iDEA: A DSP Block Based FPGA Soft Processor

Hui Yan Cheah, Suhaib A. Fahmy, Douglas L. Maskell

School of Computer Engineering
Nanyang Technological University, Singapore
hycheah1@e.ntu.edu.sg,{sfahmy,asdouglas}@ntu.edu.sg

***Abstract*—This paper presents a very lean DSP Extension Architecture (iDEA) soft processor for Field Programmable Gate Arrays (FPGAs). iDEA has been built to be as lightweight as possible, utilising the run-time flexibility of the DSP48E1 primitive in Xilinx FPGAs to serve as many processor functions as possible. We show how the primitive's flexibility can be leveraged within a general-purpose processor, what additional circuitry is needed, and present a full instruction-set architecture. The result is a very compact processor that can run at high speed, while executing a full gamut of general machine instructions. We provide results for a number of simple applications, and show how the processor's resource requirements and frequency compare to a Xilinx MicroBlaze soft core. Based on the DSP48E1, this processor can be deployed across next-generation Xilinx Artix-7, Kintex-7, and Virtex-7 families.**

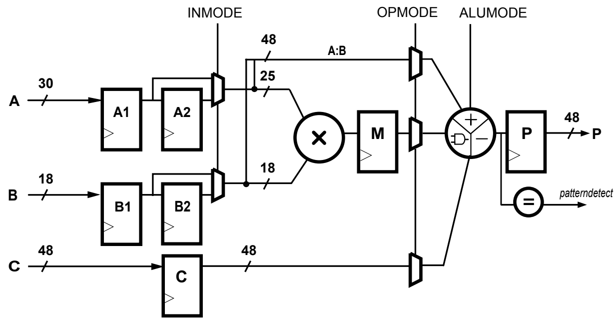

Fig. 1. A Simplified DSP48E1 with Multiplier and ALU.

I. Introduction

The flexibility of Field Programmable Gate Arrays (FPGAs) has been a key feature of the platform, and arises primarily from an architecture that provides a large amount of fine-grained, general purpose resources. However, as FPGAs have found use in particular application domains, and particular core functions have become almost uniformly required, manufacturers have sought to improve their architectures through the provision of hard blocks. After the addition of memory blocks, hard multipliers were added to speed up common signal processing tasks. These later evolved into multiply-accumulation blocks, as needed in filters. Since then, DSP blocks, with a wide range of arithmetic capabilities, have become standard on all architectures across manufacturers and price-points.

The DSP48E1 primitive [1] is found on Xilinx Virtex-6, Artix-7, Kintex-7, and Virtex-7 FPGAs, as well as the Zync-7000 EPP. It boasts increased capability over previous generations, and is also highly customisable. One key feature of this primitive, that has motivated and enabled the work presented in this paper, is its dynamic programmability. The DSP48E1 can support a large range of configurations, many of which can be modified at run-time on a cycle-by-cycle basis. This enables the same instance to be used for many different functions, if a controller is added to set the configuration appropriately. A simplified diagram of the DSP48E1 is shown in Fig. 1.

In this paper, we describe how the DSP48E1 primitive can be manipulated to function as the execution unit of a 32-bit instruction-set processor. While the primitive's usefulness for DSP applications is a given, using it within a processor means that these resources can now be used in a much wider range of application domains, meaning the silicon is not wasted for non-DSP applications. The DSP48E1 is a highly capable block, with the ability to execute a variety of arithmetic operations, from simple addition to multiply-accumulate, and logical operations. We built a lightweight Extension Architecture around the DSP48E1 primitive, resulting in a powerful, comprehensive, general-purpose processor called iDEA.

The design of the iDEA soft processor is very much architecture-focused, in order to offer maximum performance, while being as lean as possible. We have targeted the DSP48E1 as it is present in all Xilinx's 7-series device families. This paper details the architecture and instruction set, the design process, and shows some preliminary execution results.

We feel this work has significant potential when one considers the large number of such primitives available, even on low-end FPGAs. A lean soft processor such as iDEA could pave the way for exploration of massively parallel architectures on reconfigurable fabric, along with investigations on how best to arrange and program such systems.

The remainder of this paper is organised as follows: Section II covers related work. Section III presents the architecture and instruction set of the iDEA processor. Section VI presents hardware implementation results and some software execution examples. Finally, Section VII concludes the paper and presents our future work.

II. Related Work

While pure algorithm acceleration is often done through the design of custom parallel architectures, many supporting tasks are more suited to software implementation. Hence, general processing cores have long been used in FPGA-based

978-1-4673-2845-6/12/$31.00 © 2012 IEEE

systems, and now, more often than not, FPGA-based systems incorporate some sort of processor.

FPGA vendors previously offered hard processors, such as the PowerPC 405 in the Xilinx Virtex II Pro series, however these consumed significant silicon area, and required extensive supporting infrastructure to be added in logic. Meanwhile soft processors were widely adopted in many applications due to their relative simplicity, customisability, and better toolchain support. Commercial soft processors include the Xilinx Microblaze [2], Altera Nios II [3], ARM Cortex-M1 [4], and LatticeMico32 [5], in addition to the open-source Leon3 [6].

FPGA vendors' processors are generally restricted to their own platforms, hence this limits device choice when such cores are used in a design. Some effort has been put into porting these cores to alternative architectures [7], [8], [9]. However, the more generalised a core is, the less closely it fits the low-level target architecture, and hence, the less efficient its implementation and speed. This trade-off between portability and efficiency is an important choice that must be made by the system designer.

Research on soft processors has focused on a variety of issues, including the influence of underlying FPGA architecture on the performance of soft processors. The work in [10] exploits the low-level features of FPGA architecture to design a 10-stage processor that can run at the Block RAM maximum of 550 MHz on a Stratix IV device. Other work like [11] utilises the full 36-bit width of a Block RAM to design 36-bit instructions for improved performance. In addition, a number of application-specific soft processors have been proposed, including networking-oriented [11], and floating-point-specific [12], [13] architectures.

Vector soft processors have also been proposed, where a single instruction operates on an array of data. The work in [14] explores a vector processor as an alternative to a custom hardware accelerator. This is further extended in [15] to a system which includes a main processor and a vector co-processor. CUSTARD [16] is a multi-threaded soft processor that uses custom instructions for parallelising applications. A new soft vector architecture is proposed in [17]; the architecture uses a different storage medium – a scratchpad memory in place of the typical register file. This work was further optimised in [18] to improve performance and area.

Other work, such as fSE [19], has used the DSP48E1 primitive as the foundation of the arithmetic unit in a processor. However, the range of supported instructions is limited, to signal processing operations. An evolution of that work in [20] uses similar processors to implement a MIMO sphere decoder.

A key question is how the DSP blocks can be used for general computation, rather than DSP-specific functions. In [21], we showed how a DSP Block could be controlled in a manner allowing it to implement general instructions, however only a basic core, with no real program execution capability was presented.

In this paper, we build on this idea to develop a full processor and instruction-set. We incorporate standard, general processing instructions to enable a wide spectrum of applications, instead of limiting the instructions to only cater to specific domains. The key aim is to build a processor that uses the low-level primitives found in modern devices as efficiently as possible. While at first glance, this architecture specificity may appear limiting, vendors now use the same primitives across whole generations of device families. The DSP48E1 is available in all Xilinx 7-Series FPGAs, so we feel this approach is justifiable for the efficiency gain.

III. Processor Architecture

iDEA is a scalar processor, based on a load-store RISC architecture. It executes 32-bit instructions on 32-bit data with the DSP48E1 primitive serving most purposes in the execution unit. The overall architecture is shown in Fig. 2. Only a single DSP48E1 slice is used, with much of the data processing for arithmetic, logical operations and program control being done within it. We use a RAM32M LUT-based memory primitive for the register file and a RAMB36E1 Block RAM for instruction and data memory.

A. Instruction and Data Memory

The instruction and data memories are built using Block RAM (BRAM) dedicated memory in the FPGA. Different configurations of BRAM mode and latency yield different timing characteristics. Table I shows the effect of design entry and selection of number of pipeline stages on a 512 x 32 memory implemented in a BRAM. It can be observed that both CORE Generator and inference result in similar frequencies for a latency of 2 or 3 clock cycles. At a latency of 3 cycles, an extra register is enabled at the output of the primitive in addition to the output of the core.

TABLE I
Maximum frequency of instruction and data memory in Virtex-6 speed -2.

Description	Maximum Frequency (MHz)	
	Latency 2	Latency 3
CORE Generator	372	539
Inference (Read First Mode)	386	475
Inference (Write First Mode)	378	539
Inference (No Change Mode)	378	539
BRAM Virtex-6 Data Sheet	540	

We implement the instruction and data memory through inference rather than using CORE Generator, as this eases migration and portability. Memory is described behaviourally in Verilog and the synthesis tool automatically infers the required primitive. In order to maximise frequency, the memory must be described such that it infers No Change or Write First mode instead of Read First. This particular mode of the RAMB36E1 can be inferred or "controlled" through a behavioural description as detailed in documentation.

With more complex primitives like the DSP48E1, not all features can be accessed through inference, and the result is often inefficient [22], and hence direct primitive instantiation is desirable as it provides total control over all features.

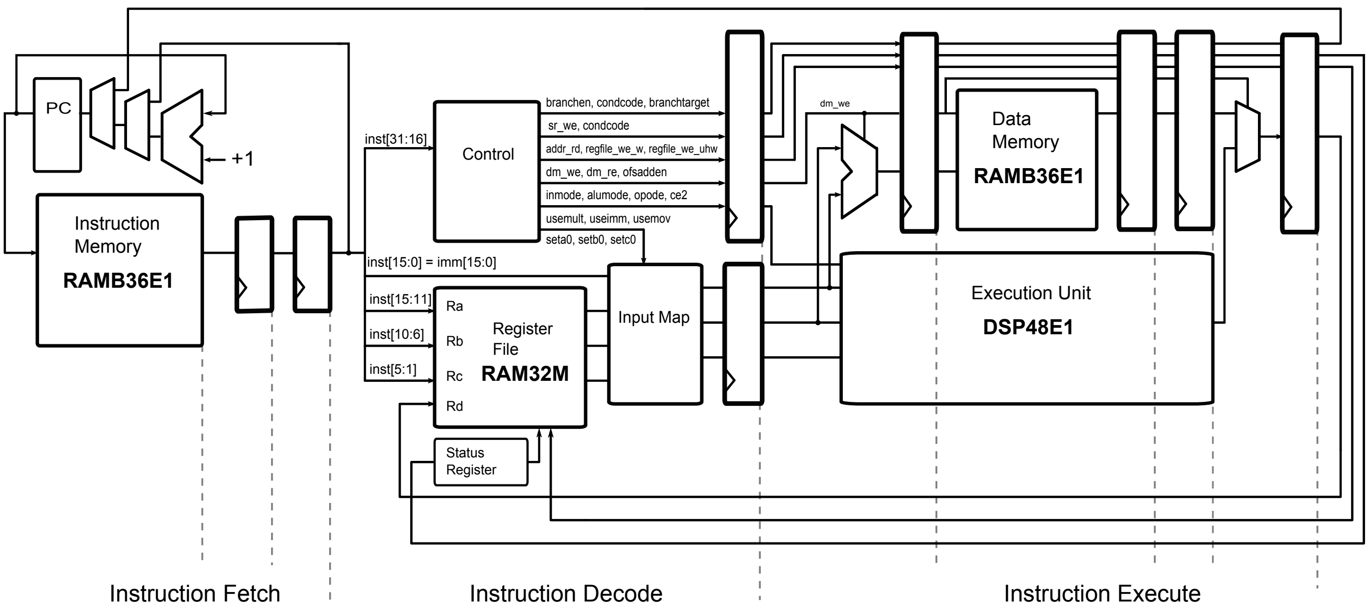

Fig. 2. Processor Block Diagram.

TABLE II
PROCESSOR INSTRUCTIONS

Instruction	Assembly	Operation
Arithmetic/ Logical		
nop	nop	none
add	add rd, ra, rb	rd[31:0] = ra[31:0] + rb[31:0]
	add rd, ra, #imm11	rd[31:0] = ra[31:0] + #imm11[10:0]
sub	sub rd, ra, rb	rd[31:0] = ra[31:0] - rb[31:0]
	sub rd, ra, #imm11	rd[31:0] = ra[31:0] - #imm11
mul	mul rd, rb, rc	rd[31:0] = rb[15:0] x rc[15:0]
mac	mac rd, rb, rc, rp	rd[31:0] = rb[15:0] x rc[15:0] + rp[31:0]
madd	madd rd, ra, rb, rc	rd[31:0] = ra[31:0] + (rb[15:0] x rc[15:0])
msub	msub rd, ra, rb, rc	rd[31:0] = ra[31:0] - (rb[15:0] x rc[15:0])
and	and rd, ra, rb	rd[31:0] = ra[31:0] and rb[31:0]
xor	xor rd, ra, rb	rd[31:0] = ra[31:0] xor rb[31:0]
xnr	xnr rd, ra, rb	rd[31:0] = ra[31:0] xnr rb[31:0]
or	or rd, ra, rb	rd[31:0] = ra[31:0] or rb[31:0]
nor	nor rd, ra, rb	rd[31:0] = ra[31:0] nor rb[31:0]
not	not rd, ra, rb	rd[31:0] = ra[31:0] not rb[31:0]
nand	nand rd, ra, rb	rd[31:0] = ra[31:0] nand rb[31:0]
Data Transfer		
mov	mov rd, ra	rd[31:0] = ra[31:0]
movu	movu rd, #imm16	rd[31:16] = #imm16[15:0]
movl	movl rd, #imm16	rd[15:0] = #imm16[15:0]
ldr	ldr rd, [ra, rb]	rd[31:0] = mem[ra[31:0]+rb[31:0]]
str	str rd, [ra, rb]	mem[ra[31:0]+rb[31:0]] = rd[31:0]
Program Control		
cmp	cmp ra, rb	ra[31:0] - rb[31:0]
	cmp ra, #imm11	ra[31:0] - #imm11[10:0]
b	b #target21	pc = #target21[20:0]
cb{cond}	cb ra, rb, #target11	(ra condition rb) pc = #target11[10:0]

B. Execution Unit

In a load-store architecture, operands are fetched from the register file and fed into the ALU for processing. The results are then written-back into the register file after processing is complete. If a memory write is desired, a separate instruction is needed to store the data from a register into memory. Likewise, a similar separate instruction is required to read from memory into the register file. Other than arithmetic and logical instructions, the execution unit is responsible for processing control instructions as well. However, memory access instructions do not require processing in the execution unit and hence it is bypassed for memory read/write operations.

The execution unit is built using the DSP48E1 primitive as the processing core. Rather than using CORE Generator or inference, the DSP48E1 is instantiated directly, allowing access to all features of the primitive, and providing total control of the configuration of the primitive.

All three pipeline stages of the DSP48E1 are enabled to enable it to run at its maximum frequency. If only a single stage is enabled, the highest frequency achievable is less than half the specified maximum. To further improve performance, a register is added to the output of the primitive. This helps ensure that routing delays out of the primitive do not impact performance. As a result, the total latency of the ALU is 4 clock cycles.

TABLE III
DATA SHEET COMPARISON OF FREQUENCY FOR DSP48E1 IN VIRTEX-6 SPEED -2.

Description	Max Frequency (MHz)
3-stage without pattern detect	540
3-stage with pattern detect	483
1-stage without pattern detect	233
1-stage with pattern detect	219

The DSP48E1 primitive comes is able to support various arithmetic functions, and we aim to utilise as many of these as possible in the design of our execution unit. One particular feature – the pattern detector – adversely impacts the achievable frequency by as much as 10%. Since this feature is optional and is not of critical importance, it has been disabled in iDEA.

Features that are both crucial and relevant to iDEA functionality are:

- 25- $\times$ 18-bit multiplier,
- 48-bit Arithmetic and Logic Unit (ALU) with add/subtract and bit-wise logic operations,
- Ports A and B as separate inputs to the multiplier and concatenated input to the ALU,
- Port C as input to the ALU,
- INMODE dynamic control signal for balanced pipelining when switching between multiply and non-multiply operations,
- OPMODE dynamic control signal for selecting operating modes,
- ALUMODE dynamic control signal for selecting ALU modes,
- optional input, pipeline, and output registers.

C. Other Functional Units

All other functional units aside from the ones mentioned in previous subsections are implemented in LUTs. These include the program counter, branch logic, control unit, status register, input map and an adder for memory address generation. The register file uses the vendor-supplied RAM32M primitive. This is an efficient quad-port (3 read, 1 read/write) memory primitive that is implemented in LUTs. The four ports are required to support two reads and one write in each clock cycle – Block RAMs only provide two ports. Furthermore, as discussed previously, Block RAMs only achieve high frequency when heavily pipelined. To implement a 32×32-bit register file, 16 of these primitives are aggregated, occupying 16 Slices. All the modules are combinational circuits except for the program counter and status register which are synchronous. These modules occupy minimal logic area as the bulk of processor functionality is inside the DSP48E1.

IV. INSTRUCTION SET FORMAT AND DESIGN

The iDEA instruction set is listed in Table II. Though not as extensive as more advanced commercial processors, it is sufficient enough to illustrate the functionality of iDEA in executing arithmetic and data processing applications. We explore this in further detail in Section VI.

A. Input Mapping

The location of input operands is specified in an instruction. Register file locations are addressed using the Ra, Rb and Rc fields while immediate operands – represented by #imm11 and #imm16, are hard-coded. The width of operands is fixed at 32 bits and immediate operands of less than 32 bits are sign-extended to the width of the desired word. The input ports of the DSP48E1 have widths of 30 bits, 18 bits and 48 bits for ports A, B and C respectively. Not only are the widths distinct, they are not byte-multiples. To process 32-bit operands, data must be correctly applied to these inputs.

The execution unit is designed to take two new 32-bit operands, addressed by Ra and Rb, in each clock cycle. In the case of 2-operation, 3-operand instructions, a third 32-bit operand, addressed by Rc is also used. Mapping a 32-bit operand to the DSP48E1 input ports requires it to be split according to the size of the ports that it is mapped to, particularly for ports A and B, which are narrower than 32 bits.

The data flow through the DSP48E1 can be represented as follows:

$$P = C + A : B \tag{1}$$

and

$$P = C + A \times B \tag{2}$$

where P is the output port of DSP48E1. The ”+” operation is performed by the DSP48E1 ALU and can include add, subtract and logical functions. Port D is not currently considered as we have not included pre-adder functionality in iDEA. To effectively map the input operands, we must consider the different internal datapaths for different operations.

Equation 1 is the flow for a 2-operand, single operation instruction. The first operand, Ra, is mapped to port C. Since C is 48 bits wide, the input is sign-extended. The second 32-bit operand, Rb, must be split between ports A and B; the least significant 18 bits are assigned to port B and the most significant 14 bits to port A, sign-extended. This is valid for operations that do not require a multiplier.

Equation 2 represents a 3-operand, 2-operation instruction. Ra is mapped to port C, while Rb is assigned to port A, and Rc to port B. The width of Rb and Rc is limited to 16 bits for multiplication. In the case of multiply only, port C is set to zero. In multiply-add, multiply-sub or multiply-acc, port C carries non-zero data. The DSP48E1 can be dynamically

TABLE IV
PROCESSOR INSTRUCTION FORMAT

Data Processing	31 28	27	26	25 21	20 16	15 11	10 6	5	0
add/sub/logic register	Cond	S	0	Opcode	Rd	Ra	Rb	0 0 0 0 0 0	
add/sub immediate	Cond	S	1	Opcode	Rd	Ra	#imm11		
mul register	Cond	S	0	Opcode	Rd	0 0 0 0 0	Rb	Rc	0
mac/madd/msub register	Cond	S	0	Opcode	Rd	Ra	Rb	Rc	0
Data Transfer									
movu/movl immediate	Cond	0	1	Opcode	Rd	#imm16			
ldr	Cond	0	0	Opcode	Rd	Base Addr.	Offset Addr.	0 0 0 0 0 0	
str	Cond	0	0	Opcode	0 0 0 0 0	Base Addr.	Offset Addr.	Rd	0
Program Control									
cmp register	Cond	S	0	Opcode	0 0 0 0 0	Ra	Rb	0 0 0 0 0 0	
cmp immediate	Cond	S	1	Opcode	0 0 0 0 0	Ra	#imm11		
b	Always	0	0	Opcode	#target21				
cb	Cond	S	0	Opcode	Rd	Ra	#target11		

switched between operations defined by 1 and 2 through the INMODE, OPMODE and ALUMODE control signals. Table V illustrates the port mappings for some common instructions.

V. Designing for Performance

A. *Functional Advantage of DSP48E1*

The multiplier in the DSP48E1 enables multiplication and shift to be performed. With the ALU that follows it, two consecutive arithmetic operations on the same set of data can be performed, including multiply-add and multiply-sub. This composite functionality is possible without any extra hardware.

The DSP48E1 primitive provides adequate circuitry for processing 32-bit data. In fact, the widest data supported is 48 bits. A single DSP48E1 primitive alone is sufficient to construct a 32-bit processor with uniform input and output of 32 bits. Although the size of ports A and B, at 30 and 18 bits respectively, are less than 32 bits, these two ports can be concatenated, $\{A{:}B\}$, to produce a combined data width of 48 bits.

The DSP48E1 primitive produces an output of 48 bits through port P, regardless of the type of arithmetic operation. Since the effective data size is 32 bits, only the least significant 32 bits are used in iDEA. It is important to note, however, that the width of the multiplier inside the DSP48E1 is only 25×18 bits. To fully implement a 32×32 multiplier, three DSP48E1 primitives can be cascaded together, but this triples the resource requirement for the benefit of only a single instruction.

Hence, we restrict multiplication to 16×16 bits, producing a 32-bit result, which still fits the iDEA specification. A wider multiplication would not be beneficial, since the result would have to be truncated to fit the 32-bit data format. For operations that involve the multiplier, data inputs are limited to 16 bits, while for other operations they are 32 bits.

TABLE VI
EFFECT OF PIPELINE STAGES ON FREQUENCY.

Pipeline Stages	Inst. Execute	Inst. Fetch	Frequency (MHz)
9-stage	4	3	407
8-stage	3	3	335
8-stage	4	2	278
7-stage	3	2	297
7-stage	4	1	209
6-stage	3	1	214

B. *Frequency and Pipeline Length*

Operating frequency is a commonly used measure of processor performance as it affects how fast instructions are executed. Increasing the number of pipeline stages in a processor often leads to improved frequency, but at the cost of longer latency and higher branch penalty. Higher branch penalties result in a higher number of wasted instruction cycles when a branch occurs. Hence, a balanced trade-off between frequency and the number of pipeline stages is required to achieve effective throughput.

Table VI lists experimental results showing different pipeline configurations of iDEA and the corresponding frequency achieved. The instruction cycle stages are reduced by removing pipeline stages from the memory and execution unit, corresponding to instruction fetch and instruction execute respectively. The instruction pipeline begins from instruction fetch, instruction decode, instruction execute and lastly write-back to the register file. The two additional stages on top of the listed instruction fetch and execute stages in the table are instruction decode and write-back. As expected, the longest pipeline yields the highest frequency of 407.5 MHz for a Virtex-6 XC6VLX240T-2 device.

Based on these experiments, and targeting speed, we have designed iDEA with a 9-stage pipeline. These are divided as follows: 3 stages for instruction fetch, 1 stage for instruction decode, 4 stages for execute and 1 stage for write-back. Not all

TABLE V
PORT MAPPING FOR DIFFERENT ARITHMETIC FUNCTIONS

Inst.	Assembly	Operation	Port A (30 bits)	Port B (18 bits)	Port C (48 bits)
add	add Rd, Ra, Rb	C + A:B	16{Rb[31]}, Rb[31:18]	Rb[17:0]	16{Ra[31]}, Ra[31:0]
	add Rd, Ra, #imm11	C + A:B	30{1'b0}	7{imm[10]}, imm[10:0]	16{Ra[31]}, Ra[31:0]
sub	sub Rd, Ra, Rb	C - A:B	16{Rb[31]}, Rb[31:18]	Rb[17:0]	16{Ra[31]}, Ra[31:0]
mul	mul Rd, Rb, Rc	C + A x B	15{Rb[15]}, Rb[15:0]	2{Rc[15]}, Rc[15:0]	48{1'b0}
madd	madd Rd, Ra, Rb, Rc	C + A x B	15{Rb[15]}, Rb[15:0]	2{Rb[15]}, Rc[15:0]	16{Ra[31]}, Ra[31:0]
movl	movl Rd, #imm16	C + A x B	30{1'b0}	18{1'b0}	32{1'b0}, imm[15:0]

instructions require the full 9 stages, instructions like branch and data memory write execute in fewer cycles. For branch, the execution unit is bypassed and thus the pipeline is just 4 stages long.

While data memory read takes 9 stages, data memory write is shorter at 6 stages. The data write operation completes after data is written to memory while data read has to retrieve data from memory and perform an additional write-back to the register file. For both read and write, the effective address of a data location is calculated in the 5th stage and the resulting address is fed to the memory input in the next stage. An alternative to using a dedicated effective address adder is to compute the address using the execution unit. However, in such a design, memory access can only happen after the execution unit computes the effective address, increasing the pipeline length to 12 stages.

C. Limitations to Performance

An increased number of pipeline stages results in increased frequency, but also contributes to an increase in the number of clock cycles taken to complete the processing of an instruction. A processor with a 9-stage pipeline with 1 clock cycle latency for each stage requires a total of 9 clock cycles to complete the execution of an instruction.

If instructions are fetched in successive clock cycles, with no change in program flow, the effect of a long pipeline is not prevalent. However, when the sequence of instruction is altered, as in the case of branching, the penalty or loss of useful instruction cycles is more severe. As the branch decision is determined at the end of the pipeline stage, the penalty incurred is 8 clock cycles.

While branching cannot be totally eliminated, we can introduce methods to minimize the recurrence of branching through conditional execution. Although branch prediction techniques improve the flow of instructions by anticipating the possibility of a branch, they increase the complexity of the processor hardware. By reducing the recurrence of branching, we reduce the penalties incurred. Addressing the penalties caused by branching is important in improving the throughput of the processor.

VI. IMPLEMENTATION RESULTS

In this section, we analyse the area and performance of iDEA, and provide an at-a-glance comparison with MicroBlaze, a commercial soft-core processsor from Xilinx. We look at operating frequency, resource consumption, instruction count, and latency. All experiments are performed on a Virtex-6 XC6VLX240T-2 device as present on the Xilinx ML605 development board.

Three applications that showcase general purpose computation are selected to demonstrate the functionality of our processor – Fibonacci, FIR filter and Median filter. These applications are data processing applications involving data read-write in memory and multiplication. For MicroBlaze, the C applications are compiled using the C compiler from the Xilinx Software Development Kit 13.2 (SDK), *mb-gcc*. For iDEA, as we are still developing a compiler toolchain, we manually translate the source into corresponding assembly code.

A. Configuration of Processor and Impact in Area

To ensure a fair comparison of the two processors, we configure the smallest possible MicroBlaze while keeping all the basic functionality necessary to run the applications. Extra peripherals and features of MicroBlaze that are not available in iDEA, such as cache, memory management and debug module, are removed.

The multiplier is enabled in MicroBlaze, set to the minimum configurable width of 32 bits. Other hardware like barrel shifter, floating point unit, integer divider and pattern comparator are disabled, significantly reducing the size of MicroBlaze. The *mb-gcc* compiler automatically ensures it does not generate instructions for disabled features.

TABLE VII
COMPARISON OF RESOURCE CONSUMPTION OF BOTH PROCESSOR SYSTEMS OVER TOTAL AVAILABLE RESOURCE IN VIRTEX-6 -2.

Resource	iDEA	MicroBlaze	Available
Slice Registers	404	514	301,440
Slice LUTs	335	878	150,720
RAMB36E1	2	1	416
DSP48E1	1	3	768
Frequency (PAR)	407 MHz	210 MHz	–

Table VII shows the post-place-and-route implementation results for both processors. For iDEA, the implementation is performed using Xilinx ISE 13.2 while MicroBlaze is implemented using Xilinx Platform Studio (XPS) 13.2, as illustrated in 3. Both implementations includes memory subsystems and the processor core. A total of 4KB is allocated for instruction and data memory for each of the processors.

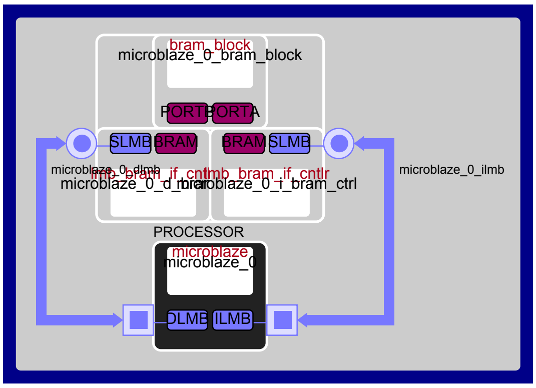

Fig. 3. MicroBlaze Block Diagram with 4KB Instruction and Data Memory.

The MicroBlaze design uses almost twice as many slice registers and LUTs as iDEA. Unlike MicroBlaze, iDEA does not have additional fixed features such as special purpose registers and an instruction buffer, which contribute to the higher logic count. Although the memory size of both processors is identical at 4KB, which can fit into a single RAMB36E1, the iDEA memory subsystem is mapped into two RAM36BE1s. This is because the instruction memory is read-only while the data memory is read/write, and the difference in read/write mode causes them to be mapped into two separate primitives. We could double the memory sizes without using further resources. MicroBlaze supports a wider multiplication width of 32×32, resulting in the use of 3 DSP48E1 slices.

In order to confirm the portability of iDEA, we also implemented the design on the next-generation Artix-7, Kintex-7, and Virtex-7 families. The results, shown in Table VIII are mostly in line with the Virtex-6 results, with the low-cost Artix-7 exhibiting reduced frequency. These results may improve slightly as tools mature, as is generally the case for new devices.

TABLE VIII
IDEA IN ARTIX-7, KINTEX-7 AND VIRTEX-7.

Resource	Virtex-6	Artix-7	Kintex-7	Virtex-7
Slice Registers	404	392	411	411
Slice LUTs	335	295	301	306
RAMB36E1	2	2	2	2
DSP48E1	1	1	1	1
PAR Freq (MHz)	407	289	408	392

B. *Impact on Instruction Count*

Total instruction count and latency are obtained by testbench profiling using an HDL simulator, as this provides a standard, consistent performance measure for both processors. An instruction set simulator is another option for profiling, however the current Xilinx toolset does not provide such a simulator for MicroBlaze. Additionally, HDL simulation is more accurate than an instruction set simulator.

The testbench and simulation files for MicroBlaze are automatically generated by XPS. In the testbench, we added a module that tracks the instruction count in every clock cycle. The tracker is started at the beginning of a computation and terminates once it is complete. With every valid instruction issued, the instruction counter is incremented. The total number of clock cycles is determined from when the tracker starts until it terminates. The start and end signal is obtained from the instruction opcode in the disassembly file.

The software application is written in C and compiled by *mb-gcc* into an .elf executable. This can be viewed as a disassembly file. From this, we locate when a computation starts and ends and the corresponding program counter address. Once the tracker module encounters these addresses, it starts and stops the count tracking accordingly.

For iDEA, the start and stop cues for tracking are determined from program counter addresses. Only instructions that are useful and valid are included in the instruction count tracking. Pipeline pads, such as *NOPs* are not taken into account. Although *NOPs* are not part of the total instruction count, the effect of NOPs will be evident in the total clock cycles taken. We also added constructs to print out the values of the register file and data memory for verification.

We set the optimisation level of *mb-gcc* to -o0, which is the default. In -o0, no optimisations are performed and the C code is compiled in the most straightforward manner possible. Since we do not yet have an iDEA compiler, we write assembly manually for iDEA, without optimising. This mirrors the -o0 option in *mb-gcc*, ensuring the comparison is fairer.

TABLE IX
COMPARISON OF INSTRUCTION COUNT OF BOTH PROCESSOR SYSTEMS IN VIRTEX-6 -2.

Application	Inst. Count	Clock Cycles	Exe. Time
iDEA			
Fibonacci	256	1,414	3.47 us
FIR	8,609	51,121	125.25 us
Median	46,704	240,516	589.26 us
Microblaze			
Fibonacci	756	1,360	6.46 us
FIR	39,406	63,410	306.90 us
Median	125,259	202,509	980.67 us

Table IX shows the total instruction count and latency of both processors for the three test applications, Fibonacci, FIR and Median filter. iDEA has a generous number of registers to support data operations and minimal complexity when executing instructions. However, due to the long pipeline and absence of branch prediction and data forwarding schemes, it requires *NOP* fillers between affected instructions. The *NOP* fillers significantly increase the number of clock cycles and overall latency of a program. MicroBlaze requires a higher number of instructions to execute the same program for -o1 compiled code. An analysis of the disassembly file shows that MicroBlaze only allocates a limited number of registers for program computation. For Fibonacci and FIR, MicroBlaze uses only three registers while it uses six for Median. As a result, the processor has to re-use the same registers for different operations, repeatedly loading data for processing, thereby increasing the instruction count.

MicroBlaze has the added advantage of an optimising compiler. If we increase the MicroBlaze optimisation level to -o1, the total clock cycles taken are 401 (1.9us), 29,295 (141.9us) and 68,433 (331.4us) for Fibonacci, FIR and Median respectively. Based on these figures and Table IX, the performance of our processor falls somewhere between an -o0 and -o1 MicroBlaze compilation. However, this comparison is more to show functional equivalence than for a direct performance comparison. To do so, we would require a custom compiler for iDEA – something we are working on at present. It is important to restate that for these results, the iDEA assembly code is not hand-optimised.

Not all three MicroBlaze applications fit into the 4KB memory size built in the original processor system in Table VII. To accommodate larger applications, such as FIR and Median, an increase of memory size to 8KB is required. However, increasing the MicroBlaze memory results in a slight reduction in the MicroBlaze clock frequency, from 210 MHz to 206 MHz. As for iDEA, 4KB is sufficient for all three applications. Execution times are given in Table IX and show that iDEA is not only lightweight, but looks promising once a compiler is available.

VII. Conclusion

This paper introduces iDEA, an instruction set-based, soft processor built with a DSP48E1 primitive as the execution core. We harness the strengths of the DSP48E1 primitive by manipulating its functionality to suit the architecture of a load-store processor. The DSP48E1 primitive is designed for signal processing implementations, but we show that it is capable of supporting all the required arithmetic functionality for a basic processor. As iDEA is designed to occupy minimal area, the logic is kept as simple as possible. By limiting the addition of hardware modules such as branch prediction, we are able to minimise control complexity.

The processor has a basic, yet comprehensive enough, instruction set for general purpose applications. Using three C applications – Fibonacci, FIR and Median filter, we show that it is on-par with a minimised MicroBlaze soft processor. It occupies about half as many slice LUTs and registers as MicroBlaze while achieving about twice the frequency. We aim to focus now on reducing the need for *NOP* fillers. These cause significant latency overhead and decrease code density, for what is otherwise an efficient, high speed processor. We are working on developing a compiler that will allow some of these limitations to be overcome at compilation.

We have presented a DSP48E1-based processor that is minimal, yet comprehensive enough to be able to run general purpose processing tasks, rather than being tailored to specific application areas. The processor can be implemented across the next generation of Xilinx FPGAs, achieving comparable performance in all cases.

References

[1] *UG369: Virtex-6 FPGA DSP48E1 Slice User Guide*, Xilinx Inc., 2011.

[2] *UG081: MicroBlaze Processor Reference Guide*, Xilinx Inc., 2011.

[3] *Nios II Processor Design*, Altera Corpration, 2011.

[4] *Cortex-M1 Processor*, ARM Ltd., 2011. [Online]. Available: http://www.arm.com/products/processors/cortex-m/cortex-m1.php

[5] *LatticeMico32 Processor Reference Manual*, Lattice Semiconductor Corp., 2009.

[6] *GRLIB IP Library User's Manual*, Lattice Semiconductor Corp., 2012.

[7] F. Plavec, B. Fort, Z. Vranesic, and S. Brown, "Experiences with soft-core processor design," in *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS)*, Apr. 2005, p. 167b.

[8] T. Kranenburg and R. van Leuken, "MB-LITE: A robust, light-weight soft-core implementation of the microblaze architecture," in *Proceedings of Design, Automation Test in Europe Conference*, Mar. 2010, pp. 997–1000.

[9] L. Barthe, L. Cargnini, P. Benoit, and L. Torres, "The SecretBlaze: A configurable and cost-effective open-source soft-core processor," in *International Symposium on Parallel and Distributed Processing (IPDPS) Workshops*, May. 2011, pp. 310–313.

[10] C. LaForest, J. Steffan, and J. Gregory, "OCTAVO: an FPGA-centric processor family," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, Feb. 2012, pp. 219–228.

[11] P. Buciak and J. Botwicz, "Lightweight multi-threaded network processor core in FPGA," in *Design and Diagnostics of Electronic Circuits and Systems*, Apr. 2007, pp. 1–5.

[12] J. Kathiara and M. Leeser, "An Autonomous Vector/Scalar Floating Point Coprocessor for FPGAs," in *Proceedings of International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, May. 2011, pp. 33–36.

[13] Y. Lei, Y. Dou, J. Zhou, and S. Wang, "VPFPAP: A special-purpose VLIW processor for variable-precision floating-point arithmetic," in *Proceedings of International Conference on Field Programmable Logic and Applications (FPL)*, Sept. 2011, pp. 252–257.

[14] J. Yu, G. Lemieux, and C. Eagleston, "Vector processing as a soft-core CPU accelerator," in *Proceedings of ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, Feb. 2008, pp. 222–232.

[15] P. Yiannacouras, J. Steffan, and J. Rose, "VESPA: Portable, scalable, and flexible FPGA-based vector processors," in *Proceedings of International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2008, pp. 61–70.

[16] R. Dimond, O. Mencer, and W. Luk, "CUSTARD - a customisable threaded FPGA soft processor and tools," in *Proceedings of International Conference on Field Programmable Logic and Applications (FPL)*, 2005.

[17] C. Chou, A. Severance, A. Brant, Z. Liu, S. Sant, and G. Lemieux, "VEGAS: Soft vector processor with scratchpad memory," in *Proceedings of ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, 2011, pp. 15–24.

[18] A. Severance and G. Lemieux, "VENICE: A compact vector processor for FPGA applications," in *Workshop on the Intersections of Computer Architecture and Reconfigurable Logic*, 2012.

[19] M. Milford and J. McAllister, "An ultra-fine processor for FPGA DSP chip multiprocessors," in *Conference Record of the Asilomar Conference on Signals, Systems and Computers*, 2009, pp. 226 –230.

[20] X. Chu and J. McAllister, "FPGA based soft-core SIMD processing: A MIMO-OFDM fixed-complexity sphere decoder case study," in *Proceedings of International Conference on Field Programmable Technology (FPT)*, 2010, pp. 479–484.

[21] H. Y. Cheah, S. A. Fahmy, D. L. Maskell, and C. Kulkarni, "A lean FPGA soft processor built using a DSP block," in *Proceedings of ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, 2012, pp. 237–240.

[22] B. Ronak and S. A. Fahmy, "Evaluating the efficiency of DSP block synthesis inference from flow graphs," in *Proceedings of International Conference on Field Programmable Logic and Applications (FPL)*, 2012.