ISA Extensions for High-Radix Online Floating-Point Addition

Pouya Dormiani[a], Miloš D. Ercegovac[a], and O. Colavin[b]

[a] Computer Science Department, UCLA, Los Angeles, CA 90025, USA
[b] ST Microelectronics Lab, San Diego, CA 92121, USA

ABSTRACT

ISA extensions for DLX type architectures are proposed to perform high radix online floating point addition on fixed point units with extended feature sets. Online arithmetic allows most significant digit first computation of results, allowing overlapped execution of dependent operations and offers greater instruction scheduling opportunities than software implementations of conventional floating point addition. In this paper we seek an ISA formulation to find a middle ground between full hardware floating point addition units and software implementations strictly based on available ALU logic.

Keywords: ISA, floating-point addition, online floating-point addition.

1. INTRODUCTION

Floating point addition has many stages, namely alignment, addition, normalization, rounding and exponent update. These stages are encapsulated by the floating point add instruction which performs all stages of the operation atomically, and transparently to the programmer. Many modern embedded processors however lack FPUs because of their power or area budget which forces programmers to use libraries that emulate floating point operations.[2] Emulation libraries perform floating point operations by mapping the desired functionality to shifting, masking and other bit wise logical operations on the ALU. Strict hardware oriented implementation and software implementation via standard ALU logic are both at extreme points in the design space with a large performance gap in between. In reality design constraints are often not this black and white and may benefit from an implementation that could increase performance without incurring the cost of a full blown floating point unit (FPU). Our motivation is to investigate an online arithmetic[3] approach that will strike a balance between cost and performance by proposing both a different software algorithm and ISA extensions to facilitate its implementation.

In this paper we will discuss two approaches to floating point addition, namely conventional floating point addition (CFPA), discussed in section 2 and online floating point addition (OFPA) discussed in section 3. We will present features of the OFPA algorithm which distinguish it as a better design solution. These features are focused around greater instruction scheduling opportunities in an OFPA ISA design as well as the ability to state the precision of the operation to be performed (discussed in greater detail in section 4.2).

2. CONVENTIONAL FLOATING POINT ADDITION

We begin by discussing conventional floating point addition (CFPA) in order to outline the algorithm. We will then draw upon this algorithm to develop the proposed ISA. Assume we have two operands A, and B, where operand X is described by the triple, (S_X, E_X, F_X) where S_X is the sign, E_X is the biased exponent and F_X is the significand. The algorithm consists of the following steps,

Exponent difference $d \leftarrow E_A - E_B$, where d is the exponent difference that is used to determine which operand is to be shifted to the right for alignment.

Swap The operand with the smaller exponent must be shifted right. Swap the operands so that the correct operand is shifted right. This eliminates the need for two shifters.

Right shift The right shift involves shifting the significand of the operand to be aligned by $|d|$ positions.

Advanced Signal Processing Algorithms, Architectures, and Implementations XVII, edited by Franklin T. Luk
Proc. of SPIE Vol. 6697, 66970T, (2007) · 0277-786X/07/$18 · doi: 10.1117/12.734916

Add significands Adding the significands typically involves two's complement addition.

LOD Leading One Detection finds the leading non-zero digit in the result of significand addition.

Left shift The resulting significand F_R is shifted to the left until the leading one becomes the hidden bit, or the tentative exponent becomes zero and the number becomes a denormal.

Round Rounding occurs after normalization at which point the rounding mode and the guard bits determine the final rounded value.

Adjust exponent Depending on how much left shift was performed, the exponent must be updated to reflect these shifts.

Special cases Special cases such as $\pm$ inf and NaN need separate detection and treatment.

A simplified implementation of the stages described in CFPA is shown in Fig. 1 which is adapted from.[5]

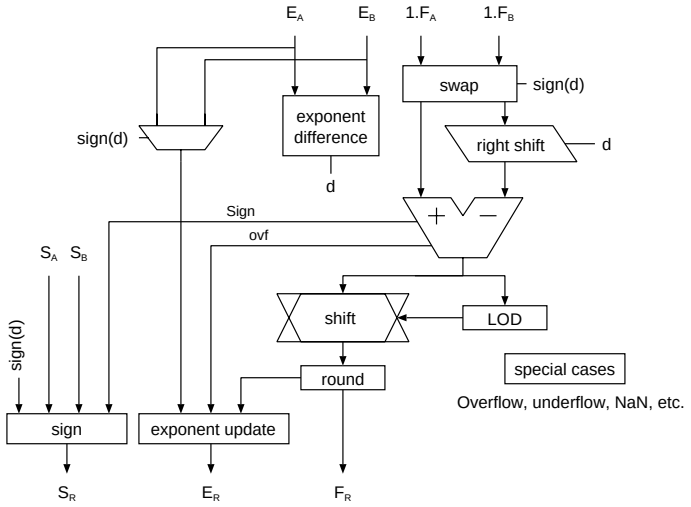


Figure 1. Single path CFPA implementation.

3. ONLINE FLOATING POINT ADDITION

In this section we briefly discuss radix-2 online floating-point addition and present the algorithm characteristics which were driving factors in selecting an online approach (a much more detailed discussion of OFPA is presented in.[4] OFPA is identical to CFPA in the stages that it performs; the main difference between the two designs is that CFPA typically uses digit-parallel addition while OFPA is strictly digit-serial and consequently the significand in OFPA is represented in redundant form.[6] The OFPA algorithm[3,5–8,15–17] is discussed in several papers with different degrees of specificity. It is also noteworthy to state that one such design[6] claims IEEE compliance while others[7,8] where developed before the IEEE 754 standard was established.

3.1 Representation

Online floating point operands are represented via the 3-tuple (S_X, E_X, F_X) where S_X and E_X are the sign and exponent which retain the same representation as CFPA. The significand F_X however is now represented via redundant borrow-save digit format to allow online processing. In light of this serial digit flow, other components in the design must also be converted to their serial counterparts and appropriate control added in order for the execution unit to be an online operator. Exponent processing can be done in parallel or serially. Since exponent precision is short, parallel evaluation would be preferable. Fig. 2 demonstrates the representation of a double precision operand for the OFPA execution unit.

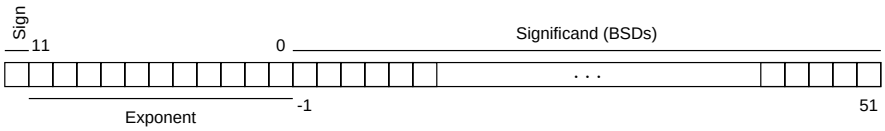


Figure 2. Online floating point representation for a double precision operand.

3.2 Timing model

Online execution implies that the sign is first delivered to the execution unit, followed by the exponent and the significand. The exponents are in non-redundant format, and shifted in serially. After the exponents are fully shifted into the unit, the exponent difference d is computed in parallel. Since the alignment quantity (d) must be evaluated before alignment can be performed, all the exponent bits must be received prior to the addition of the significands. Once the exponents are consumed, online addition of the significands can be initiated. The timing model in Fig. 3 demonstrates the timing behavior of the OFPA execution unit.

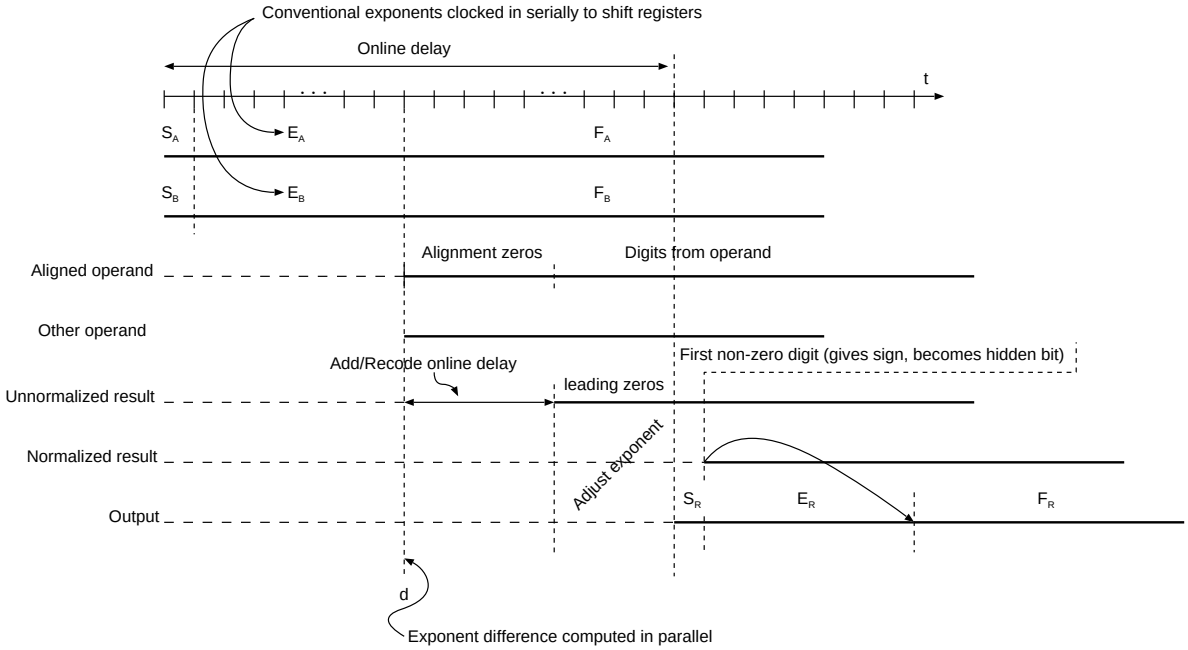


Figure 3. Timing model of OFPA. This figure is does not demonstrate a realistic run as there can not be leading zeros in the unnormalized result if there is an alignment shift in the operand.

The online delay (δ) of the execution unit is dependent on the presence of destructive cancellation in the result, but is guaranteed to be larger than the time required to deliver the sign and exponent, in addition to the online delay of the radix two adder and recoder. For a double precision execution, the bound $\delta \geq 16$ holds on the online delay, where delay is measured in clock cycles.

3.3 OFPA hardware implementation

Here we show the data path of an OFPA execution unit which is adapted from [6]. Figure 4 illustrates how CFPA has been translated to OFPA, and serves as a stepping stone to formulating an ISA that would perform a logically equivalent operation.

4. OFPA: AN ISA FORMULATION

This section formulates an ISA and algorithm for OFPA. We begin by stating our assumptions and constraints; make an argument for OFPA ISA extensions, and present some of the main instructions and their implementation in greater detail.

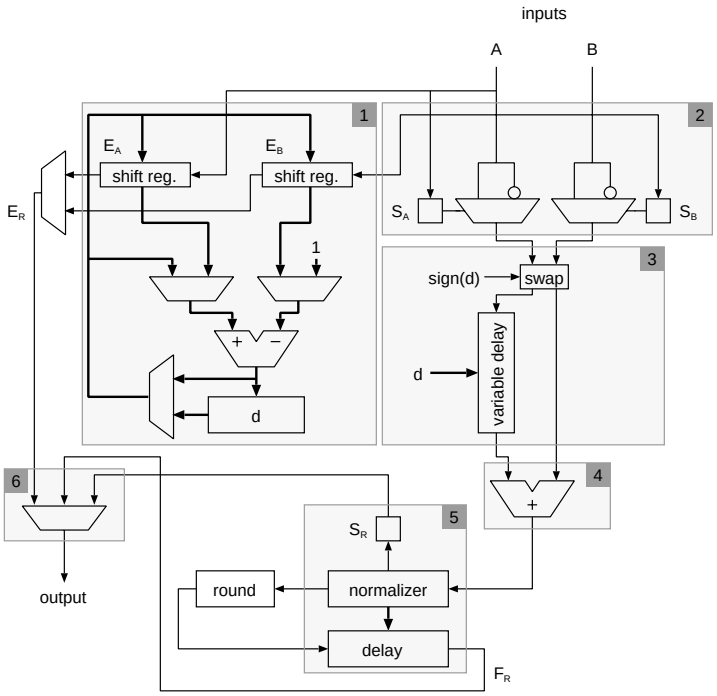
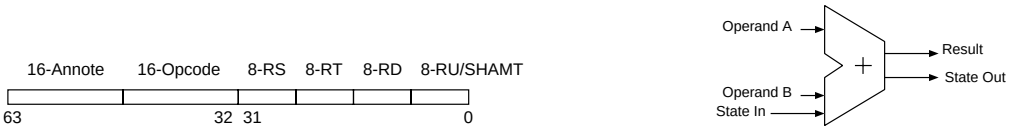


Figure 4. The OFPA data path. (1) Computes the exponent difference $d \leftarrow E_A - E_B$. Also performs exponent adjustment, by performing $E_X \leftarrow E_X \pm 1$. (2) Negates the significand digits according to the sign of the operand (the effective sign due to subtraction is omitted). (3) Swaps the operands accordingly so that the operand with the smaller exponent will be delayed (i.e. shifted right) by d cycles (positions) through the variable delay module. (4) Radix two online addition followed by PN^2 recoding[6,9]. (5) The normalization unit scans the digits being output by the adder/recoder to check for overflow as well as leading zeros. (6) The output of the module must be sequenced such that the sign (S_R) is output first followed by the exponent (E_R) and the significand of the result (F_R).

4.1 Assumptions and Constraints

An instruction set architecture (ISA) defines the logical semantics of a processing platform and the operations it provides to its users. Our proposed ISA is aimed to be an *extension* to a current ISA and thus should leverage the architecture's available characteristics, and not demand architectural enhancements at a system level–such as changing execution semantics, data/instruction bus width, etc. In order to adhere to a concrete architectural specification, the SimpleScalar[1] toolkit was picked as the architecture of choice. SimpleScalar constitutes a complete tool chain which utilizes GCC[10] as a compiler for its ISA specification (a 32-bit MIPS derivative with extended 64-bit instructions), and provides several simulators for the binary executable with different levels of complexity, where each simulator models an architectural implementation of the ISA specification.

The SimpleScalar ISA specifies an instruction format as shown in Fig. 5(a), where each instruction has access to four registers and can have three read dependencies and two write dependencies. This is convenient because our implementation approach will require that an instruction generally consumes two operands and a state, and provides a result with an updated state as indicated in Fig. 5(b).



(a) (b)

Figure 5. (a) SimpleScalar instruction format (b) The ALU model.

4.2 Motivation for OFPA

The width of available execution units forces us to break the operands into a series of *packets*. On a 32-bit architecture double precision inputs and outputs are represented via 5 packets: the first packet accounts for the sign and the exponent which is stored in conventional format, followed by four packets, each holding 16 borrow-save digits, requiring 32 bits of storage (the last packet is partially occupied). This is shown in Fig. 6(a).

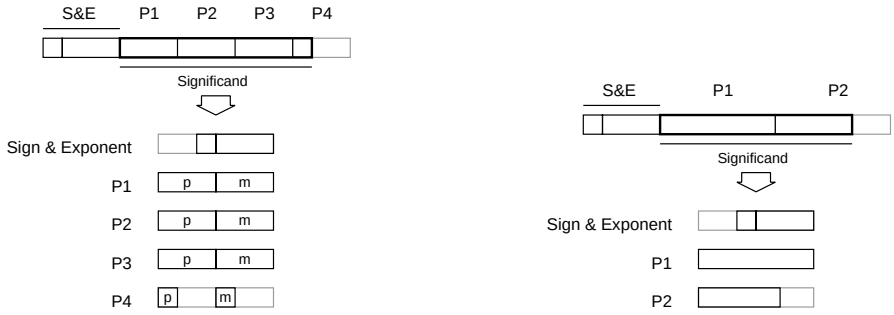


Figure 6. (a) Quantization of double precision type into packets for a 32-bit wide machine implementing OFPA. Note that packet $P4$ is partially filled as there are more digits available to four packets than present in a double precision type (b) Quantization of double precision type into packets for CFPA.

For an ISA based on the CFPA algorithm, a double precision type quantized to packets would require fewer packets than OFPA as the significand would be stored in conventional format, thus requiring less space. The quantization of a double precision type into packets for a CFPA ISA is shown in Fig. 6(b). One may quickly observe that fewer packets would require less time to process, however this would be an inaccurate observation when dealing with dependent operations. In such a scenario an OFPA based design would allow overlapped execution of operations which the CFPA based design would fail to accomplish. In the OFPA design, the most significant packet of the significand is the first to be computed and as a result, normalization can follow immediately; while in the CFPA design, packet $P2$ must be processed before packet $P1$, after which normalization can take place. Because of this feature, an OFPA based design would lend itself better to scheduling of packets with parallel resource availability. In addition to scheduling, an OFPA based design would give an extra level of control on how many packets to process, and thus enable the caller of the operation to state the precision of calculation (at the available quantization level).

4.3 OFPA ISA

Here we present the algorithm for OFPA in terms of an ISA, after which individual instructions will be enumerated and explained. Procedures OFPA-CONV and OFPA-ONLINE perform OFPA via the proposed ISA, where OFPA-CONV has operands in IEEE 754 standard (without quantization into packets) and OFPA-ONLINE's operands are in packet form. In addition, each procedure uses a state value which is consumed and updated as in the ALU model shown in Fig. 5(b).

4.3.1 dbl_extr_exps

Extracts the exponent and the sign of an IEEE double precision operand to form the Sign and Exponent packet as shown in Fig. 7.

4.3.2 ola_exp_add / ola_exp_sub

Compute the exponent difference d, through which the swap state and alignment quantity state are set. In addition set the effective sign of operand B from S_B and the operation to be performed (addition/subtraction)– the initialized state indicates what operation is to be performed. Shown in Fig. 8.

Note: this instruction's name is appended by the operation to be performed. The instruction itself is ola_exp which computes the exponent difference and swap state for the operands. Instruction ola_exp_add sets the

Algorithm: OFPA-CONV(A, B)

setup
$ES^A \leftarrow dbl\_extr\_exps(A)$
$ES^B \leftarrow dbl\_extr\_exps(B)$
$State \leftarrow ola\_exp\_add(ES^A, ES^B)$
$D_1^A \leftarrow dbl\_extr\_dig(A, 1)$
$D_1^B \leftarrow dbl\_extr\_dig(B, 1)$
$D_1, D_1^\star \leftarrow ola\_swap(State, D_1^A, D_1^B)$
$State, R_{Align} \leftarrow ola\_alignf(State, D_1^\star)$
$State, R_{Add}^1 \leftarrow ola\_add(State, D_1, R_{Align})$
$D_2^A \leftarrow dbl\_extr\_dig(A, 2)$
$D_2^B \leftarrow dbl\_extr\_dig(B, 2)$
$D_2, D_2^\star \leftarrow ola\_swap(D_2^A, D_2^B)$
$State, R_{Align} \leftarrow ola\_align(State, D_1^\star, D_2^\star)$
$State, R_{Add}^2 \leftarrow ola\_add(State, D_2, R_{Align})$
$State, R_{Norm}^1 \leftarrow ola\_normi(State, R_{Add}^1, R_{Add}^2)$

if $R_{Norm}^1 \neq 0$

do
$E_t \leftarrow ola\_extr\_exp(State, R_{Norm}^1)$
$R_1 \leftarrow ola\_assign\_res(State, R_{Norm}^1)$
$D_3^A \leftarrow dbl\_extr\_dig(A, 3)$
$D_3^B \leftarrow dbl\_extr\_dig(B, 3)$
$D_3, D_3^\star \leftarrow ola\_swap(State, D_3^A, D_3^B)$
$State, R_{Align} \leftarrow ola\_align(State, D_2^\star, D_3^\star)$
$State, R_{Add}^3 \leftarrow ola\_add(State, D_3, R_{Align})$
$State, R_{Norm}^2 \leftarrow ola\_norm(State, R_{Add}^2, R_{Add}^3)$
$R_2 \leftarrow ola\_assign\_res(State, R_{Norm}^2)$
$D_4^A \leftarrow dbl\_extr\_dig(A, 4)$
$D_4^B \leftarrow dbl\_extr\_dig(B, 4)$
$D_4, D_4^\star \leftarrow ola\_swap(State, D_4^A, D_4^B)$
$State, R_{Align} \leftarrow ola\_align(State, D_3^\star, D_4^\star)$
$State, R_{Add}^4 \leftarrow ola\_add(State, D_4, R_{Align})$
$State, R_{Norm}^3 \leftarrow ola\_norm(State, R_{Add}^3, R_{Add}^4)$
$R_3 \leftarrow ola\_assign\_res(State, R_{Norm}^3)$
$State, R_{Norm}^4 \leftarrow ola\_normf(State, R_{Add}^4)$
$R_4 \leftarrow ola\_assign\_res(State, R_{Norm}^4)$

else

do
$D_3^A \leftarrow dbl\_extr\_dig(A, 3)$
$D_3^B \leftarrow dbl\_extr\_dig(B, 3)$
$D_3, D_3^\star \leftarrow ola\_swap(D_3^A, D_3^B)$
$State, R_{Align} \leftarrow ola\_align(State, D_2^\star, D_3^\star)$
$State, R_{Add}^3 \leftarrow ola\_add(State, D_3, R_{Align})$
$State, R_{Norm}^1 \leftarrow ola\_norm(State, R_{Add}^2, R_{Add}^3)$
if $R_{Norm}^1 \neq 0$

do
$E_t \leftarrow ola\_extr\_exp(State, R_{Norm}^1)$
$R_1 \leftarrow ola\_assign\_res(State, R_{Norm}^1)$
$D_4^A \leftarrow dbl\_extr\_dig(A, 4)$
$D_4^B \leftarrow dbl\_extr\_dig(B, 4)$
$D_4, D_4^\star \leftarrow ola\_swap(State, D_4^A, D_4^B)$
$State, R_{Align} \leftarrow ola\_align(State, D_3^\star, D_4^\star)$
$State, R_{Add}^4 \leftarrow ola\_add(State, D_4, R_{Align})$
$State, R_{Norm}^2 \leftarrow ola\_norm(State, R_{Add}^3, R_{Add}^4)$
$R_2 \leftarrow ola\_assign\_res(State, R_{Norm}^2)$
$State, R_{Norm}^4 \leftarrow ola\_normf(State, R_{Add}^4)$
$R_3 \leftarrow ola\_assign\_res(State, R_{Norm}^4)$
$r_4 \leftarrow 0$

else

do
$D_4^A \leftarrow dbl\_extr\_dig(A, 4)$
$D_4^B \leftarrow dbl\_extr\_dig(B, 4)$
$D_4, D_4^\star \leftarrow ola\_swap(State, D_4^A, D_4^B)$
$State, R_{Align} \leftarrow ola\_align(State, D_3^\star, D_4^\star)$
$State, R_{Add}^4 \leftarrow ola\_add(State, D_4, R_{Align})$
$State, R_{Norm}^1 \leftarrow ola\_norm(State, R_{Add}^3, R_{Add}^4)$
if $R_{Norm}^1 \neq 0$

do
$E_t \leftarrow ola\_extr\_exp(State, R_{Norm}^1)$
$R_1 \leftarrow ola\_assign\_res(State, R_{Norm}^1)$
$State, R_{Norm}^2 \leftarrow ola\_normf(State, R_{Add}^4)$
$R_2 \leftarrow ola\_assign\_res(State, R_{Norm}^2)$
$R_3 \leftarrow 0$
$R_4 \leftarrow 0$

$\vdots$

Algorithm: OFPA-ONLINE(A, B)

setup
$State \leftarrow ola\_exp\_add(ES^A, ES^B)$
$D_1, D_1^\star \leftarrow ola\_swap(State, D_1^A, D_1^B)$
$State, R_{Align} \leftarrow ola\_alignf(State, D_1^\star)$
$State, R_{Add}^1 \leftarrow ola\_add(State, D_1, R_{Align})$
$D_2, D_2^\star \leftarrow ola\_swap(D_2^A, D_2^B)$
$State, R_{Align} \leftarrow ola\_align(State, D_1^\star, D_2^\star)$
$State, R_{Add}^2 \leftarrow ola\_add(State, D_2, R_{Align})$
$State, R_{Norm}^1 \leftarrow ola\_normi(State, R_{Add}^1, R_{Add}^2)$

if $R_{Norm}^1 \neq 0$

do
$E_t \leftarrow ola\_extr\_exp(State, R_{Norm}^1)$
$R_1 \leftarrow ola\_assign\_res(State, R_{Norm}^1)$
$D_3, D_3^\star \leftarrow ola\_swap(State, D_3^A, D_3^B)$
$State, R_{Align} \leftarrow ola\_align(State, D_2^\star, D_3^\star)$
$State, R_{Add}^3 \leftarrow ola\_add(State, D_3, R_{Align})$
$State, R_{Norm}^2 \leftarrow ola\_norm(State, R_{Add}^2, R_{Add}^3)$
$R_2 \leftarrow ola\_assign\_res(State, R_{Norm}^2)$
$D_4, D_4^\star \leftarrow ola\_swap(State, D_4^A, D_4^B)$
$State, R_{Align} \leftarrow ola\_align(State, D_3^\star, D_4^\star)$
$State, R_{Add}^4 \leftarrow ola\_add(State, D_4, R_{Align})$
$State, R_{Norm}^3 \leftarrow ola\_norm(State, R_{Add}^3, R_{Add}^4)$
$R_3 \leftarrow ola\_assign\_res(State, R_{Norm}^3)$
$State, R_{Norm}^4 \leftarrow ola\_normf(State, R_{Add}^4)$
$R_4 \leftarrow ola\_assign\_res(State, R_{Norm}^4)$

else

do
$D_3, D_3^\star \leftarrow ola\_swap(D_3^A, D_3^B)$
$State, R_{Align} \leftarrow ola\_align(State, D_2^\star, D_3^\star)$
$State, R_{Add}^3 \leftarrow ola\_add(State, D_3, R_{Align})$
$State, R_{Norm}^1 \leftarrow ola\_norm(State, R_{Add}^2, R_{Add}^3)$
if $R_{Norm}^1 \neq 0$

do
$E_t \leftarrow ola\_extr\_exp(State, R_{Norm}^1)$
$R_1 \leftarrow ola\_assign\_res(State, R_{Norm}^1)$
$D_4, D_4^\star \leftarrow ola\_swap(State, D_4^A, D_4^B)$
$State, R_{Align} \leftarrow ola\_align(State, D_3^\star, D_4^\star)$
$State, R_{Add}^4 \leftarrow ola\_add(State, D_4, R_{Align})$
$State, R_{Norm}^2 \leftarrow ola\_norm(State, R_{Add}^3, R_{Add}^4)$
$R_2 \leftarrow ola\_assign\_res(State, R_{Norm}^2)$
$State, R_{Norm}^4 \leftarrow ola\_normf(State, R_{Add}^4)$
$R_3 \leftarrow ola\_assign\_res(State, R_{Norm}^4)$
$r_4 \leftarrow 0$

else

do
$D_4, D_4^\star \leftarrow ola\_swap(State, D_4^A, D_4^B)$
$State, R_{Align} \leftarrow ola\_align(State, D_3^\star, D_4^\star)$
$State, R_{Add}^4 \leftarrow ola\_add(State, D_4, R_{Align})$
$State, R_{Norm}^1 \leftarrow ola\_norm(State, R_{Add}^3, R_{Add}^4)$
if $R_{Norm}^1 \neq 0$

do
$E_t \leftarrow ola\_extr\_exp(State, R_{Norm}^1)$
$R_1 \leftarrow ola\_assign\_res(State, R_{Norm}^1)$
$State, R_{Norm}^2 \leftarrow ola\_normf(State, R_{Add}^4)$
$R_2 \leftarrow ola\_assign\_res(State, R_{Norm}^2)$
$R_3 \leftarrow 0$
$R_4 \leftarrow 0$

else

do
$State, R_{Norm}^1 \leftarrow ola\_normf(State, R_{Add}^4)$
$E_t \leftarrow ola\_extr\_exp(State, R_{Norm}^1)$
$R_1 \leftarrow ola\_assign\_res(State, R_{Norm}^1)$
$R_2 \leftarrow 0$
$R_3 \leftarrow 0$
$R_4 \leftarrow 0$

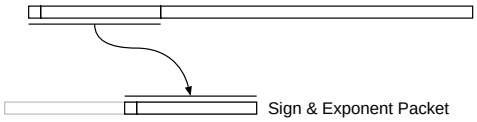


Figure 7. dbl_extr_exps

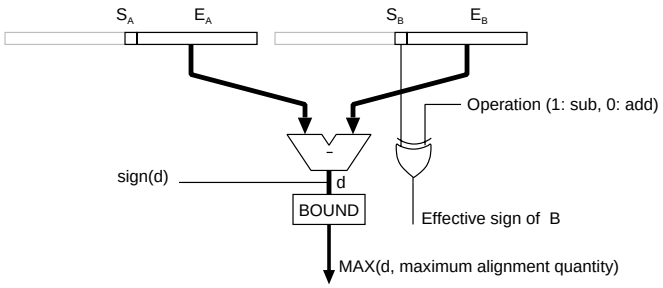


Figure 8. ola_exp_add

appropriate effective sign of operand B when performing addition, i.e. $A + B$, while ola_exp_sub sets the appropriate effective sign for B where the operation to be performed is subtraction $A - B$.

4.3.3 dbl_extr_dig

Extract digit packet i from an IEEE double precision operand. The significand is divided into four packets, where packets $i = 1$ and $i = 4$ are special cases. Packet $i = 1$ will contain the most significant 13 bits of the significand (as p bits, with 13 zero m bits) as shown. The leading three borrow-save digits (BSD, where digit x is represented via two bits $x_p x_m$ with value $x_p - x_m$) could either be $1\overline{11}$ or 000 depending on whether the operand is a denormal. The fourth packet is only partially filled as there are fewer digits than four packets would accommodate. Shown in Fig. 9.

4.3.4 ola_swap

Swap digit packet operands based on the swap state, with the resulting digit packet being the one that is aligned. Shown in Fig. 10.

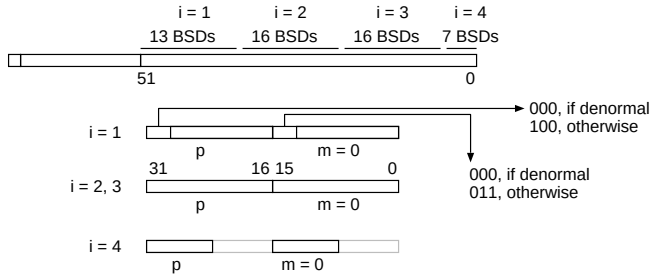


Figure 9. dbl_extr_dig

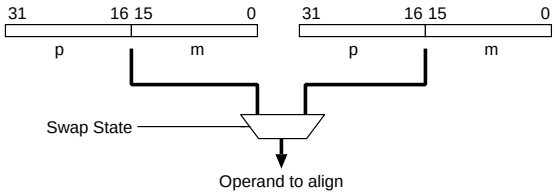
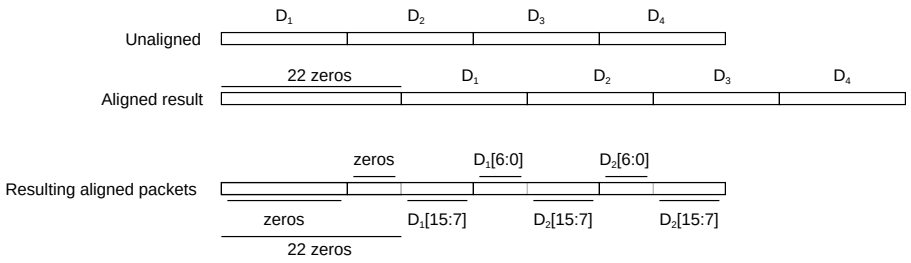


Figure 10. ola_swap

4.3.5 Alignment

Since the significand is in borrow save digit form, alignment constitutes shifting an operand to the right by injecting zeros at the most significant position. This operation is trivial yet expensive for CFPA where a parallel barrel shifter is required. Its equivalent counterpart in OFPA which is implemented as a variable delay is less expensive as only a register chain is required with taps to select the output. However, in both CFPA and OFPA the alignment logic is dependent on the precision of the significand. Even though dealing with packets is slightly more complex, the benefit of this approach is that the hardware requirements are independent of the significand precision and dependent on the size of packets.

Alignment uses two state variables: the amount by which the operand must be aligned (denoted $A_q = |d|$) and whether operand alignment is complete. The following example will be used to illustrate the alignment procedure. Consider an alignment operation with $A_q = 22$; supposing that all of the digit packets were available to us at the time of alignment, we could visualize this alignment procedure as an operation on the digit packets as such,



Since the operand is not aligned and there are more than sixteen zeros, the first result packet of alignment should contain all zeros. For the second packet there is a remaining amount of alignment $A_q = 6$ to be performed which means that the result should be 6 leading zeros, concatenated with the most significant 11 digits of D_1. After this aligned packet is output, the operand has been fully aligned and the alignment state is set. The alignment quantity $A_q = 6$ now serves as a stride which determines that A_q digits of packet D_i and $16 - A_q$ digits of packet D_{i+1} will form the next aligned packet. The third packet is constructed from D_1 and D_2 by concatenating the least significant 6 digits of D_1 to the most significant 11 digits of D_2.

We can therefore generalize this behavior into the following stages,

- If A_q is greater than 16, then the aligned state can not be set. Output 16 zeros as the resulting aligned packet and update $A_q \leftarrow A_q - 16$.
- If A_q is less than 16 and the alignment status is not set, then prepend A_q zeros to the 16 - A_q significant digits of D_1, this would construct the output packet and set the alignment state.
- If the alignment state is set then $16 - A_q$ least significant digits of packet D_i concatenated with A_q digits of packet D_{i+1} constructs the resulting packet.

From this procedure, it can be observed that no more than two digit packets are needed to construct the required aligned packet. Moreover it is desired that the alignment procedure is self contained so that a minimal amount of control is performed at the assembly level. To accomplish this, we observe that until the operand is aligned we would need to pass D_1 and D_2 as inputs to the ALU. Figure 11 shows the different outputs which could be produced for $0 \leq i \leq 16$.

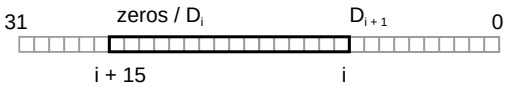


Figure 11.

The design shown in Fig. 12 will perform alignment based on the alignment quantity/stride and output the appropriate result packet.

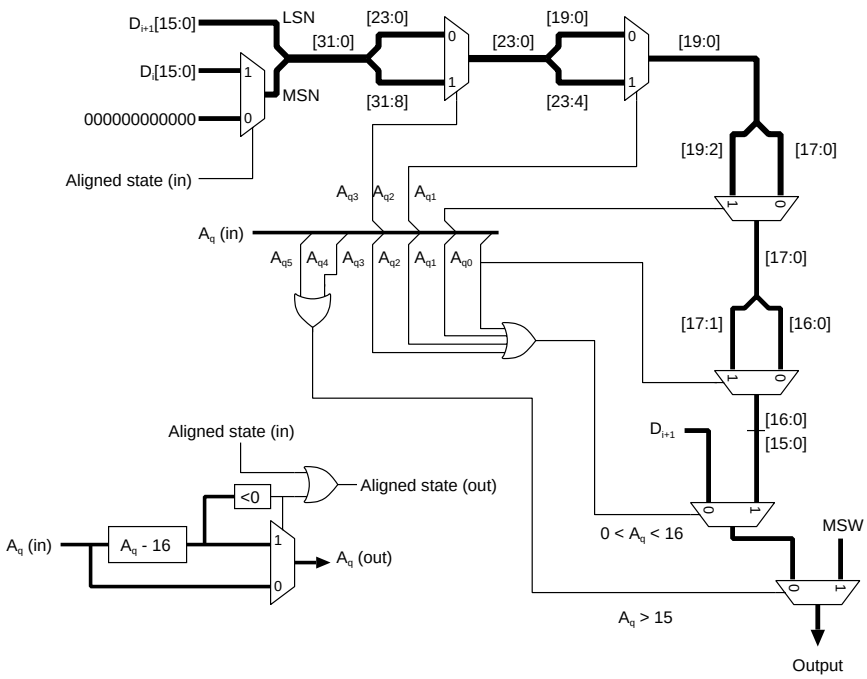


Figure 12. Select a 16 digit window from two operands based on the alignment state and A_q.

4.3.6 ola_add

Perform online addition followed by PN^2 recoding[96].[4] The following operation requires 6 bits for the state which are initially zero. The width-16 BSD addition[18,19] and PN^2 recoder are shown in Fig. 13.

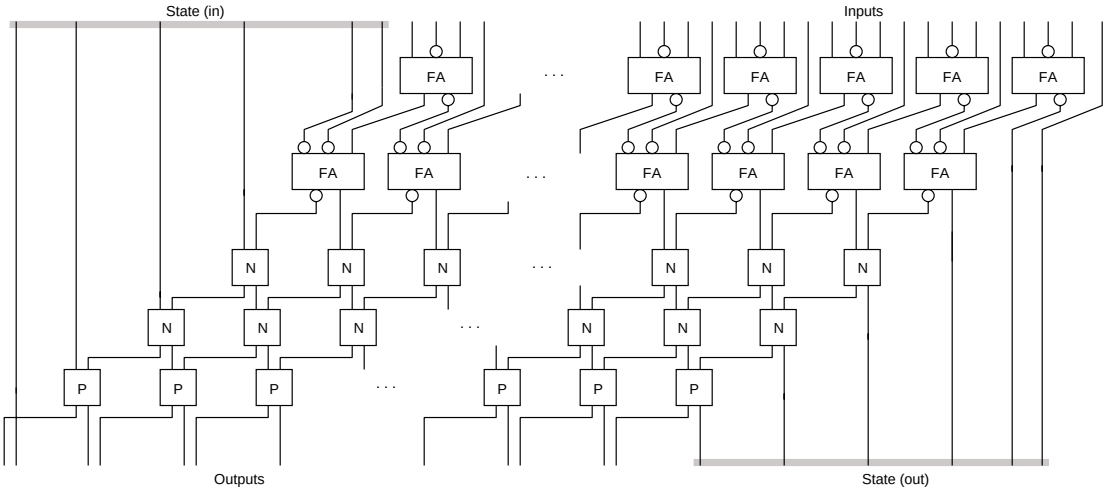


Figure 13. Width-16 online BSD addition (adapted from[18,19]) followed by PN^2 recoding.

4.3.7 ola_normi / ola_norm

If the output of ola_add has still not been normalized, then determine the position of the leading non-zero digit in the packet and adjust the tentative exponent by the amount of leading zeros (or overflow) present. The first result packet obtained from ola_add contains leading digits from the online delay which should not be considered by normalization. The first normalization instruction is therefore different than the rest in order to account for

these digits. In Fig. 14 we demonstrate how the leading digits can be accounted for by forcing their values to zero depending on whether an ola_normi or ola_norm instruction is being executed.

In order to determine the position of the leading non-zero digit, the xor of the BSDs are used as inputs to a priority encoder such that the the output of the encoder signifies the number of leading zeros present in the packet. This is shown by the wire indices, which range from 0 to 31. Moreover, the encoder's output determines the quantity by which the tentative exponent should be adjusted. This value determines what the resulting packet of ola_norm would be: if there are greater than 16 leading zeros then the result would be zero, otherwise, the result would be the 16 digit window starting from the digit *after* the leading non-zero digit (i.e. the leading non-zero digit is not included as it becomes the hidden bit).

We can quickly observe that this is essentially the same functionality provided by the aligner module shown in section 4.3.5 as it selects a 16 digit window based on the number of leading zeros. Therefore, by merging the two designs together, the aligner module can also accommodate the needs of ola_norm.

Finally it is important to observe that a zero value is output if the number of leading zeros present is greater than sixteen since each output packet must contain sixteen digits. As there aren't enough digits to form an output packet, another ola_add must be performed whose result will be padded to the previous packet in order to select a sixteen digit window. As it can be seen in OFPA-CONV and OFPA-ONLINE, a conditional branch checks to see if the result of ola_norm was non-zero, in which case it knows that normalization is complete, and can produce the appropriate output packet; otherwise, another ola_add and ola_norm is performed followed by a subsequent check to determine if the result is normalized.

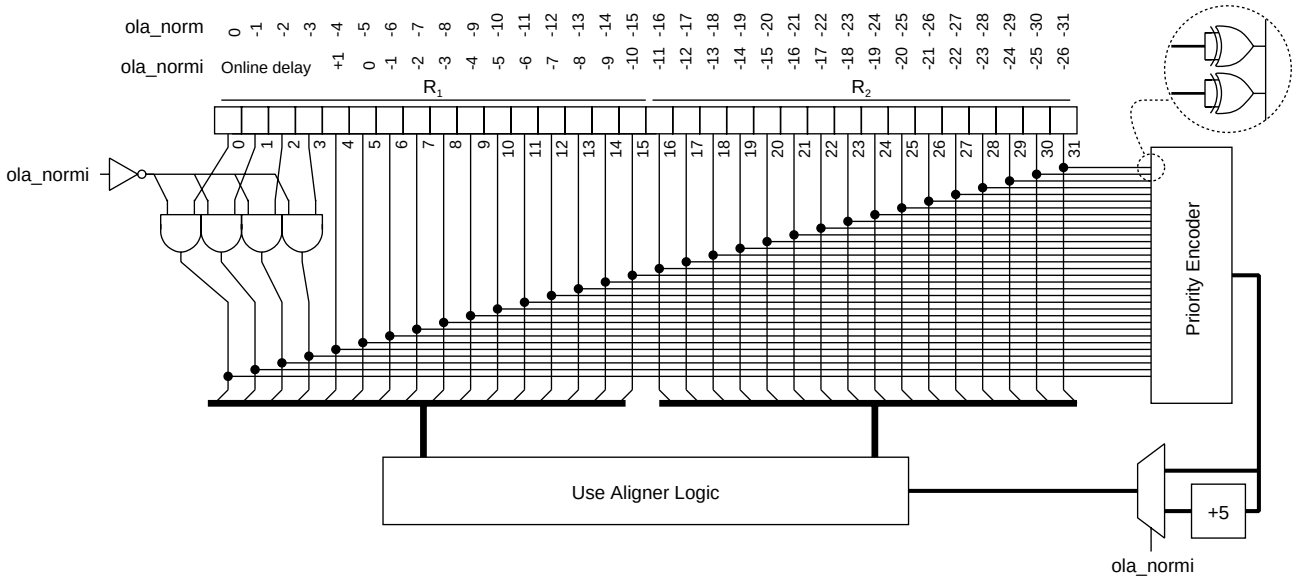
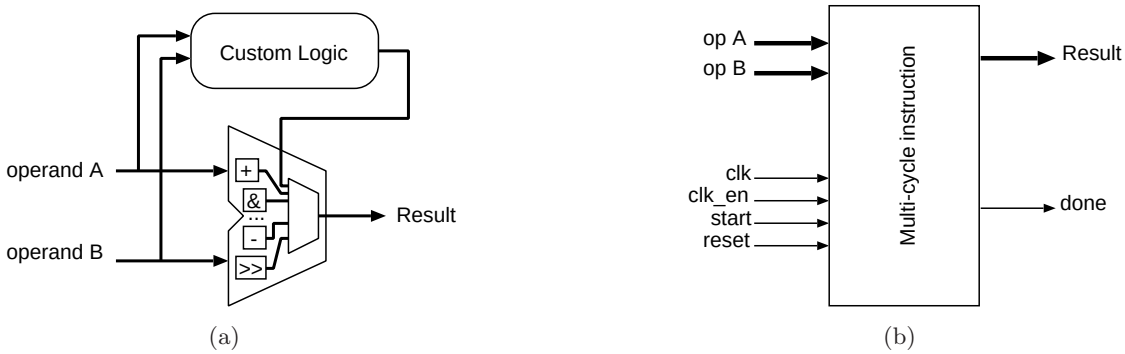


Figure 14. ola-norm

5. RESULTS AND FUTURE WORK

As it was pointed out in section 4.1, the SimpleScalar tool chain was used to prototype the ISA formulation that was outlined in section 4. Although SimpleScalar allows us to develop the discussed algorithm to verify the correctness of the design, it does not provide any measure of hardware cost as each instruction is implemented as a software routine in the simulator. However, integrating the design into a general framework such as SimpleScalar allows us to compare our results to other floating point emulation libraries such as SoftFloat.[11] SoftFloat was ported to SimpleScalar and a program was used to add two numbers (no special cases). Table 1 shows a performance comparison between the two implementations.



(a) (b)

Figure 15. (a) Custom instruction logic for the NIOS II processor. (b) Multi-cycle NIOS II custom instruction interface.

Program	#instructions	#instructions wrt. Baseline
Baseline	6313	-
SoftFloat	6600	287
OFPA ISA Extensions	6347	34

Table 1. Instruction count for executing floating point addition in SoftFloat vs OFPA ISA extensions

In the above table the column "#instructions" denotes the *total* number of instructions that were committed during the simulation. The baseline benchmark is simply a program which returns an integer, as shown below.

```
int main() {
   return 1;
}
```

Since there is some setup and tear down code for an executable, this program gives an idea of how many of the total instructions committed are irrelevant to the floating point add operation. The second column in the table: "#instructions wrt. Baseline" shows how many more instructions were executed with respect to (wrt) the Baseline. We can see that the performance gap is quite large between the two implementations. Of course one should note that SoftFloat is fully IEEE compliant while our ISA extensions still needs to account for special cases and rounding modes.

At this time we are working on implementing the ISA extensions for Altera NIOS II processor[12] which is a softcore processor for Altera's reconfigurable architectures. The NIOS II processor allows the addition of custom instructions to the execution unit as shown in Fig. 15(a) (adapted from [14]). Unlike SimpleScalar, this approach provides the ability to obtain exact hardware costs and performance benchmarks with respect to Altera's floating point execution units and software handlers in addition to SoftFloat. Although the NIOS II execution model does not allow three operand inputs and two result outputs, it does allow multi-cycle operations with a clock qualifier (clk_en) such that all rising edges not pertaining to the instruction can be dismissed. This allows us to embed the state of the operation as registers inside the functional unit. Fig. 15(b) shows the interface for a multi-cycle operation used by the NIOS II processor (adapted from [14]).

6. CONCLUSIONS

In this paper we have investigated an approach which strives to find a middle ground between hardware implementations of floating point addition and software handlers which emulate the functionality strictly using available ALU logic. Our approach shows promising results with respect to the number of instructions committed per floating point add operation. Until the final hardware cost for the ISA implementation is determined, no claim can be made regarding the location of this approach in the design space. Nevertheless, some early implementations indicate that the hardware cost is also promising. The online design approach chosen relaxes data dependencies and provides more flexible instruction scheduling, in addition to providing the user with the ability to control operation precision. These features can be explored as the design matures.

REFERENCES

1. D. Burger, T. M. Austin. "The SimpleScalar Tool Set Version 2.0", Technical Report 1342, Computer Sciences Department, University of Wisconsin–Madison, May 1997. http://citeseer.ist.psu.edu/burger97simplescalar.html
2. C. Bertin, N. Brisebarre, B.D. de Dinechin, C.-P. Jeannerod, C. Monat, J.-M. Muller, S. Raina, and A. Tisserand, "A Floating-Point Library for Integer Processors", Proceedings of the SPIE Conference, 2003.
3. M. D. Ercegovac "On-Line Arithmetic: An Overview". Proceedings of the SPIE, Real Time Signal Processing VII, pp.86-93, 1984.
4. P. Dormiani "On-line floating point addition". UCLA Technical Report 2007.
5. M.D. Ercegovac and T. Lang, Digital Arithmetic, (Chapter 9), Morgan Kaufmann Publishers, San Francisco, 2004.
6. S. D. Krueger andP.-M.l Seidel, "Design of an On-Line IEEE Floating-Point Addition Unit for FPGAs" Proc IEEE FCCM 2004.
7. O. Watanuki and M.D. Ercegovac, "Floating-point on-line algorithms", Proc. of the 5th IEEE Symposium on Computer Arithmetic, 1981.
8. O. Watanuki and M. D. Ercegovac, "Error analysis of certain floating-point on-line algorithms", IEEE Transactions on Computers, C-32:352-358,1983.
9. Daumas, M., Matula, D. W.: "Recoders for Partial Compression and Rounding"; Technical report RR97–01, Ecole Normale Superieure de Lyon, LIP, available at http://www.ens-lyon.fr/LIP. http://citeseer.ist.psu.edu/daumas97recoders.html
10. "GCC, the GNU Compiler Collection", http://gcc.gnu.org
11. "SoftFloat", http://www.jhauser.us/arithmetic/SoftFloat.html
12. "Altera NIOS II processor" http://www.altera.com/products/ip/processors/nios2/ni2-index.html
13. "NIOS II Processor Reference Handbook", http://www.altera.com/literature/lit-nio2.jsp
14. "NIOS II Custom Instruction User Guide" http://www.altera.com/literature/lit-nio2.jsp
15. P. K.-G. Tu, "On-Line Arithmetic Algorithms for Efficient Implementation", PhD Dissertation, Computer Science Department, University of California Los Angeles, 1990
16. P.K.-G. Tu and M.D. Ercegovac. "Gate array implementation of on-line algorithms for floating-point operations", *J. of VLSI Signal Processing*, (3):307–317, 1991.
17. P.K. Tu and M.D. Ercegovac. "Application of on-line arithmetic algorithms to the SVD computation: Preliminary results". *Proc. 10th IEEE Arithmetic Symposium*, pages 246–255, 1991.
18. A.F. Tenca, "Variable Long-Precision Arithmetic (VLPA) for Reconfigurable Architectures", PhD Dissertation, Computer Science Department, University of California Los Angeles, 1998.
19. A.F. Tenca and M.D. Ercegovac. "High-Radix Digit-Slices for On-Line Computations". *Proc. SPIE on High-Speed Computing, Digital Signal Processing, and Filtering using Reconfigurable Logic*, volume 2914, pages 14–25, 1996.