock on a 32×32 search window are quoted in Fig. 14.

A number of points are notable. Firstly, the programs are very efficient, verifying the techniques described in the previous section. However, the programs are extremely large—35,375 instructions for matrix multiplication (MM) and 284,428 for FS-ME. To store this number of instructions, a very large PM is required, requiring a lot of FPGA resources—for FS-ME, 241 BRAMs would be required for the PM alone. These demands are a direct result of the FPE's restriction to direct addressing. This is because, in a direct addressing scheme then every operation requires an instruction; for MM and ME, this translates a very large number of instructions.

However, both of these operations and their operand accesses are very regular and can be captured in programs with many fewer instructions than those quoted above. Both repeat the same operation many times on small subsets of the input data at regularly-spaced memory locations. For example, Bock-MM of two matrices $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{n \times p}$ when $m = n = p = 8$ via four 4×4 submatrices. Assuming that A and B are stored in contiguous memory locations in row-major order and that C is derived in row-major order, the operand memory access are as illustrated in Fig. 15.

To compute an element of a submatrix of C, the inner product of a four-element vector of contiguous locations in A (a row of the submatrix) and a four-element vector of elements spaced by 8 locations in B (a column of the submatrix) is formed. Afterwards either or both of the row of A or column of B are incremented to derive the next element of C, before operation proceeds to the next submatrix. The resulting memory accesses are highly predictable: a regular repeated increment along the rows of A and columns of B, periodic re-alignment to a new row of A and/or column of B, repeated multiple times before realigning for subsequent submatrices.

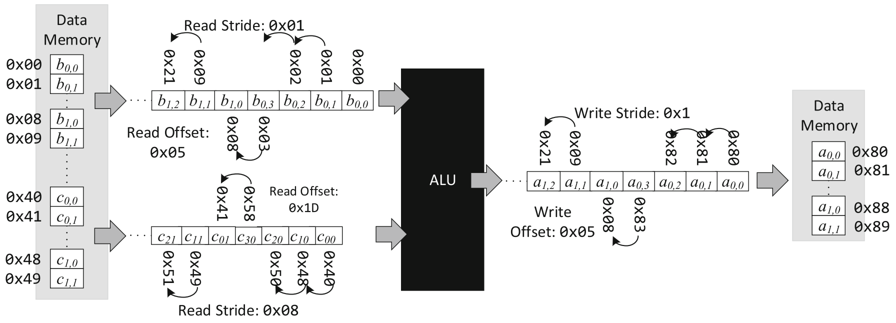

Fig. 15 sFPE block matrix multiply operand addressing

These patterns can be used to enable highly compact programs if two features are available—`repeat`-style behaviour with the ability for a single instruction to address blocks or memory are regularly-spaced locations when invoked multiple times by a `repeat`.

7.1 Loop Execution Without Overheads

To enable low-overhead loop operation, the sFPE is augmented with the ability to perform `repeat`-type behaviour. This means managing the PC such that when a `repeat` instruction is encountered, the body of the associated block of statements is executed a number of times. This task if fulfilled by a PC Manager (PCM), the behaviour of which is described in Fig. 16.

The PCM controls PC update given its previous value and the instruction referenced in PM given pieces of information—the start and end lines of the body statements to be repeated `S` and `E`, the number of repetitions `N`. These are encoded in a `RPT` instruction added to the sFPE instruction set. These instructions are encoded as:

`RPT N S E`

The behaviour of `RPT` is shown in Listing 1. This dictates five repetitions of lines 2–5. Any number of repeat instructions can be nested to allow efficient execution of loop nests with static and compile-time known loop bounds.

Listing 1 RPT Instruction Coding

```
RPT 5 2 4
  INSTR1...
  INSTR2...
  INSTR3...
```

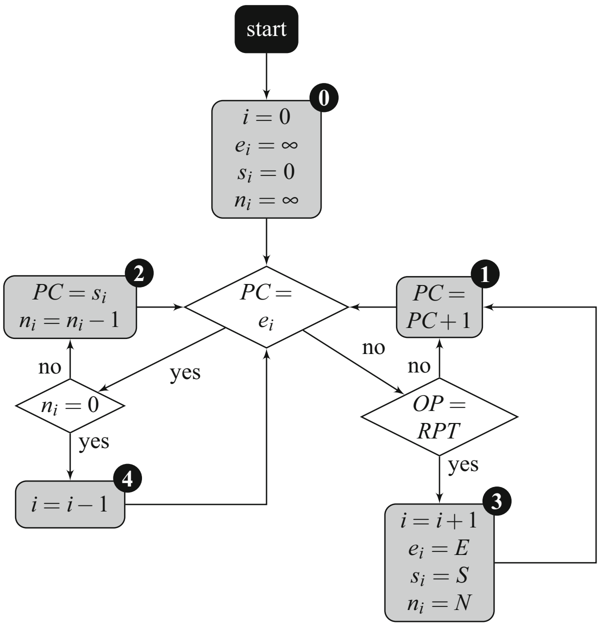

Fig. 16 sFPE PCM behaviour

The PCM arbitrates the PC to ensure that the body statements are repeated the correct number of times and support the construction of nested repeat operations. It enacts the flowchart in Fig. 16. For an n-level nest it maintains a $n + 1$-element lists of metrics, with an additional element added to support infinite repetition of the top-level program, considered to be an implicit infinite repeat instruction. For layer i of the loop nest, the start line, end line and number of repetitions are stored in element $i + 1$ of the lists s, e and n respectively. In all cases $s_0 = 0$, $e_0 = \infty$ and $n_0 = \infty$ to represent the start line, end line and number of repetitions of the top-level program (0 in Fig. 16).[1] Every time a repeat instruction is encountered i, the current index into s, e and n is incremented and the values of the new element initialised using `S`, `E` and `N` from the decoded instruction in 3. Regular PC updating then proceeds (1) until either another repeat instruction is detected or until e_i is encountered. In the latter case, the number of iterations of the current statement is decremented (2) or, if $n_i = 0$ all of the iterations of the current repeat statement have been completed and control of the loop nest reverts to the previous level (4).

The PCM component requires 36 LUTs and hence imposes a relatively high resource cost as compared to the FPE. This can be controlled by compile-time customisation via the parameters listed in Table 10.

[1]Note that this assumes that the end line of the program is a **`JMP`** instruction with the start line as the target.

Table 10 PC configuration parameters

Parameter	Meaning	Values
`pcm_en`	Enable/disable PCM	Boolean
`pcm_depth`	Max. repeat nest depth	$\mathbb{N} \in [1, 2^{32} - 1]$

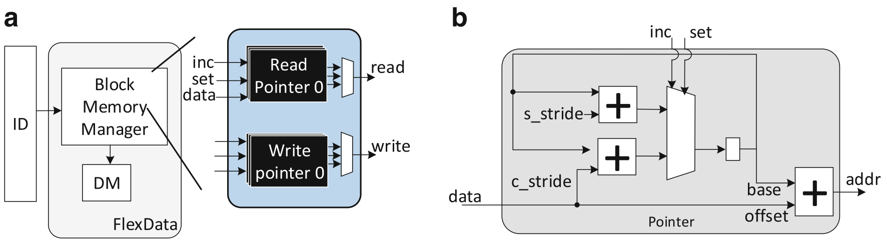

Fig. 17 sFPE block memory management elements. (**a**) sFPE FlexData. (**b**) Pointer Architecture

The `pcm_en` parameter is a Boolean which dictates whether the PCM is included or not. When it is, the maximum depth of loop nest is configurable via `pcm_en` which can take, hypothetically, any integer value. As such, the PCM may be included or excluded and hence imposes no cost when it is not required; further, when it is included its cost can be tuned to the application at hand by adjusting the maximum depth of loop nest.

7.2 *Block Data Memory Access*

Enabling block memory access requires three important capabilities:

- Auto-increment with any constant stride
- Manual increment with any stride
- Custom offset

The need for each of these is evident in MM: auto-increment traverses along rows and columns with a fixed memory stride—there are many such operations and so eliminating the need for an individual instruction for each reduce overall instruction count considerably. Manual increment is required for movement between rows/columns, whilst custom offset is used to identify the starting point for the increments, such as the first element of a submatrix.

A *Block Memory Manager* (BMM) is incorporated in the sFPE FlexData, as illustrated in Fig. 17a, to enable these properties. The BMM arbitrates access to DM via *Read Pointers* (RPs) and *Write Pointers* (WPs). The architecture of FlexData and a pointer is illustrated in Fig 17b.

Each pointer controls access to a subset (block) of the sFPE DM and addresses individual elements of that block via a combination of two subaddress elements: a

Table 11 BMM configuration parameters

Parameter	Meaning	Values
mode	Addressing mode	Direct, block
n_rptrs / n_wptrs	No. of read /write pointers	$\mathbb{N} \in [1, 2^{32}]$
s_stride	Constant stride	$\mathbb{N} \in [1, 2^{32}]$

Table 12 BMM instructions

Operand field	Meaning
INC_RP / INC_WP	Increment base of RP/WP n to val
SET_RP /SET_WP	Set offset of RP/WP n to val

Table 13 ALU block operand instruction coding

Operand field	ofs	idx	!
Meaning	Offset	Pointer reference	Autoincrement base

base and an **offset**. The offset selects the root block data element whilst the base iterates over elements relative to the offset.

Pointers operate in one of three modes. Either the base auto-increments, or it is incremented by explicit instruction, or the offset increments by explicit instruction. All three modes are supported under the control of the set, inc and data interfaces. The offset selects the root data element of the submatrices of A, B and C, with the base added to address elements relative to the offset. The base is updated via two mechanisms, under the control of inc. The first auto-increments by a value (s_stride in Fig.17b) set as a constant at synthesis time. Manually incrementing the base is achieved by c_stride, which is defined at run-time. Finally, when update of the offset is required, data is accepted on assertion of set. To allow absolute minimum cost for any operation, configuration parameters for the sFPE FlexData, BMM and pointer components are configurable by the parameters in Table 11.

It is notable that addressing mode is now a configuration parameter of the sFPE, with direct and block modes supported. In direct mode, the BMM is absent whilst it is included in the block mode. In that case, the cost can be minimised via control of the number of read and write pointers via n_rptrs and n_rptrs. Finally, the auto-increment stride s_stride for each pointer is fixed at the point of synthesis.

To support custom increment of the base and offset for each pointer, BMM instructions take the form

```
INSTR n val
```

where n specify the pointer. The permitted values of INSTR are given in Table 12.

ALU operands accessing DM have an encoding of the form &<ofs><idx><!>, elaborated in Table 13.

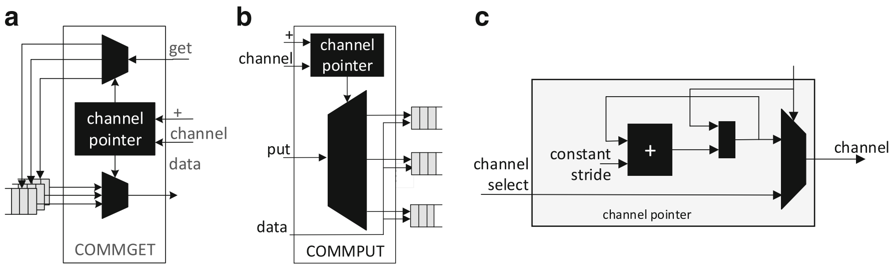

Fig. 18 sFPE COMM adapters. (**a**) COMMGET. (**b**) COMMPUT. (**c**) COMM pointer

Table 14 COMM configuration parameters

Parameter	Meaning	Values
`mode`	Addressing mode	Direct, block
`n_chan`	No. channels	$\mathbb{N} \in [1, 64)$
`s_stride`	Constant stride	$\mathbb{N} \in [1, 64)$

Table 15 sFPE-based MM and ME: itemized PM

	Matrix multiply			Motion estimation		
Class	sFPE	sFPE-B	δ (%)	sFPE	sFPE-B	δ (%)
ALU	32,768	32	−99.9	268353	26	−99.9
COMM	2048	6	−99.7	2467	14	−99.4
CTRL	559	4	−99.7	12582	12	−99.9
NOP	0	6		1026	6	−99.6
Total	35,375	54	−99.8	284428	58	−99.9

7.3 *Off-sFPE Communications*

The COMMGET and COMMPUT components, illustrated in Fig. 18 are also both configurable according to the parameters in Table 14.

Each of COMMGET and COMMPUT can operate under direct and block addressing modes. In direct mode, individual FIFO channels and be accessed via addresses encoded within the instruction. Instructions for either COMM unit are encoded as:

`^<p><ofs/idx><!>`

where `p` differentiates *peek* (read-without-destroying) and *get* (read-and-destroy) operations, `ofs` denotes the offset, `idx` the pointer reference and `!` autoincrement.

7.4 *Stream Frame Processing Efficiency*

The effect of these streaming and block addressing features can be profound. The number of instructions required by direct (sFPE) and block-based (sFPE-B) sFPE modes are quoted in Table 15. Very large reductions in program size have resulted from the addition of block memory management—sFPE-B requires fewer than 1% of the number of instructions required by sFPE. Hence, the stream processing and advanced program and memory control features of the sFPE have a clear beneficial effect on program efficiency and scale. Section 8 compares sFPE-based accelerators for a number of typical signal and image processing operations against real-time performance criteria and custom circuit and soft processor alternatives.

8 Experiments

Accelerators were created using the sFPE for five typical operations:

- 512-point Fast Fourier Transform (FFT)
- 1024 × 1024 Matrix Multiplication
- Sobel Edge Detection (SED) on 1280 × 768 image frames.
- FS-ME: 16 × 16 macroblock, 32 × 32 search window on CIF 352 × 288 images.
- Variable Block Size ME (VBS-ME) with 16 × 16 macroblock, 32 × 32 search window on CIF 720 × 480 images.

The sFPF configurations used to realise each of these operations are described in Table 16. All accelerators target Xilinx Kintex®-7 XC7K70TFBG484 using Xilinx ISE 14.2.

These configurations expose the flexibility of the sFPE. One notable feature is the complete absence of RF in many components, such as MM, FS-ME and FFT. This is a very substantial resource saving which has been enabled as a result of the sFPE being able to stream data from and to COMM components and DM. This flexibility also enables a number of performance and cost advantages, as quoted in Fig. 19. Specifically, the FSME accelerator exhibits real-throughput for H.264; VBS-ME can support real-time processing of 480p video in H.264 Level 2.2. To the best of the authors' knowledge, these are the first time an FPGA-based software-programmable component has demonstrated this capability.

To compare the performance and cost of sFPE-based accelerators relative to custom circuits, sFPE FFTs for IEEE 802.11ac have been developed and compared

Table 16 sFPE-based accelerator configurations

	MM	FS-ME	SED	FFT
Config.	sFPE$_8$	sFPE$_{32}$	3-sFPE$_3$	5-sFPE
`data_ws`	32	16	16	16
`data_type`	Real	Real	Real	Complex
`dm_depth`	1024	1009	1800	[0,32,32,128,512]
`pm_depth`	64	64	113	[68,78,190,758,1949]
`rf_depth`	0	0	32	0
`n_rptrs`	2	2	1	1
`n_wptrs`	1	1	1	1

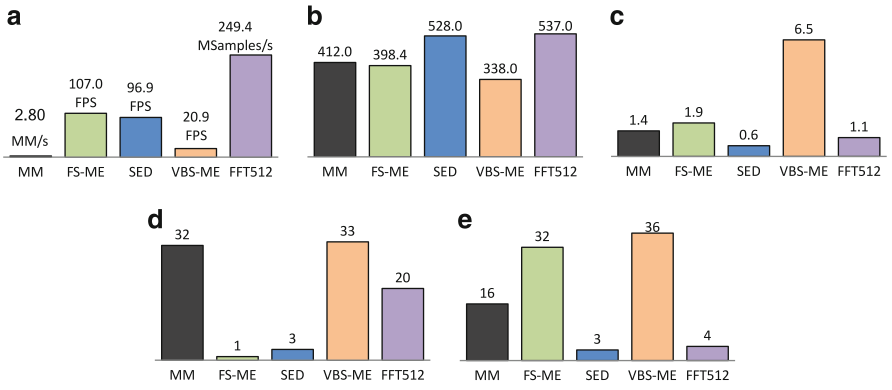

Fig. 19 sFPE accelerators. (**a**) T. (**b**) clk (MHz). (**c**) LUTs. (**d**) DSP48e. (**e**) BRAM

Table 17 802.11ac FFT characteristics

Frequency (MHz)	20	40	80	160
FFT	64	128	256	512
Throughput ($\times 10^6$ Samples/s)	160	320	640	1280

Table 18 sFPE FFT configurations

Parameter	FFT_{64}	FFT_{128}	FFT_{256}	FFT_{512}
Config.	1-$sFPE_3$	1-$sFPE_8$	3-$sFPE_8$	5-$sFPE_8$
data_ws	16			
data_type	Complex			
dm_depth	192	128	[32,256]	[0,32,32,128,512]
pm_depth	1184	902	[134,1852]	[68,78,190,758,1949]
rf_depth	0			
sm_depths	32	64	[32,128]	[0,32,32,64,256]

to both the Xilinx FFT and those generated by Spiral [18]. The IEEE 802.11ac standard [1] mandates 8-channel FFT operations on 20 MHz, 40 MHz, 80 MHz and 160 MHz frequency bands with FFT size and throughput requirements as outlined in Table 17.

These multi-sFPE accelerator configurations are summarised in Table 18—in the case where more than one sFPE is used, the configurations of each are presented in vector format.[2] The performance and cost of the resulting architectures are described in Fig. 20.

Figure 20 shows that the sFPE FFT accelerators for 802.11ac, supported by clock rates of 528 MHz (FFT_{64}, FFT_{128}), 506 MHz (FFT_{256}) and 512 MHz (FFT_{512}), the real-time throughput requirements listed in Table 17 are satisfied. In addition, performance and cost are highly competitive with the Xilinx and Spiral custom circuits. The LUT, DSP48e and BRAM costs are lower than the Xilinx FFT in 9 out

[2]Note that FFT_{512} takes a different configuration to the 512-point FFT previously addressed.

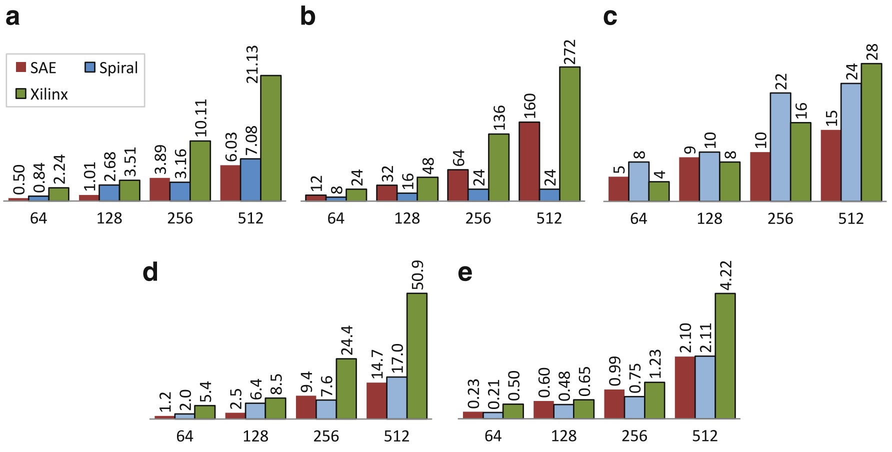

Fig. 20 FPGA-based FFT: performance and cost. (**a**) LUT cost ($\times 10^3$). (**b**) DSP48e cost. (**c**) BRAM cost. (**d**) % device occupied. (**e**) T ($\times 10^9$ Samples/s)

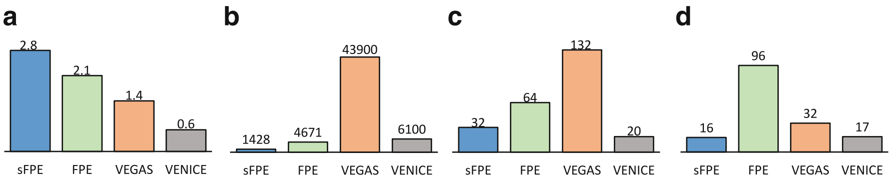

Fig. 21 Softcore matrix multiplication: performance and cost comparison. (**a**) T (MM/s). (**b**) LUTs. (**c**) DSP48e. (**d**) BRAM

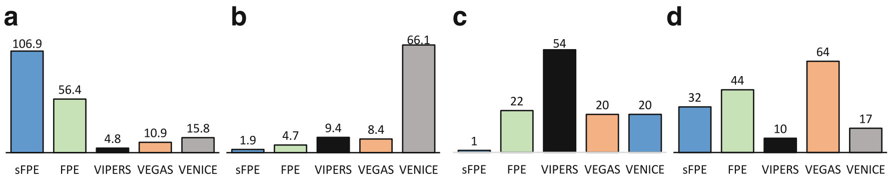

Fig. 22 Softcore FS-ME: performance and cost comparison. (**a**) T (FPS). (**b**) LUTs ($\times 10^3$). (**c**) DSP48e. (**d**) BRAM

of 12 cases, with savings of up to 69, 53 and 56%. Relative to the Spiral FFT, the performance and cost of the sFPE accelerators are similarly encouraging, enabling increased throughput in all but one case and reduced LUT and BRAM costs in 7 out of 8 cases; savings reaching 62.8% and 55% respectively. The Spiral FFTs have consistently lower DSP48e cost, however the total proportion of the device occupied by each, reported in Fig. 20d, remains in favour of the sFPE in all but one instance.

The performance and cost of sFPE-based MM and FS-ME is compared with other soft processors in Figs. 21 and 22.

When applied to MM, the performance and cost advantages relative to 32-way VEGAS (VEGAS$_{32}$) [9] and 4-way VENICE (VENICE$_4$) [24] are clear. Relative to

VEGAS$_{32}$, throughput is increased by a factor 2 despite requiring only 25% of the number of datapath lanes. As compared to VENICE$_4$, throughput is increased by a factor 4.7 whilst LUT and BRAM cost are reduced by 76% and 5% respectively.

sFPE-based ME is compared with VIPERS$_{16}$, VEGAS$_4$ and VENICE$_4$ and the FPE in Fig. 22. sFPE$_{32}$ is the only realisation capable of supporting the 30 FPS throughput requirement for standards such as H.264, with absolute throughput increased by factors of 22.3, 9.8 and 6.8 relative to VIPERS$_{16}$, VEGAS$_4$ and VENICE$_4$.

These results demonstrate the benefit of the sFPE relative to other soft processors—coupled performance/cost increases of up to three orders of magnitude. Of course, the softcores to which the sFPE is compared here are general purpose components and hence offer substantially greater run-time processing capability than the sFPE, which is highly tuned to the operation for which it was created. In that respect, the sFPE is more a component for constructing fixed-function accelerators than a general-purpose softcore. However, despite employing similar multi-lane processing approaches as VIPERS, VEGAS and VENICE the sFPE's focus on extreme efficiency, multicore processing, stream processing and novel block memory management have enabled very substantial performance and cost benefits.

9 Summary

Soft processors for FPGA suffer from substantial cost and performance penalties relative to custom circuits hand-crafted at register transfer level. Performance and resource overheads associated with the need for a host general purpose processor, load-store processing, loop handling, addressing mode restrictions and inefficient architectures combine to amplify cost and limit performance.

This paper describes the first approach which challenges this convention. The sFPE presented realises accelerators using multicore networks of fine-grained, high performance and standalone processors. The sFPE enables performance and cost unprecedented amongst soft processors by adopting a streaming operation model to ensure high efficiency. combined with advanced loop handling and addressing constructs for very compact and high performance operation on large data sets. These enable efficiency routinely in excess of 90% and performance and cost which are comparable to custom circuit accelerators and well in advance of existing soft processors.

Specifically, real-time accelerators for 802.11ac FFT and H.264 FS-ME VBS-ME are described; the former of these exhibits performance and cost which are highly competitive with custom circuits. In addition, it is shown how sFPE-based MM and ME accelerators offer improvements in resource/cost by up to three orders of magnitude. To the best of the authors' knowledge, these capabilities are unique, not only for FPGA, but for any semiconductor technology.

This work lays a promising foundation for the construction of complete FPGA accelerators, but in addition may be used to further ease the design process. For example, in the case where off-chip memory access is required, the programmable nature of the SAE means that it may also be used as a memory controller to execute custom memory access schedules and highly efficient block access. However, resolving this and other accelerator peripheral functions is left as future work.

References

1. 802.11 Working Group: IEEE P802.11ac/D2.2 Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications Amendment 4: Enhancements for Very High Throughput for Operation in Bands below 6 GHz (2012)
2. Altera Inc.: Nios II Processor Reference Handbook (2014)
3. Altera Inc.: Stratix V Device Handbook (2014)
4. Antikainen, J., Salmela, P., Silven, O., Juntti, M., Takala, J., Myllyla, M.: Application-Specific Instruction Set Processor Implementation of List Sphere Detector. In: Conf. Record of the Forty-First Asilomar Conf. on Signals, Systems and Computers, 2007, pp. 943–947 (2007). https://doi.org/10.1109/ACSSC.2007.4487358
5. Barbero, L., Thompson, J.: Fixing the Complexity of the Sphere Decoder for MIMO Detection. IEEE Trans. Wireless Communications pp. 2131–2142 (2008). https://doi.org/10.1109/TWC.2008.060378
6. Barbero, L.G., Thompson, J.S.: Rapid Prototyping of a Fixed-Throughput Sphere Decoder for MIMO Systems. In: IEEE Intl. Conf. on Communications, pp. 3082–3087 (2006). https://doi.org/10.1109/ICC.2006.255278
7. Burg, A., Borgmann, M., Wenk, M., Zellweger, M., Fichtner, W., Bolcskei, H.: VLSI Implementation of MIMO Detection Using The Sphere Decoding Algorithm. IEEE Journal of Solid-State Circuits **40**(7), 1566–1577 (2005). https://doi.org/10.1109/JSSC.2005.847505
8. Cheah, H.Y., F., B., Fahmy, S., Maskell, D.L.: The iDEA DSP Block Based Soft Processor for FPGAs. ACM Trans. Reconfigurable Technol. Syst. **7**(1) (2014)
9. Chou, C.H., Severance, A., Brant, A.D., Liu, Z., Sant, S., Lemieux, G.G.: VEGAS: Soft Vector Processor with Scratchpad Memory. In: Proc. ACM/SIGDA Intl. Symp. Field Programmable Gate Arrays, FPGA '11, pp. 15–24. ACM, New York, NY, USA (2011). https://doi.org/10.1145/1950413.1950420. URL http://doi.acm.org/10.1145/1950413.1950420
10. Chu, X., McAllister, J.: FPGA Based Soft-core SIMD Processing: A MIMO-OFDM Fixed-Complexity Sphere Decoder Case Study. In: IEEE Int. Conf. on Field-Programmable Technology (FPT), pp. 479–484 (2010). https://doi.org/10.1109/FPT.2010.56814639
11. Chu, X., McAllister, J.: Software-Defined Sphere Decoding for FPGA-Based MIMO Detection. IEEE Transactions on Signal Processing **60**(11), 6017–6026 (2012). https://doi.org/10.1109/TSP.2012.2210951
12. Hannig, F., Lari, V., Boppu, S., Tanase, A., Reiche, O.: Invasive Tightly-Coupled Processor Arrays: A Domain-Specific Architecture/Compiler Co-Design Approach. ACM Trans. Embed. Comput. Syst. **13**(4s), 133:1–133:29 (2014). https://doi.org/10.1145/2584660
13. Hanzo, L., Webb, W., Keller, T.: Single and Multi-carrier Quadrature Amplitude Modulation: Principles and Applications for Personal Communications, WLANs and Broadcasting (2000)
14. IEEE802.11n: 802.11n-2009 IEEE Local and metropolitan area networks–Specific requirements Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications Amendment 5: Enhancements for Higher Throughput (2009). https://doi.org/10.1109/IEEESTD.2009.5307322
15. Janhunen, J., Silven, O., Juntti, M., Myllyla, M.: Software Defined Radio Implementation of K-best List Sphere Detector Algorithm. In: Intl. Conf. on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), pp. 100–107 (2008). https://doi.org/10.1109/ICSAMOS.2008.4664852

16. Li, M., Bougard, B., Xu, W., Novo, D., Van Der Perre, L., Catthoor, F.: Optimizing Near-ML MIMO Detector for SDR Baseband on Parallel Programmable Architectures. Design, Automation and Test in Europe (DATE) pp. 444–449 (2008). https://doi.org/10.1109/DATE.2008.4484721
17. McAllister, J.: FPGA-based DSP. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, 2nd edn., pp. 363–392. Springer US (2010)
18. Milder, P., Franchetti, F., Hoe, J.C., Püschel, M.: Computer Generation of Hardware for Linear Digital Signal Processing Transforms. ACM Trans. Des. Autom. Electron. Syst. **17**(2), 15:1–15:33 (2012). https://doi.org/10.1145/2159542.2159547
19. Parhami, B.: Computer Arithmetic: Algorithms and Hardware Designs, 2nd edition edn. OUP USA (2010)
20. Pohst, M.: On The Computation of Lattice Vectors of Minimal Length, Successive Minima and Reduced Bases with Applications. SIGSAM Bull. **15**(1), 37–44 (1981). http://doi.acm.org/10.1145/1089242.1089247
21. Qi, Q., Chakrabarti, C.: Parallel High Throughput Soft-output Sphere Decoder. In: IEEE Workshop on Signal Processing Systems (SIPS), pp. 174–179 (2010). https://doi.org/10.1109/SIPS.2010.5624783
22. Ravindran, K., Satish, N., Jin, Y., Keutzer, K.: An FPGA-based soft multiprocessor system for IPv4 packet forwarding. In: Field Programmable Logic and Applications, 2005. International Conference on, pp. 487–492 (2005). https://doi.org/10.1109/FPL.2005.1515769
23. Schnorr, C.P., Euchner, M.: Lattice Basis Reduction: Improved Practical Algorithms and Solving Subset Sum Problems. Mathematical Programming **66**(1), 181–199 (1994)
24. Severance, A., Lemieux, G.: VENICE: A Compact Vector Processor for FPGA Applications. In: Field-Programmable Technology (FPT), 2012 Intl. Conf. on, pp. 261–268 (2012). https://doi.org/10.1109/FPT.2012.6412146
25. Unnikrishnan, D., Zhao, J., Tessier, R.: Application specific customization and scalability of soft multiprocessors. In: Field Programmable Custom Computing Machines, 2009. FCCM '09. 17th IEEE Symposium on, pp. 123–130 (2009). https://doi.org/10.1109/FCCM.2009.41
26. Wolniansky, P., Foschini, G., Golden, G., Valenzuela, R.: V-BLAST: An Architecture for Realizing Very High Data Rates Over The Rich-Scattering Wireless Channel. In: 1998 URSI Int. Symp. Signals, Systems, and Electronics, pp. 295–300 (1998). https://doi.org/10.1109/ISSSE.1998.738086
27. Wu, B., Masera, G.: A Novel VLSI Architecture of Fixed-Complexity Sphere Decoder. In: 13th Euromicro Conf. on Digital System Design: Architectures, Methods and Tools, pp. 737–744 (2010). https://doi.org/10.1109/DSD.2010.10
28. Xilinx Inc.: LogiCORE IP CORDIC v4.0 (2011)
29. Xilinx Inc.: LogiCORE IP Fast Fourier Transform v7.1 (2011)
30. Xilinx Inc.: 7 Series DSP48E1 Slice User Guide (2013)
31. Xilinx Inc.: 7 Series FPGAs Memory Resources User Guide (2014)
32. Xilinx Inc.: MicroBlaze Processor Reference Guide (2014)
33. Yiannacouras, P., Steffan, J., Rose, J.: Portable, Flexible, and Scalable Soft Vector Processors. Very Large Scale Integration (VLSI) Systems, IEEE Transactions on **20**(8), 1429–1442 (2012). https://doi.org/10.1109/TVLSI.2011.2160463
34. Yu, J., Eagleston, C., Chou, C.H., Perreault, M., Lemieux, G.: Vector Processing as a Soft Processor Accelerator. ACM Trans. Reconfigurable Technology and Systems **2**(2) (2009)