The iDEA DSP Block-Based Soft Processor for FPGAs

HUI YAN CHEAH, FREDRIK BROSSER, SUHAIB A. FAHMY,
and DOUGLAS L. MASKELL, Nanyang Technological University

DSP blocks in modern FPGAs can be used for a wide range of arithmetic functions, offering increased performance while saving logic resources for other uses. They have evolved to better support a plethora of signal processing tasks, meaning that in other application domains they may be underutilised. The DSP48E1 primitives in new Xilinx devices support dynamic programmability that can help extend their usefulness; the specific function of a DSP block can be modified on a cycle-by-cycle basis. However, the standard synthesis flow does not leverage this flexibility in the vast majority of cases. The lean DSP Extension Architecture (iDEA) presented in this article builds around the dynamic programmability of a single DSP48E1 primitive, with minimal additional logic to create a general-purpose processor supporting a full instruction-set architecture. The result is a very compact, fast processor that can execute a full gamut of general machine instructions. We show a number of simple applications compiled using an MIPS compiler and translated to the iDEA instruction set, comparing with a Xilinx MicroBlaze to show estimated performance figures. Being based on the DSP48E1, this processor can be deployed across next-generation Xilinx Artix-7, Kintex-7, Virtex-7, and Zynq families.

Categories and Subject Descriptors: B.6.1 [**Logic Design**]: Design Styles

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Field programmable gate arrays, reconfigurable computing, soft processors

ACM Reference Format:
Hui Yan Cheah, Fredrik Brosser, Suhaib A. Fahmy, and Douglas L. Maskell. 2014. The iDEA DSP block-based soft processor for FPGAs. ACM Trans. Reconfig. Technol. Syst. 7, 3, Article 19 (August 2014), 23 pages.
DOI: http://dx.doi.org/10.1145/2629443

1. INTRODUCTION

The flexibility of field programmable gate arrays (FPGAs) has been their key feature 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. Following the addition of memory blocks, hard multipliers were added to speed up common signal processing tasks. These later evolved into multiply-accumulate blocks, often used 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 [Xilinx 2011b] is found on Xilinx Virtex-6, Artix-7, Kintex-7, and Virtex-7 FPGAs, as well as the Zynq-7000 hybrid ARM-FPGA platform. It boasts increased capabilities over previous generations of DSP blocks and is highly

Authors' addresses: H. Y. Cheah (corresponding author), F. Brosser, S. A. Fahmy, and D. L. Maskell, School of Computer Engineering, Nanyang Technological University, Block N4, Nanyang Avenue, Singapore 639798; email: hycheah1@e.ntu.edu.sg

DOI: http://dx.doi.org/10.1145/2629443

customisable. One key feature of this primitive that has motivated and enabled the work presented in this article is its dynamic programmability. The DSP48E1 datapath can be customised at runtime to compute a number of different functions, for example, it is possible to use just the multiplier or to combine it with the accumulator or postadder, as well as to include the preadder. The function to compute can be selected at runtime on a cycle-by-cycle basis by modifying control signals. This enables the same primitive instance to be used for different functions if a controller is added to appropriately set the configuration. The DSP48E1 has a wider datapath, operates at higher frequency, accepts more inputs, and is more flexible than previous DSP blocks. While these more capable DSP blocks were previously only found in high-end FPGAs, the decision to include them across all 7-series devices means that designs tailored to them are still portable within the same vendor's FPGAs, making architecture-targeted designs more feasible.

The DSP48E1 primitive is composed of a preadder, multiplier, and adder/subtracter/logic unit (ALU). These functions are all frequently required in typical DSP algorithms like finite impulse-response filters. Though primarily intended for DSP applications, the DSP48E1 can also be used in any application that requires high-speed arithmetic, and the synthesis tools will infer them whenever wider arithmetic operations are used. DSP blocks are more power efficient, operate at a higher frequency, and consume less area than the equivalent operations implemented using the logic fabric. As such, they are heavily used in the pipelined datapaths of computationally intensive applications [de Dinechin and Pasca 2011; Xu et al. 2014]. However, we have found that DSP block inference by the synthesis tools can be suboptimal [Ronak and Fahmy 2012] and the dynamic programmability feature is not mapped except in very restricted cases. As the number of DSP blocks on modern devices increases, finding ways to use them efficiently outside of their core application domain becomes necessary.

Today, FPGAs are often used to implement full systems rather than just accelerators, hence processors have become a more important feature of many FPGA designs. Previous attempts at introducing Power PC hard processors in the Virtex II Pro [Xilinx 2011c] and Virtex 4 FX [Xilinx 2010] were not entirely successful as a particular fixed processor may not suit the wide range of applications that might be implemented on an FPGA. Hence, "soft" processors built using logic resources have continued to dominate. They have the advantage of flexibility; a designer can choose which features are needed for their particular application. One issue with many soft processor designs is that they pay little attention to the underlying architecture and hence exhibit poor performance. While vendors do offer their own optimised designs, we have yet to see a processor design built from the architecture up.

In this article, we present the design of a lightweight extension architecture built around the DSP48E1 primitive, resulting in a lean, comprehensive, general-purpose processor called iDEA. The design of iDEA is very much architecture focused, in order to offer maximum performance while being as lean as possible. This work is a proof-of-concept demonstration of how dynamic programmability of the DSP block means the resources can be used beyond the current limits of synthesis inference for arithmetic. This article details the architecture and instruction set, the design process, and presents detailed performance results and further discussion beyond those presented in Cheah et al. [2012a]. 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 serve as the basis for a massively parallel architecture on reconfigurable fabric and enable research on how best to arrange and program such systems.

iDEA is being released to the wider community in the hope that it will spur further research [Cheah 2013].

The remainder of this article is organised as follows. Section 2 covers related work, while Section 3 discusses the functionality of the DSP48E1. Section 4 presents the iDEA architecture and Section 5 details the instruction set. Section 6 discusses hardware implementation results. Subsequently, Section 7 describes how we have compiled and simulated small benchmark applications for iDEA and the resulting performance. Finally, Section 8 concludes the article and presents our future work.

2. RELATED WORK

While pure algorithm acceleration is often done through the design of custom parallel architectures, many supporting tasks within a complete FPGA-based system are more suited to software implementation. Hence, soft processor cores have long been used and now, more often than not, FPGA-based systems incorporate some sort of processor. A processor enhances the flexibility of hardware by introducing some level of software programmability to the system.

Although a hard processor can offer better performance than an equivalent soft processor, they are inflexible and cannot be tailored to suit the needs of different applications. Furthermore, their fixed position in the fabric can complicate floorplanning, and a large amount of supporting infrastructure is required in logic. If such a processor is not used or underutilised, it represents a significant waste of silicon resources.

Meanwhile, soft processors have been widely adopted in many applications due to their relative simplicity, customisability, and good tool-chain support. Soft processors can be tailored to the specific needs of an application and, since they are implemented entirely in the logic fabric, additional features can be easily added or removed at design time. Commercial soft processors include the Xilinx MicroBlaze [Xilinx 2011a], Altera Nios II [Altera 2011], ARM Cortex-M1 [ARM 2011], and LatticeMico32 [Lattice Semiconductor 2009], in addition to the open-source Leon3 [Aeroflex Gaisler 2012].

Generally, FPGAs are used when there is a desire to accelerate a complex algorithm. As such, a custom datapath is necessary, consuming a significant portion of the design effort. Soft processors generally find their use in the auxiliary functions of the system, such as managing noncritical data movement, providing a configuration interface [Vipin and Fahmy 2012], or even implementing the cognitive functions in an adaptive system [Lotze et al. 2009]. The flexibility of their programming lends ease of use to the system without adversely impacting the custom datapath.

FPGA vendors provide processors that are generally restricted to their own platforms, limiting device choice when such cores are used in a design. Some effort has been put into porting these cores to alternative architectures [Plavec et al. 2005; Kranenburg and van Leuken 2010; Barthe et al. 2011]. However, the more generalised a core, the less closely it fits the low-level target architecture and hence the less efficient its implementation in terms of area 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 the underlying FPGA architecture on their performance. The work in LaForest et al. [2012] exploits the low-level features of the FPGA architecture to design a multithreaded 10-stage processor that can run at the block RAM maximum of 550 MHz on a Stratix IV device. No programming model was discussed in that work. The work in Buciak and Botwicz [2007] utilises the full 36-bitwidth 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 [Buciak and Botwicz 2007] and floating-point-specific [Kathiara and Leeser 2011; Lei et al. 2011] architectures.

Vector soft processors have also been proposed, where a single instruction operates on an array of data. The work in Yu et al. [2008] explores a vector processor as an

alternative to a custom hardware accelerator, further extended in Yiannacouras et al. [2008] to a system that includes a main processor and a vector coprocessor. CUSTARD [Dimond et al. 2005] is a multithreaded soft processor that uses custom instructions for parallelising applications. A new soft vector architecture is proposed in Chou et al. [2011]; the architecture uses a different storage medium, namely a scratchpad memory, in place of the typical register file. This work was further optimised in Severance and Lemieux [2012] to improve performance and area. fSE [Milford and McAllister 2009] uses the dynamic programmability of the DSP48E1 primitive and extends logic around it to support a set of instructions for signal processing operations. They demonstrate a MIMO sphere decoder using these processors in Chu and McAllister [2010]. That work, however, is restricted to the instructions required in a specific domain and generalisation is not discussed.

The motivation behind this work lies in the dynamic programmability of modern DSP blocks, as well as the advantages we believe this offers in terms of using these resources beyond the original intent of DSP applications and basic arithmetic through the standard synthesis flow. As Xilinx now includes these primitives across all their products from low-cost to high-end, it becomes imperative to find more general ways to exploit them and an architecture-specific design that can still find general use while remaining somewhat portable.

In Cheah et al. [2012b], we explored how a DSP block could be controlled to allow it to implement general instructions. In Brosser et al. [2013], we applied the same principle to basic floating-point operations and in Cheah et al. [2012a], we extended this idea and presented the first discussion of the iDEA soft processor and some preliminary results. iDEA incorporates standard, general processing instructions to enable a wide spectrum of applications, instead of limiting the instructions to only cater to specific domains. In this article we explore the motivation further, discuss design optimisations, and present a much wider range of results including a detailed motivation for building a custom compiler.

3. THE DSP48E1 PRIMITIVE

3.1. Evolution of the DSP Block

The Xilinx DSP48E1 primitive is an embedded hard core present in the Xilinx Virtex-6 and new 7-series FPGAs. Designed for high-speed digital signal processing computation, it is composed of a multiplier, preadder, and arithmetic logic unit (ALU), along with various registers and multiplexers. A host of configuration inputs allow the functionality of the primitive to be manipulated at both runtime and compile time. It can be configured to support various operations like multiply-add, add-multiply-add, pattern matching, and barrel shifting, among others [Xilinx 2011b]. As DSP blocks evolved from simple multiplier blocks to include extra functions, control inputs were added to allow functionality to be modified.

Over time, numerous enhancements have been made to the architecture to improve speed, frequency, logic functionality, and controllability. Table I presents a comparison of previous iterations of the DSP block from Xilinx. The input wordlengths have increased, along with more supported functions, and the maximum frequency has risen. ALU-type functionality has also been incorporated, allowing logical operations. Further functions such as pattern detection logic, cascade paths, and SIMD mode are all incorporated into the latest DSP48E1 primitive.

This work has been motivated by a recognition of the possibilities afforded by the dynamic programmability of the Xilinx DSP block. Other vendors' blocks are designed only to support basic multiply and add operations, in a few combinations, in contrast to the many possible configurations for a DSP48E1. Furthermore, other DSP blocks

Table I. Comparison of Multiplier and DSP Primitives in Xilinx Devices

Primitive	Device	Max Freq (MHz)	Mult	Port Width A	B	C	D	ALU Function	Control Signals	Pre-add	Patt. Detect
MULT18X18S	Virtex-2	105	18×18	18	18	—	—	—	—	No	No
DSP48A	Spartan-3A	250	18×18	18	18	48	18	add, sub	opmode	Yes	No
DSP48A1	Spartan-6	250	18×18	18	18	48	18	add, sub	opmode	Yes	No
DSP48	Virtex-4	500	18×18	18	18	48	—	add, sub	opmode	No	No
DSP48E	Virtex-5	550	25×18	30	18	48	—	add, sub, logic	opmode, alumode	No	Yes
DSP48E1	Virtex-6 Artix-7 Kintex-7 Virtex-7	600 628 741 741	25 × 18	30	18	48	25	add, sub, logic	opmode, alumode, inmode	Yes	Yes

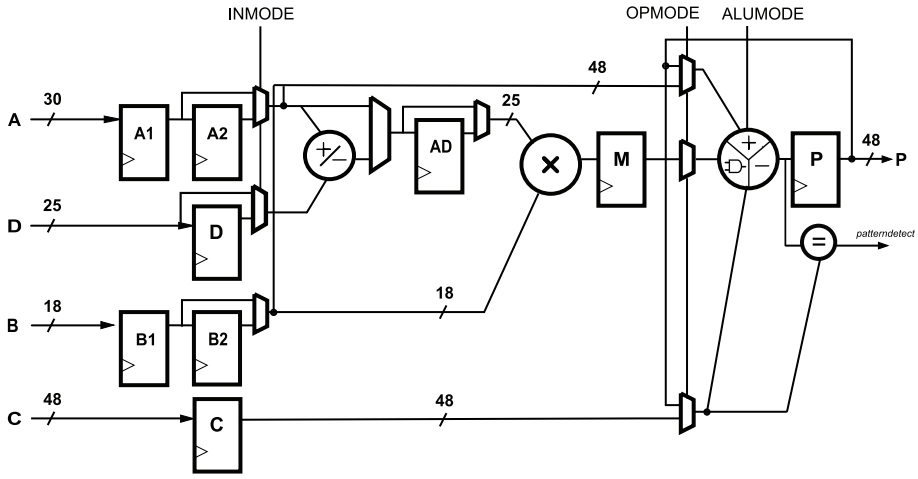

Fig. 1. Architecture of the DSP48E1.

have their configuration fixed at design time, meaning their functionality cannot be dynamically modified. Hence, the basic premise behind this work is not currently applicable to other vendor's DSP blocks at present.

3.2. Xilinx DSP48E1 Primitive

Figure 1 shows a representation of the DSP48E1 slice with three ports, A, B, and C, that supply inputs to the multiplier and add/sub/logic block, as well as port D that allows a value to be added to the A input prior to multiplication. We later discuss why the D input is not used in iDEA.

In general, the datapath of all arithmetic operations can be categorised into multiplier and nonmultiplier paths. The data inputs of a multiplier path are fed to a multiplier unit before being processed by the ALU unit. Except for multiply and shift, other operations bypass the multiplier and the data travels through the nonmultiplier path. As shown in Figures 2 and 3, the ALU unit is utilised in both multiplier and nonmultiplier paths.

The functionality of DSP48E1 is controlled by a combination of dynamic control signals and static parameter attributes. Dynamic control signals allow it to run in different configuration modes in each clock cycle. For instance, the ALU operation can

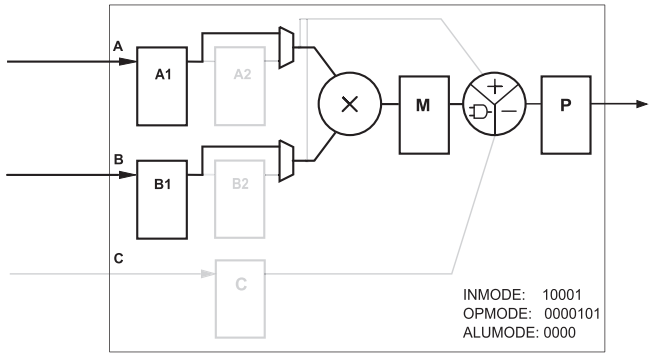

Fig. 2. Datapath for multiplication. Path C is not used.

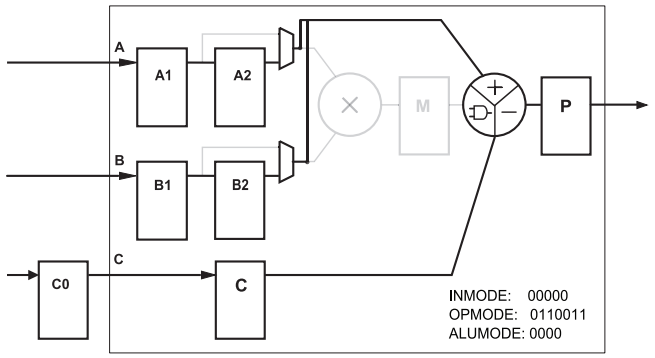

Fig. 3. Datapath for addition. The extra register C0 balance the pipeline.

be changed by modifying ALUMODE, the ALU input selection by modifying OPMODE, and the preadder and input pipeline by modifying INMODE. Other static parameter attributes are specified and programmed at compile time and cannot be modified at runtime. Table II summarises the functionality of the dynamic control signals.

3.3. Executing Instructions on the DSP48E1

Four different datapath configurations are explained here to demonstrate the flexibility and capability of the DSP48E1 in its various operating modes.

3.3.1. Multiplication. In multiplication, input data is passed through ports A and B as the first and second operand, respectively, with port C unused. Four register stages, A1, B1, M, and P, are enabled along the multiplication datapath to achieve maximum frequency. A simplified version of the datapath is shown in Figure 2. The number of registers in the multiplier input path is controlled by the parameters AREG and BREG that are fixed at compile time. Inputs are registered by A1 and B1 prior to entering the multiplier, while final results emerge at register P two cycles later. In other multiplication-based operations, the P output is fed back to the ALU for accumulation purposes.

3.3.2. Addition. When not using a multiplier, inputs A, B, and C are fed straight to the ALU unit. To compensate for bypassing of the M register, registers A2 and B2 are enabled to keep the pipeline length the same. When the multiplier is bypassed, the first operand is split over ports A and B, while port C carries the second operand. In

Table II. DSP48E1 Dynamic Control Signals

Signal	Description
ALUMODE	Selects ALU arithmetic function
OPMODE	Selects input values to ALU
INMODE	Selects preadder functionality and registers in path A, B and D
CARRYINSEL	Selects input carry source
CEA1, CEA2	Enable register A1, A2
CEB1, CEB2	Enable register B1, B2
CEC	Enable register C
CEAD	Enable register AD
CED	Enable register D
CEM	Enable register M
CEP	Enable register P

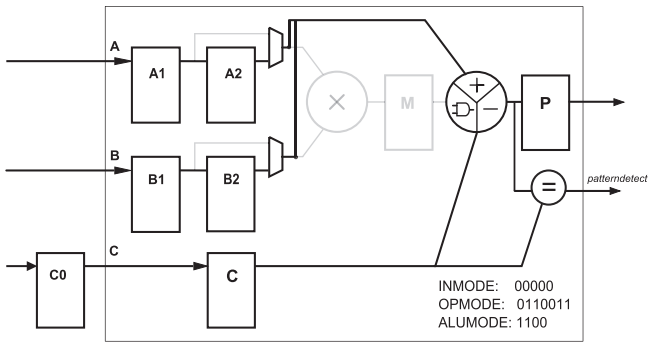

Fig. 4. Datapath for compare. Pattern detect logic compares the value of C and P.

order to match the pipeline stages of paths A and B, an extra register is placed in the logic fabric in addition to the internal C register, as shown in Figure 3.

3.3.3. Compare. The compare operation can be configured using a nonmultiplier datapath with additional pattern detect logic enabled. The pattern detect logic compares the value in register P against a pattern input. If the pattern matches, the *patterndetect* output signal is set to high. The pattern field can be obtained from two sources, namely a dynamic source from input C or a static parameter field.

Figure 4 shows the datapath of a compare operation. Path A:B carries the first operand while path C carries the second operand that is the value to be compared against. The comparison is made between P and C. Prior to reaching P, all input data is processed by the ALU so we must ensure the value carried by A:B remains unchanged through the ALU. Logical AND is applied between A:B and C through the ALU. If the two values are equal, the result at P is the same A:B, since ANDing a value with itself returns the same value. At the pattern detect logic, the P output is again compared with C and if P is equal to C, the status flag *patterndetect* is set to high. Otherwise, if the pattern does not match, the status flag is set to low. In iDEA, we use subtraction followed by a comparison on the output, performed in logic, to test equality, greater than, and less than.

3.3.4. Shift. Shift shares the same datapath configuration as multiply. Data is shifted left n bits by multiplying by 2^n. This requires additional logic for a lookup table to convert n to 2^n before entering path B. For a shift right, higher-order bits of P are

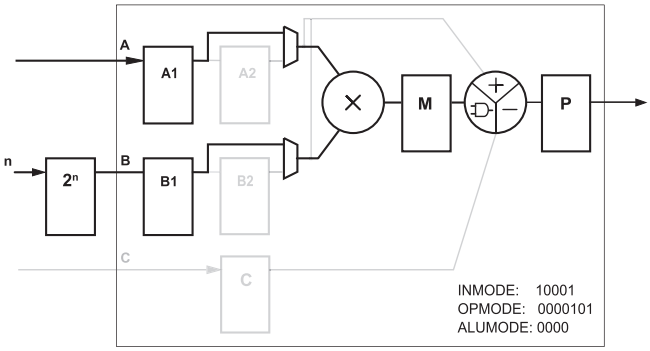

Fig. 5. Datapath for shift. An extra shift LUT is required.

Table III. Operations Supported by the DSP48E1

					DSP Stage		
Operation	INMODE	OPMODE	ALUMODE	Path	1	2	3
mul	10001	0000101	0000	mult	A1, B1	M	P
add	00000	0110011	0000	non-mult	A1, B1, C0	A2, B2, C	P
sub	00000	0110011	0011	non-mult	A1, B1, C0	A2, B2, C	P
and	00000	0110011	1100	non-mult	A1, B1, C0	A2, B2, C	P
xor	00000	0110011	0100	non-mult	A1, B1, C0	A2, B2, C	P
xnr	00000	0110011	0101	non-mult	A1, B1, C0	A2, B2, C	P
or	00000	0111011	1100	non-mult	A1, B1, C0	A2, B2, C	P
nor	00000	0110011	1110	non-mult	A1, B1, C0	A2, B2, C	P
not	00000	0110011	1101	non-mult	A1, B1, C0	A2, B2, C	P
nand	00000	0110011	1100	non-mult	A1, B1, C0	A2, B2, C	P
mul-add	10001	0110101	0000	both	A1, B1, C0	M, C	P
mul-sub	10001	0110101	0001	both	A1, B1, C0	M, C	P
mul-acc	10001	1000101	0000	both	A1, B1, C0	M, P	P

used instead of the normal lower-order bits. Logical shift left, logical shift right, and arithmetic shift right can all be achieved using the DSP48E1 slice.

The preceding examples demonstrate the flexibility of the DSP48E1 primitive. In a similar manner, we can enable a number of different instructions as detailed in Table III.

4. PROCESSOR ARCHITECTURE

iDEA is a scalar processor based on a load-store RISC architecture. The main advantage of using RISC is the uniform instruction set that leads to more straightforward decode logic and simpler hardware. It is worth noting that this decision is to allow us to demonstrate the feasibility of our approach, but alternative architectures are something we hope to explore in the future. iDEA executes 32-bit instructions on 32-bit data. Only a single DSP48E1 is used, with much of the processing for arithmetic, logical operations, and program control done within it. The overall architecture is shown in Figure 6.

We use a RAM32M LUT-based memory primitive for the register file and a RAMB36E1 block RAM primitive for instruction and data memory. While synthesis tools can infer primitives from high-level code, this work is all about leveraging the wide array of low-level functions supported by the DSP48E1, so we directly instantiate the primitive to exercise full control over it. The synthesis tools are

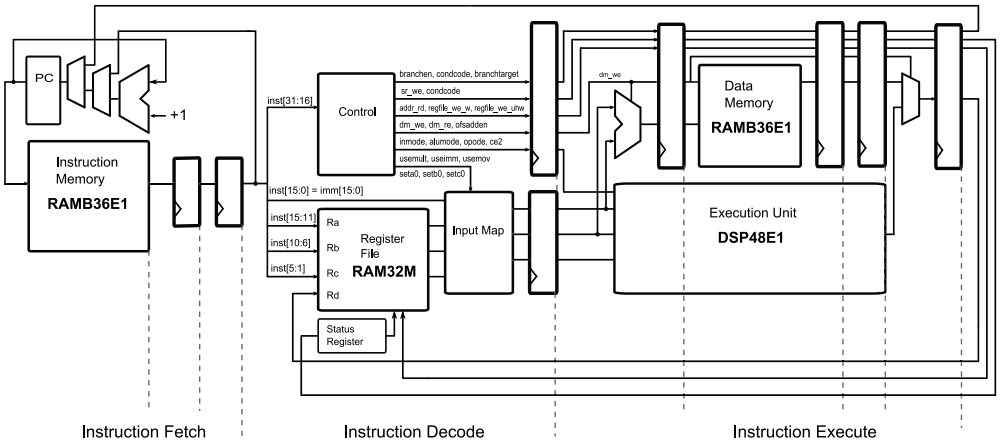

Fig. 6. iDEA processor block diagram.

currently unable to take full advantage of the dynamic programmability of the DSP block. Similarly, the RAM32M is directly instantiated for efficiency.

4.1. Instruction and Data Memory

The instruction and data memories are built using Block RAMs (BRAMs). BRAM read and write operations require only 1 clock cycle and the user has the option to enable an internal output register to improve the clock-to-output timing of the read path. The internal register is located inside the BRAM itself and, if enabled, improves the timing by 2.7× at the cost of an extra clock cycle of latency.

Adding another register in the logic fabric to register the output of the BRAM further improves timing. Without this, the critical path runs from the BRAM to the register of the next pipeline stage. For example, if the logic output register of the instruction memory is absent, the critical path will terminate at the pipeline register of the instruction decode stage.

We implement the instruction and data memories through inference rather than using direct instantiation or the CORE generator, as this eases design and portability while still offering the maximum performance.

4.2. Register File

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. To implement a 32×32-bit register file, 16 of these primitives are aggregated. While manual instantiation ensures only the required logic is used. The RAM32M is an optimised collection of LUT resources, and a 32×32 register file built using 16 RAM32Ms consumes 64 LUTs while relying on synthesis inference resulting from usage of 128 LUTs.

Using a block RAM for the register file may be beneficial as it would offer a significantly higher register count, however, this would require a custom design to facilitate the quad-port interface. In replication, an extra BRAM is needed to support each additional read port while the number of write ports remains unchanged. If banking is used, data is divided across multiple memories, but each read/write port can only access its own memory section. Similar to replication, banking requires 2 BRAMs for a 3-read, 1-write register file.

Table IV. Frequency for Different Pipeline Configurations of the DSP48E1 in Virtex-6 Speed-2

Pipeline	Freq. (MHz)
3-stage without pattern detect	540
3-stage with pattern detect	483
2-stage without pattern detect	311
2-stage with pattern detect	286
1-stage without pattern detect	233
1-stage with pattern detect	219

Creating multiported memories for an arbitrary number of ports out of BRAMs is possible, but entails both an area and speed overhead. In LaForest and Steffan [2010], an implementation using Altera M9K BRAMs for a 2-write, 4-read multiported memory achieves a clock frequency of 361 MHz, a 52.3% drop in frequency against a distinct M9K primitive. Hence, since we are targeting a design that is as close to the silicon capabilities as possible with as small an area as possible we do not use BRAMs for the register file, though this would be an option for more advanced architectures.

4.3. 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. Only through direct instantiation can we exploit the dynamic flexibility of the control signals of the DSP block.

All three pipeline stages of the DSP48E1 are enabled to allow it to run at its maximum frequency. With only a single stage enabled, the highest frequency achievable is reduced by half. Table IV shows the advertised frequency for different configurations of the primitive. To further improve performance, a register is added to the output of the primitive, helping to ensure that routing delays at the output do not impact performance. As a result, the total latency of the execution unit is 4 clock cycles.

The DSP48E1 primitive 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. Due to the adverse impact on the frequency of enabling the pattern detector, we instead use subtraction with a subsequent comparison for this purpose. This also allows us to test for greater than and less than, in addition to equality.

DSP48E1 features that are relevant to iDEA functionality are:

—25×18-bit multiplier;
—48-bit Arithmetic and Logic Unit (ALU) with add/subtract and bitwise 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 nonmultiply operations;

—OPMODE dynamic control signal for selecting operating modes;
—ALUMODE dynamic control signal for selecting ALU modes; and
—optional input, pipeline, and output registers

Incorporating the D input and preadder would make the instruction format more complex, likely requiring it to be widened, and would also require a more complex register file design to support four simultaneous reads. Preliminary compiler analysis on a set of complex benchmarks has shown that patterns of add-multiply-add/sub instructions are very rarely used. Since we do not have access to intermediate stages of the pipeline within the DSP block, we can only create a merged instruction when three suitable operations are cascaded with no external dependencies, hence the benefits of incorporating the D input and preadder into iDEA are far outweighed by the resulting cost, and so we disable them.

4.4. Other Functional Units

All other functional units are implemented in LUTs. These include the program counter, branch logic, control unit, status register, input map, and an adder for memory address generation. All the modules are combinational circuits except for the program counter and status register that are synchronous. These modules occupy minimal LUT and flip-flop resources as the bulk of processor functionality is inside the DSP48E1, thus achieving a significant area and power advantage over a LUT-based implementation.

5. THE IDEA INSTRUCTION SET

The iDEA instruction set is listed in Table V. Though not as extensive as more advanced commercial processors, it is sufficient for illustrating the functionality of iDEA in executing arithmetic and data processing applications. A uniform 32-bit instruction width is used. Unlike a typical execution unit that processes only two input operands, our execution unit is capable of processing three input operands and the instruction format is designed to cater for a third operand field to reflect the extra processing capability, as detailed in Table VI.

5.1. Input Mapping

The location of input operands in the register file 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; these widths are distinct and 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 the ports that it is mapped to, particularly for ports A and B, which are concatenated for ALU functions. The dataflow through the DSP48E1 can be represented as

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

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.

Table V. iDEA Instruction Set

Instruction	Assembly	Operation
Arithmetic/ Logical		
nop	nop	none
add	add rd, ra, rb	rd[31:0] = ra[31:0] + rb[31:0]
sub	sub rd, ra, rb	rd[31:0] = ra[31:0] − rb[31:0]
mul	mul rd, rb, rc	rd[31:0] = rb[15:0] × rc[15:0]
mac	mac rd, rb, rc, rp	rd[31:0] = rb[15:0] × rc[15:0] + rp[31:0]
madd	madd rd, ra, rb, rc	rd[31:0] = ra[31:0] + (rb[15:0] × rc[15:0])
msub	msub rd, ra, rb, rc	rd[31:0] = ra[31:0] − (rb[15:0] × 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 rd, ra, rb	rd = 1 if ra[31:0] < rb[31:0]
	cmp rd, ra, #imm11	rd = 1 if 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]

Table VI. iDEA Processor Instruction Format

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

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

Eq. (2) shows a 3-operand, 2-operation instruction where 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

Table VII. Port Mapping for Different Arithmetic Functions

Assembly Inst.	Operation	Port A (30b)	Port B (18b)	Port C (48b)
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 Rd, Ra, Rb	$C - A{:}B$	16{Rb[31]}, Rb[31:18]	Rb[17:0]	16{Ra[31]}, Ra[31:0]
mul Rd, Rb, Rc	$C + A \times B$	15{Rb[15]}, Rb[15:0]	2{Rc[15]}, Rc[15:0]	48{1'b0}
madd Rd, Ra, Rb, Rc	$C + A \times B$	15{Rb[15]}, Rb[15:0]	2{Rb[15]}, Rc[15:0]	16{Ra[31]}, Ra[31:0]
movl Rd, #imm16	$C + A \times B$	30{1'b0}	18{1'b0}	32{1'b0}, imm[15:0]

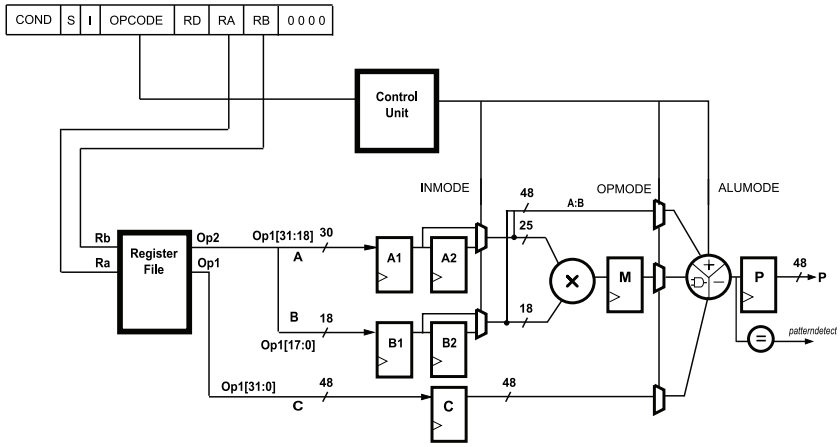

Fig. 7. 2-operand input map.

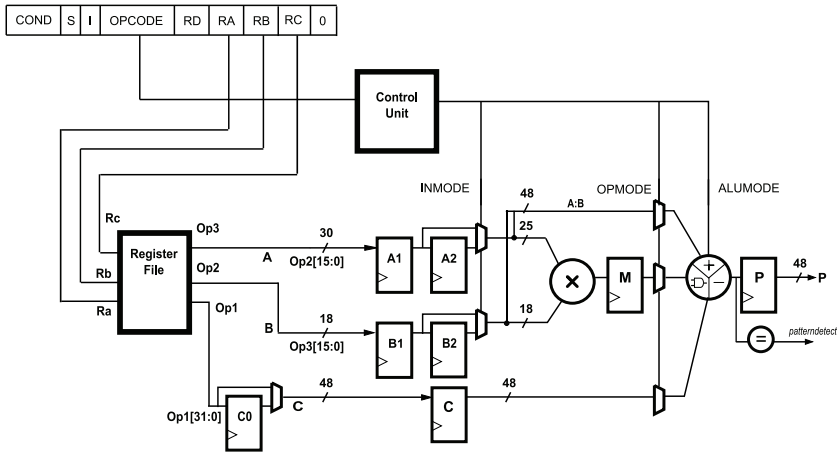

Fig. 8. 3-operand input map.

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 nonzero data.

The DSP48E1 can be dynamically switched between operations defined by Eqs. (1) and (2) through the INMODE, OPMODE, and ALUMODE control signals. Table VII illustrates the port mappings for some common instructions while Figure 7 and Figure 8 show how the fields in the instruction are mapped to an operation in the DSP48E1 execution unit.

Table VIII. Frequency and Area for iDEA and MicroBlaze

Stages	Freq. (MHz)	Registers	LUTs	DSP48E1
iDEA				
5	193	273	255	1
6	257	256	254	1
7	266	308	263	1
8	311	319	293	1
9	405	413	321	1
9 *LUTs*	173	571	864	1
MicroBlaze				
3	189	276	630	3
5	211	518	897	3

5.2. DSP Output

The multiplier in the DSP48E1 enables multiplication and shift to be efficiently 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-accumulate. The DSP48E1 primitive produces an output of 48 bits through port P, regardless of the type of arithmetic operation; the least significant 32 are written back to the register file.

It is important to note that the multiplier width 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 that 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. If a wide multiply is required, it can be executed as a series of 16-bit multiplications.

For floating-point operations, we can use the compiler to translate them into a series of steps that can be executed using the DSP48E1 primitive as per the method in Brosser et al. [2013].

6. 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 processor from Xilinx. In Section 7, we benchmark a few general-purpose applications to demonstrate the functionality of iDEA.

All implementation and testing was performed on a Xilinx Virtex-6 XC6VLX240T-2 device as present on the Xilinx ML605 development board.

6.1. Area and Frequency Results

Table VIII shows the post-place-and-route implementation results for iDEA and MicroBlaze. For iDEA, the implementation is performed using Xilinx ISE 14.5 while MicroBlaze is implemented using Xilinx Platform Studio (XPS) 14.5. Both implementations include memory subsystems and the processor core and a total of 4KB is allocated for instruction and data memory for each of the processors.

As the Xilinx DSP48E1 and RAMB36E1 primitives used in iDEA are highly pipelinable, we study the effect of varying the number of pipeline stages from 1 to 3. This translates to an overall processor pipeline depth of 5 to 9 stages. As expected, a deeper pipeline yields a higher clock frequency; from the minimum pipeline depth of 5 to a

Table IX. iDEA in Artix-7, Kintex-7, and Virtex-7

Resource	Virtex-6	Artix-7	Kintex-7	Virtex-7
Slice Registers	413	416	420	420
Slice LUTs	321	324	343	324
RAMB36E1	2	2	2	2
DSP48E1	1	1	1	1
Freq (MHz)	405	281	410	378

maximum pipeline depth of 9, the clock frequency increases by 52% at a cost of 33% more slice registers and 20.5% more slice LUTs. The extra slice registers and LUTs are consumed by extra registers external to the DSP block, required to maintain alignment.

In order to quantify the benefit of using the DSP48E1 in the manner we have, we coded the exact behaviour of the DSP-based execution unit and implemented it in the logic fabric. Table VIII shows that a LUT-based equivalent occupies 38% more registers and 169% more LUTs compared to iDEA. Apart from slice logic, the tool still synthesized a DSP block for the 16×16 multiplication, whereas the LUT-based equivalent achieved a clock frequency of 173 MHz, just 42% of iDEA's frequency.

The MicroBlaze results are presented purely to give a sense of relative scale and we do not claim that iDEA can replace MicroBlaze in all scenarios. To make the comparison fairer, we configure the smallest possible MicroBlaze while keeping all the basic functionality necessary to run applications. Extra peripherals and features that are not available in iDEA, such as cache, memory management, and the debug module, are disabled. The multiplier is enabled and set to the minimum configurable width of 32 bits. Other hardware like the barrel shifter, floating-point unit, integer divider, and pattern comparator are disabled.

MicroBlaze can be configured with two different pipeline depths-3 stages for an area-optimized version or 5 stages for a performance-optimized version. The 5-stage MicroBlaze uses 25% more slice registers and 179% more slice LUTs. MicroBlaze includes some additional fixed features such as special-purpose registers, instruction buffer, and bus interface that contribute to the higher logic count. These MicroBlaze features are not optional and cannot be removed by the end-user. MicroBlaze also supports a wider multiplication width of 32×32, resulting in the use of 3 DSP48E1 slices instead of one. A 3-stage area-optimized MicroBlaze occupies 33% fewer slice registers and 96% more LUTs than iDEA. A reduced 5-stage version of iDEA would be on par with the 3-stage MicroBlaze in terms of frequency and slice registers, but still consume 59% fewer slice LUTs.

To confirm the portability of iDEA, we also implemented the design on the Xilinx Artix-7, Kintex-7, and Virtex-7 families. The resource consumption and maximum operating frequency post-place-and-route, shown in Table IX, 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.

6.2. Optimising the Number of Pipeline Stages

A crucial question to answer is how to balance the number of pipeline stages with the frequency of operation. Enabling extra pipeline stages in FPGA primitives allows us to maximise operating frequency as demonstrated in the previous section, however, we must also consider how this impacts the processor as a whole, as well as the resulting costs in executing code. Table X shows a breakdown of the pipeline stages of each stage of the processor, alongside the resulting achievable frequency for different overall pipeline depths. Note that we have taken the best result where different breakdowns are possible. It is clear that 9 pipeline stages provides the highest performance in terms of operating frequency. However, this comes at a cost: since iDEA is a simple processor

Table X. iDEA Achievable Frequency for Different Pipeline Lengths

IF	ID	EX	WB	Total	Freq. (MHz)
1	1	2	1	5	193
2	1	2	1	6	257
3	1	2	1	7	266
3	1	3	1	8	311
3	1	4	1	9	405

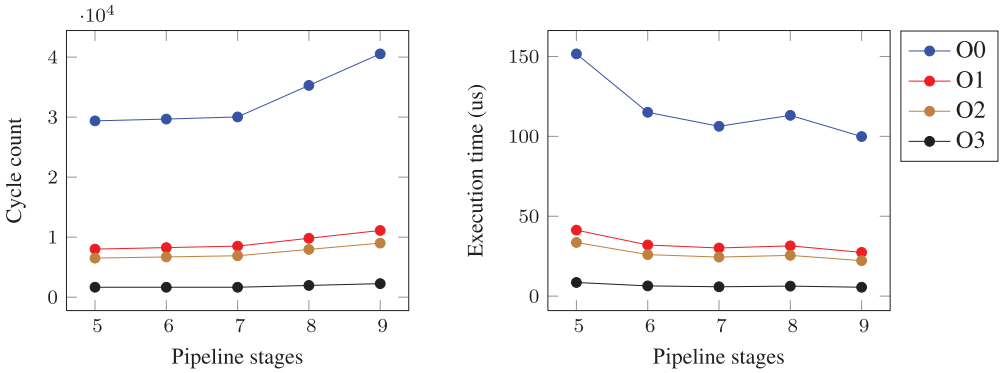

Fig. 9. Cycle count and execution time for FIR application with varied number of pipeline stages.

with no data forwarding, we must insert empty instructions to overcome data hazards. With a longer pipeline, more of these are needed, thus impacting the actual runtime.

As an example, we take an FIR filter application and investigate the result of increasing the number of pipeline stages on the total number of cycles required after insertion of blank instructions and factoring in the increased frequency. We do this using the compiler and simulator detailed in Section 7. The FIR example is representative of the other benchmarks we will describe in Section 7 in this regard.

Looking at Figure 9, the cycle count shows only a slight increase, namely, 1.7%, from 5 to 6 and 6 to 7 pipeline stages, while from 7 to 8 and from 8 to 9 stages we see a steep increase of 18.9% and 15.9%, respectively (note that the different optimisation levels are discussed in Section 7). This can be explained by the fact that from 5 stages to 6 or 7, only IF stages are added to the pipeline. IF stages are placed before ID in the pipeline, which means that they will not contribute to latency when waiting for a previous instruction to finish, as IF stages can overlap with the previous instruction. This is illustrated by Figure 10, showing the execution of two consecutive instructions in iDEA for 6, 7, and 8 pipeline stages. In this example, the second instruction depends on the result of the first instruction.

By adding EX/MEM stages as we do when increasing from 7 to 8 or 8 to 9 pipeline stages, we add one extra clock cycle of latency for dependent instructions, resulting in one extra *NOP* being inserted to resolve the dependency. This extra *NOP* adds to the instruction count, hence the increase in cycle count seen in Figure 9 can be fully attributed to the added *NOP* insertions required. Adding different types of stages will impact the cycle count in different ways, depending on how the instructions can overlap when there is a dependency and also on how branching is implemented. While adding IF stages will not increase the number of *NOP*s inserted to resolve dependencies, it will still increase the number of cycles that are lost when branching, because of the added delay between fetching the branch instruction to the actual update of the PC.

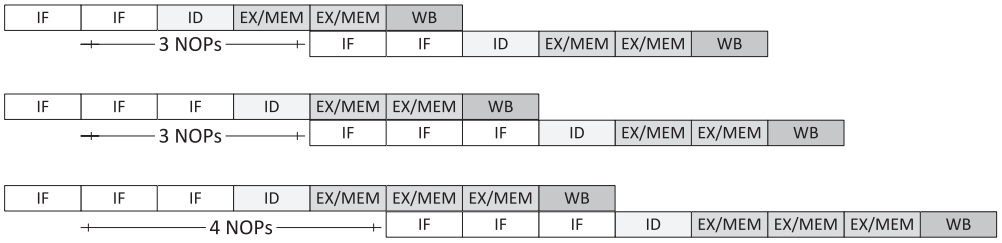

Fig. 10. Dependency in iDEA pipeline for 6, 7, and 8 stages.

This explains why there is a slight increase from 5 to 6 and from 6 to 7 stages. However, this has a comparatively small impact on cycle count for most programs. As expected, -O0 has a much higher cycle count than the other optimisation levels, while for FIR, the -O3-optimised code benefits greatly from loop unrolling and shows a much lower cycle count than -O2.

The execution-time graph brings together the cycle count and operating frequency presented in Table X. It shows a steep improvement from 5 to 6 stages and from 8 to 9 stages, attributable to the large frequency increases between these numbers of stages. Going from 7 to 8 stages, the percentage of increases in cycle count and frequency are roughly equal, resulting in almost equal execution time. The 9-stage pipeline gives the lowest execution time for FIR, despite having the largest number of *NOP* insertions. This result applies to the other benchmarks as well, making the 9-stage pipeline the best performing overall. These 9 stages 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 instructions require the full 9 stages, for example, branch instructions and data memory accesses execute in fewer cycles.

7. IDEA SIMULATOR AND BENCHMARKS

Having built iDEA, we would now like to evaluate its performance for executing applications. It is important to state that iDEA is, by definition, a lean processor for which performance was not the primary goal. Additionally, as of now, there is no optimised compiler for iDEA, so the results presented in this section are aimed primarily at proving functionality and giving a relative performance measure. Only with a custom compiler can we extract maximum performance and enable the use of iDEA's unique extra capabilities.

For the MicroBlaze results, the C applications are compiled using the C compiler from the Xilinx Software Development Kit 14.5 (SDK), *mb-gcc*. The compiler automatically ensures it does not generate instructions for features of the MicroBlaze that have been disabled in our implementation.

We have prepared a number of small application benchmarks and gather results using an instruction-set simulator written for iDEA. Using the simulator, we obtain performance metrics such as instruction count and number of clock cycles, as well as ensuring logical and functional correctness. Since we do not have a custom compiler, we chose an existing MIPS I compiler that supports an instruction set similar to that of iDEA. The benchmark programs are written in C and compiled to *elf32-bigmips* assembly code using the *mips-gcc* toolset. The instructions generated must be translated to equivalent iDEA instructions. The simulator consists of an assembly code converter and a pipeline simulator. The complete tool chain from C program to simulator is illustrated in Figure 11.

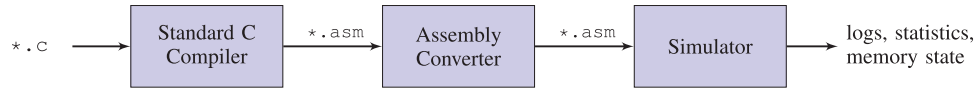

Fig. 11. Simulator and tool-chain flow.

7.1. Assembly Converter

The assembly converter takes as its input an *elf32-bigmips* format assembly file, the standard assembly code format produced by *mips-elf-gcc*. By setting the appropriate flags, *-c -g -O*, the compiler bypasses the linker, instead giving only the assembly code. This assembly code is converted to the format used by the iDEA simulator and then processed to prepare for execution in the simulator. These preparations include expanding pseudo-instructions found in the assembly code, performing instruction substitutions where necessary to fit the iDEA instruction set, NOP insertion to resolve dependencies, and classification of instructions for collation of statistics. The simulator's data memory is preloaded with the data specified in the *elf32-bigmips* assembly code. This preloaded data includes constants declared in the original C program. Because the iDEA architecture does not currently implement hardware stalling or data forwarding, dependencies have to be resolved in software. The code is checked for dependencies and *NOP* instructions are inserted where necessary to avoid data hazards.

A dependency is detected if the operand register of an instruction is equal to the destination register of a previous instruction that has not yet completed its write-back. For example, in the 8-stage pipeline of Figure 10, a dependent instruction can only be fetched after four instruction cycles to ensure correct computation. *NOP*s are inserted to resolve the dependency.

Overlapping stores are defined as any store instructions with references to the same memory location. After the data dependencies have been found in this first iteration, all branch and jump targets are reevaluated according to the new instruction memory locations. In the third and final iteration, branches and jumps are checked for dependencies across PC changes, which may require additional *NOP*s to be inserted. Finally, the branch and jump targets are again reevaluated and the final instruction list with the inserted *NOP*s is passed to the simulator.

7.2. Simulator

The simulator can be configured to model a variable number of pipeline stages and different pipeline configurations. All iDEA instructions that are used in the benchmarks are supported by the simulator. The simulator models the iDEA pipeline stage-by-stage and the execution of the instructions as they pass between stages. The register file, data memory, and instruction memory are modelled individually as separate modules. The statistics collected during the simulation run are cycle count, *NOP* count, simulation runtime, and core cycle count.

We manually insert the start and end tags in the assembly source code to define where the computational cores of the programs start and end, with the purpose of eliminating initialization instructions from the final cycle count. The results presented in Section 7.4 are the core cycle counts.

7.3. Benchmarks

Seven benchmarks, briefly presented shortly, are used to evaluate performance. They are written in standard C with an internal self-checking loop to verify correctness and to reduce simulator complexity. The applications are fundamental loops; they are kernels of larger algorithms and often the core computation loops of more extensive,

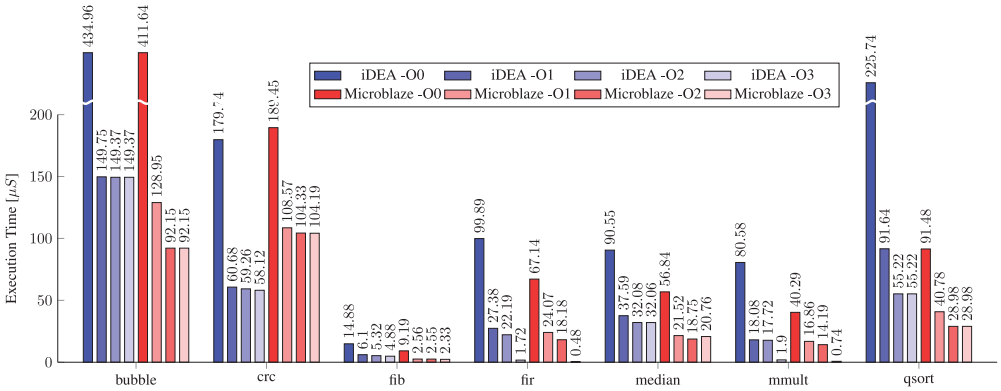

Fig. 12. Comparison of execution time of iDEA and MicroBlaze at maximum pipeline stages.

practical applications. For example, sorting algorithms are commonly found in networking applications, FIR and Median are found in digital signal processing applications, and CRC in storage devices and matrix multiplication in linear algebra.

—*Bubble.* This is a bubble sort on an array of 50 random integers.
—*CRC.* This is an 8-bit CRC on a random 50-byte message.
—*Fib.* This calculates the first 50 Fibonacci numbers.
—*FIR.* This is an FIR filter using 5 taps on an array of 50 integers.
—*Median.* This is a median filter with a window size of 5, on an array of 50 integers.
—*MMult.* This consists of matrix multiplication of two 5×5 integer matrices.
—*QSort.* This is a quick-sort on an array of 50 random integers.

These benchmarks are sufficient to demonstrate the complete tool chain of iDEA including compiler, simulator, and processor. We aim to explore more complex benchmarks after building a custom compiler, also allowing us to present more accurate performance metrics as the code will be optimised for the special features of iDEA.

The instruction count and clock-cycle count for MicroBlaze are obtained by testbench profiling using an HDL simulator, as the current Xilinx tools do not provide an instruction-set simulator for MicroBlaze. 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 signals are obtained from the instruction opcode in the disassembly file, while the C code is 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 accordingly starts and stops the count tracking.

7.4. Execution Results

With a compiler, we can generate instruction code for iDEA for the benchmarks at different optimization levels. Figure 12 shows the total execution time of both processors for all seven test applications (Bubble, Fibonacci, FIR, Median, matrix multiplication, Quick-sort and CRC) at four different optimisation levels (O0, O1, O2, O3).

Overall, iDEA has a higher execution time compared to MicroBlaze due to the insertion of *NOPs* to handle data hazards. Figure 13 shows the relative execution

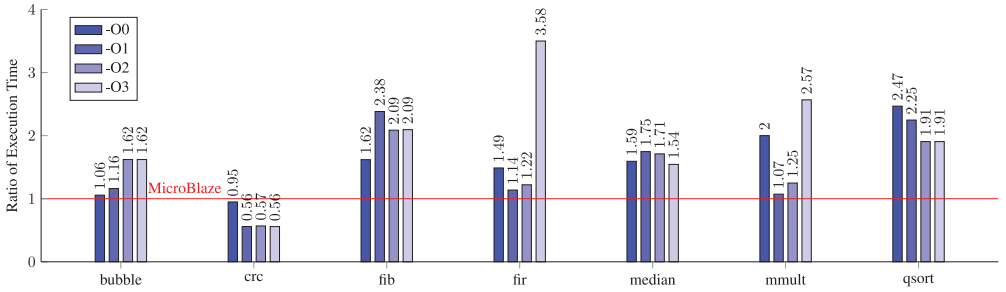

Fig. 13. Execution time of iDEA relative to MicroBlaze.

time of iDEA with MicroBlaze normalised to 1 for each optimisation level. Of all the benchmarks, CRC is the only application that has faster execution time on iDEA than MicroBlaze; despite a higher number of clock cycles, the improved frequency of iDEA results in a lower execution time.

In most benchmarks, the *NOP* instructions make up the vast majority of the total instructions executed, that is, between 69.0% and 86.5%. This can partially be traced to the compiler, that, targets the MIPS platform rather than iDEA and, therefore, does not generate efficient code. The compiler follows the MIPS conventions for register usage, resulting in many *NOP*s being inserted to resolve dependencies. Currently, the same registers are often reused for consecutive instructions (v0 and v1 in the MIPS convention), thereby creating dependencies that have to be resolved by *NOP* insertion. Spreading out register usage over the whole register file would significantly decrease the *NOP* count, however, at the moment this must be done manually due to the lack of a compiler targeted to iDEA. These factors together contribute to a significant overhead added by the inserted *NOP* instructions.

The effect of register reuse is particularly evident in FIR and matrix multiplication. At optimisation level -O3, the loop is unrolled to a repeated sequence of *add* and *store* instructions without any branching in between. While the absence of branching reduces the branch penalty, the consecutive dependency between the instructions demands that *NOPs* be inserted, causing an increase in overall execution time.

Recall that memory operations consume fewer cycles as they do not use the execution unit. Hence, 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 instructions 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.

In order to maintain the leanness of iDEA, we have avoided the addition of data forwarding or stalling. This makes the compiler critical in extracting maximum performance. An ideal compiler for iDEA would rearrange instructions to exploit the NOP slots and make use of composite instructions where several operations can be executed with a single instruction. With the availability of two arithmetic components in iDEA (or three in the DSP48E1 if the preadder is enabled), we can explore the possible benefits of this. For example, two consecutive instructions `mul r3, r2, r1; add r5, r4, r3` have a read-after-write dependency and *NOPs* have to be inserted to allow the result of the first instruction to be written back to the register file before execution of the second. By combining these into a single instruction `mul-add r5, r1, r2, r4`, two instructions can be executed as one, reducing the number of useful instructions required to perform the same operations and also removing the necessity for NOPs in between.

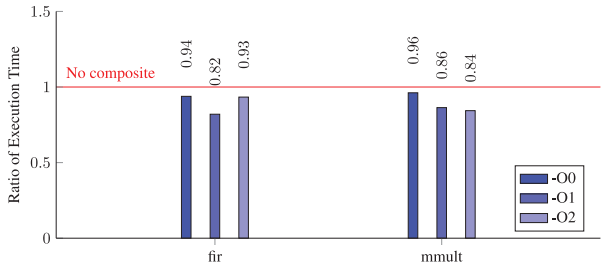

Fig. 14. Relative execution time of benchmarks using compound instructions.

The three-operand multiply-accumulate instruction maps well to the DSP48E1 block and is supported by the iDEA instruction set. To explore the potential performance improvement when using composite instructions, we examine the FIR and Mmult benchmarks after modifying the code to use the `mac` instruction. Currently, this modification is done manually, as the compiler does not support this instruction. We manually identify the pattern `mult r3, r1, r2; add r4, r4, r3` and change it to `mac r4, r1, r2`. A compiler could automatically identify the multiply-accumulate pattern and make use of the instruction.

Figure 14 shows the relative performance when using these composite instructions compared to the standard compiler output (normalised to 1). We see that the use of composite instructions in a 9-stage iDEA pipeline can indeed provide a significant performance improvement. FIR-O1 shows the best execution-time improvement of 18%, while the -O0 optimisation levels for both benchmarks show only slight improvements of 6% and 4% for FIR and Mmult, respectively. The benchmarks that are shown here use computation kernels that are relatively small, making the loop overhead more significant than the computations themselves and thus limiting the potential for performance savings. For more complex benchmarks, there is a greater potential for performance improvement resulting from the use of compound instructions. Our preliminary analysis shows that it is possible to extract opportunities for compound instructions in common embedded benchmark programs, not just programs that are domain specific such as DSP processing or media processing.

However, there are still limitations in the DSP block architecture and in the design of iDEA that cannot be addressed without a custom compiler. On the hardware side, to support 4-operand instructions would require the register file to handle four reads and one write per cycle, which is not possible with the RAM32M.

7.5. Summary

The small benchmark results in this section demonstrate that iDEA is functional and offers performance comparable to other soft processors. iDEA does suffer a long datapath and, when padding dependent instructions with NOPs, significant overhead is introduced. To truly exploit its capabilities, a custom compiler is required that can overcome issues related to register reuse and can support the use of compound instructions. With the custom compiler we are working on, we intend to extend our experimental analysis from small benchmarks to more practical fixed-point embedded application benchmarks.

8. CONCLUSION

This article has presented iDEA, an instruction-set-based soft processor for FPGAs built with a DSP48E1 primitive as the execution core. We harness the strengths of the

DSP48E1 primitive by dynamically manipulating its functionality to build a load-store processor. This makes the DSP48E1 usable beyond just signal processing applications.

As iDEA is designed to occupy minimal area, the logic is kept as simple as possible. By precluding more complex features 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. We have shown that iDEA runs at about double the frequency of MicroBlaze while occupying around half the area. iDEA can be implemented across the latest generation of Xilinx FPGAs, achieving comparable performance on all devices.

We presented a set of seven small benchmark applications and evaluated the performance of iDEA by using translated MIPS-compiled C code. We showed that, even without an optimised compiler, iDEA can offer commendable performance, though it suffers significantly from the need for NOP insertion to overcome data hazards. We also evaluated the potential benefits of iDEA's composite instructions, motivating the need for a custom compiler to maximise performance.

Our current and future work is focused on developing a custom compiler that can produce more efficient programs for iDEA. This will allow us to explore more complex benchmarks and present more representative performance results. We are also keen to explore how such a lean processor might be used in a massively parallel manner, given the high number of DSP48E1s on modern FPGAs.

REFERENCES

Aeroflex Gaisler. 2012. GRLIB ip library user's manual. http://aeroflex.bentech-taiwan.com/aeroflex/pdf/grlib.pdf.

Altera Corpration. 2011. Nios II processor design. http://www.altera.com/literature/lit-nio2.jsp.

Arm Ltd. 2011. Cortex-m1 processor. http://www.arm.com/products/processors/cortex-m/cortex-m1.php.

L. Barthe, L. V. Cargnini, P. Benoit, and L. Torres. 2011. The secretblaze: A configurable and cost-effective open-source soft-core processor. In *Proceedings of the International Symposium on Parallel and Distributed Processing Workshops (IPDPSW'11)*. 310–313.

F. Brosser, H. Y. Cheah, and S. A. Fahmy. 2013. Iterative floating point computation using FPGA DSP blocks. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'13)*.

P. Buciak and J. Botwicz. 2007. Lightweight multi-threaded network processor core in FPGA. In *Proceedings of the Design and Diagnostics of Electronic Circuits and Systems Conference (DDECS'07)*. 1–5.

H. Y. Cheah, 2013. iDEA FPGA soft processor. https://github.com/archntu/idea.

H. Y. Cheah, S. A. Fahmy, and D. L. Maskell. 2012a. iDEA: A DSP block based FPGA soft processor. In *Proceedings of the International Conference on Field Programmable Technology (FPT'12)*. 151–158.

H. Y. Cheah, S. A. Fahmy, D. L. Maskell, and C. Kulkarni. 2012b. A lean FPGA soft processor built using a DSP block. In *Proceedings of ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'12)*. 237–240.

C. H. Chou, A. Severance, A. D. Brant, Z. Liu, S. Sant, and G. Lemieux. 2011. VEGAS: Soft vector processor with scratch-pad memory. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'11)*. 15–24.

X. Chu and J. Mcallister. 2010. FPGA based soft-core SIMD processing: A MIMO-OFDM fixed-complexity sphere decoder case study. In *Proceedings of the International Conference on Field Programmable Technology (FPT'10)*. 479–484.

F. de Dinechin and B. Pasca. 2011. Designing custom arithmetic data paths with flopoco. *IEEE Des. Test Comput.* 28, 4, 18–27.

R. Dimond, O. Mencer, and W. Luk. 2005. CUSTARD—A customisable threaded FPGA soft processor and tools. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'05)*.

J. Kathiara and M. Leeser. 2011. An autonomous vector/scalar floating point coprocessor for FPGAS. In *Proceedings of the International Symposium on Field-Programmable Custom Computing Machines (FCCM'11)*. 33–36.

T. Kranenburg and R. van Leuken. 2010. MB-LITE: A robust, light-weight soft-core implementation of the microblaze architecture. In *Proceedings of the Design, Automation, and Test in Europe Conference (DATE'10)*. 997–1000.

C. E. Laforest and J. G. Steffan. 2010. Efficient multi-ported memories for FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'10)*.

C. E. Laforest, J. G. Steffan, and J. Gregory. 2012. OCTAVO: An FPGA-centric processor family. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'12)*. 219–228.

Lattice Semiconductor Corp. 2009. LatticeMico32 processor reference manual. http://www.lattice.us/~/media/Documents/UserManuals/JL/LatticeMico32HardwareDeveloperUserGuide.PDF?document_id=35467

Y. Lei, Y. Dou, J. Zhou, and S. Wang. 2011. VPFPAP: A special-purpose VLIW processor for variable-precision floating-point arithmetic. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'11)*. 252–257.

J. Lotze, S. A. Fahmy, J. Noguera, B. Ozgul, L. Doyle, and R. Esser. 2009. Development framework for implementing FPGA-based cognitive network nodes. In *Proceedings of the IEEE Global Communications Conference (GLOBECOM'09)*.

M. Milford and J. Mcallister. 2009. An ultra-fine processor for FPGA DSP chip multiprocessors. In *Conference Record of the Asilomar Conference on Signals, Systems and Computers*. 226–230.

F. Plavec, B. Fort, Z. G. Vranesic, and S. D. Brown. 2005. Experiences with soft-core processor design. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'05)*. 167b.

B. Ronak and S. A. Fahmy. 2012. Evaluating the efficiency of DSP block synthesis inference from flow graphs. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'12)*. 727–730.

A. Severance and G. Lemieux. 2012. VENICE: A compact vector processor for FPGA applications. In *Proceedings of the Workshop on the Intersections of Computer Architecture and Reconfigurable Logic*.

K. Vipin and S. A. Fahmy. 2012. A high speed open source controller for FPGA partial reconfiguration. In *Proceedings of the International Conference on Field-Programmable Technology (FPT'12)*. 61–66.

Xilinx. 2010. Virtex-4 family overview. http://www.xilinx.com/support/documentation/data_sheets/ds112.pdf.

Xilinx. 2011a. UG081: MicroBlaze processor reference guide. http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_1/mb_ref_guide.pdf.

Xilinx. 2011b. UG369: Virtex-6 FPGA DSP48E1 Slice user guide. http://www.xilinx.com/support/documentation/user_guides/ug369.pdf.

Xilinx. 2011c. Virtex-II pro and Virtex-II Pro X platform FPGAs data sheet. http://www.xilinx.com/support/documentation/data_sheets/ds083.pdf

S. Xu, S. A. Fahmy, and I. V. Mcloughlin. 2014. Square-rich fixed point polynomial evaluation on FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'14)*. 99–108.

P. Yiannacouras, J. G. Steffan, and J. Rose. 2008. VESPA: Portable, scalable, and flexible FPGA-based vector processors. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'08)*. 61–70.

J. Yu, G. Lemieux, and C. Eagleston. 2008. Vector processing as a soft-core CPU accelerator. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'08)*. 222–232.

Received May 2013; revised November 2013; accepted January 2014