A Decimal Fully Parallel and Pipelined Floating Point Multiplier

Ramy Raafat[1], Amira M. Abdel-Majeed[1], Rodina Samy[1]
Tarek ElDeeb[1], Yasmin Farouk[1], Mostafa Elkhouly[1]
Hossam A. H. Fahmy[2]
[1] SilMinds, LLC. Smart Village, B115, 12577, Giza, Egypt.
[2] Electronics and Communication Department, Cairo University, Giza, Egypt.
ramy.raafat@silminds.com

***Abstract*- Decimal arithmetic is important in several commercial applications including financial analysis, banking, tax calculation, currency conversion, insurance, and accounting. This paper presents a fully parallel Decimal64 floating point (FP) multiplier compliant to IEEE Std 754-2008 for floating point arithmetic. The proposed multiplier possesses novel methods to target low latency. The proposed design is based on a previously published fixed point multiplier [1] that uses a novel BCD–4221 recoding for decimal digits to improve the area and latency of the partial product generation and the partial product reduction tree. Several enhancements are introduced to the design; the final carry propagation adder is implemented using a fully parallel decimal adder with a Kogge-Stone prefix tree, the sticky bit is generated in parallel to the shifter to reduce the critical path delay. The design is extendable to support Decimal128 floating point multiplication. The multiplier is hardware verified for functionality on an FPGA.**

I. INTRODUCTION

Decimal arithmetic received an increased attention in the last decade because of its growing need in many commercial applications and database systems where the binary arithmetic is not sufficient. The arithmetic operations in these applications need to be executed in decimal format. This is because the inexact mapping between some decimal and binary numbers, such as 0.1, cannot be represented accurately using binary format in a limited precision. This leads to an inaccuracy of floating point decimal arithmetic emulation by floating point binary arithmetic units.

Decimal arithmetic software libraries have been developed to overcome the decimal to binary conversion error but they are about 100 to 1000 times slower than what can be implemented in hardware [2]. In the near future, Decimal floating-point (DFP) units are expected to be embedded in many processors' cores to perform the decimal operation faster than the software packages and with higher accuracy. Due to the importance and the growing need of the decimal arithmetic, its specifications are included in the new IEEE standard for floating point arithmetic (IEEE Std 754-2008) [6].

This paper introduces a decimal floating point multiplier based on radix-10 fixed point multiplier [1] that introduced an efficient implementation by the parallel generation of partial products followed by a novel carry save addition (CSA) tree to end the reduction of the partial products in Carry Save (CS) format. This carry save addition tree uses a BCD-4221 recoding for decimal digits to improve the area and latency. In our proposed design, a novel decimal carry propagation adder is used to add the outputs of the carry save addition tree in order to get the intermediate product. Since our design is for floating point multiplier, there is a need to calculate additional information to correctly round the number. The shift amount calculations and sticky counter calculations are executed early to reduce the design latency. The novelty of the proposed design is that it has a low latency and low area compared to previous decimal multiplier designs.

The paper is organized as follows: Section (II) contains background information about the decimal multiplication and an overview on the IEEE Std 754-2008. Section (III) explains in details the proposed multiplier design and highlights the novelty in design. Section (IV) contains testing and synthesis results emphasizing the pipelining results, followed by conclusions in section (V).

II. BACKGROUND

Decimal multiplication performs the computation,

$$P = A \times B \quad (1)$$

Where A is the multiplicand, B is the multiplier, and P is the product. It is assumed that A and B are each n digits hence P is maximally $2n$ digits that must be rounded in order to fit in a limited precision of n digits. Several approaches to decimal multiplication are proposed, the simple and straight forward one is to iterate over the digits of the multiplier B and based on the value of the current digit, add the corresponding multiple of the multiplicand A to an intermediate product. In this approach the multiplier multiples 2A through 9A must be generated which consumes large area and delay. Equation 2 represents this approach to decimal multiplication.

$$X_{i+1} = (X_i + A.B_i).10^{-1} \quad (2)$$

Where X is the partial product, $X_0 = 0$ and $0 \le i \le n\text{-}1$.

Another approach [5] is to generate secondary multiples which are a reduced set of multiples and generate any other missing multiple by adding two multiples from this secondary set based on the value of the current digit of the multiplier B. This approach reduces the complexity of generating eight mul-

tiples using an addition operation. Equation 3 represents this approach to decimal multiplication.

$$X_{i+1} = (X_i + A.B_l^{'} + A.B_l^{''}).10^{-1} \quad (3)$$

Where $A.B_l^{'}$, $A.B_l^{''}$ are the secondary multiples which together equal the proper primary multiple. A parallel decimal multiplication is proposed in [1]; this approach improves the latency of the decimal multiplication and satisfies the concept of pipelining.

The fixed multiplication operation consists of three main stages: generation of partial products, reduction of partial products to two operands and a final carry propagate addition. The proposed design is based on the parallel decimal fixed multiplier in [1] that recodes the BCD8421 multiplier operand into redundant signed digit radix-10 representation for fast and efficient generation of the partial products. It presents also a novel BCD4221 recoding for decimal digits to improve the area and latency of partial products reduction tree.

The IEEE Std 754-2008 specifies DFP formats of 64, and 128 bits [6]. An IEEE Std 754-2008 DFP number contains a sign bit, an integer significand with a precision of n digits, and a biased exponent. The value of a finite DFP number is:

$$D = -1^{Sign} \times Significand \times 10^{E-Bias} \quad (4)$$

Where, E is the biased non-negative integer exponent. Biased exponents in this paper represented by E relate to IEEE Std 754-2008's exponents by:

$$E = e + Bias \quad (5)$$

Where **e** is the unbiased exponent defined in the IEEE Std 754-2008. The significand can be encoded either in binary or in Densely Packed Decimal (DPD), which in the IEEE Std 754-2008 is referred to as the decimal encoding. The exponent must be in the range [E_{min}, E_{max}], after biasing. The IEEE Std 754-2008 introduced representations for special values such as infinity and Not-a-Number (NaN).

The IEEE Std 754-2008 defines the significand of the decimal floating point number as a non-normalized significand. Thus, the decimal floating-point number may have redundant representations. For example, the value of 320×10^{24} may be represented as 320×10^{24}, 32×10^{25}, or 3200×10^{23}. This set of representations for a certain decimal floating-point number is called a cohort. Because of this possibility of multiple representations, IEEE Std 754-2008 defines a preferred exponent for each arithmetic operation, which for multiplication is:

$$PE = E_a + E_b - Bias \quad (6)$$

Where, E_a and E_b are the biased exponents of the multiplicand and multiplier operands, respectively. The multiplier uses the preferred exponent when encoding the result of a multiplication, in order not to have a loss of precision in the output result.

The multiplier design presented in this paper uses the defined Decimal64 numbers with significands encoded in the DPD format. This format has 16 decimal digits of precision (n=16) in the significand, an unbiased exponent range of [−383, 384], and a bias of 398.

III. Proposed Multiplier Design

A. System Overview

The multiplier reads the two operands and extracts the sign, exponent and significand. The significand is decoded from a densely-packed decimal (DPD) encoding into Binary Coded Decimal (BCD). Special values (NaN and Infinity) are detected and handled separately in parallel. The proposed multiplier contains two main paths: Significand path to generate the output significand of the product, and the exponent path to generate the output exponent of the product and the corresponding flags. The proposed design block diagram is shown in Fig. 1.

The fixed point multiplier (FPM) operates once the decoded significands become available. The FPM generates (n+1) partial products in parallel and reduces them to two vectors (sum and carry) using a carry save addition (CSA) tree. These two vectors are added using a novel fast decimal carry propagation adder based on a Kogge-Stone prefix tree.

In parallel with FPM, the exponent of the intermediate product, shift amount and sticky counter are calculated in the Master control (MC) unit. The calculated shift amount is used to align the output product to match the preferred exponent. The calculated sticky counter represents the number of digits to the right of the round digit. The MC unit produces a bit vector (mask) to generate the sticky bit in parallel to the shifter, rather than waiting for its result. This technique improves the latency significantly. The output of the shifter and the sticky bit are introduced to the rounder block which is mainly a decimal incrementer and a multiplexer to fit the result in the required precision. Finally, the output formulation selects either the rounded intermediate product or the special values based on exceptional conditions specified by the MC unit. The output is encoded and formulated to match the IEEE Std 754-2008 format.

B. Fixed Point Multiplier

Fixed point multiplier consists of three main blocks. The partial product generation generates multiples of multiplicand based on the multiplier digits. The CSA reduction tree reduces the generated partial product to two vectors (CS format) to be added. The decimal carry propagation adder adds the output from the CSA tree, generating an intermediate product to be aligned based on the shift amount generated from the MC and then the shifted product is rounded to fit in the required precision of n digits.

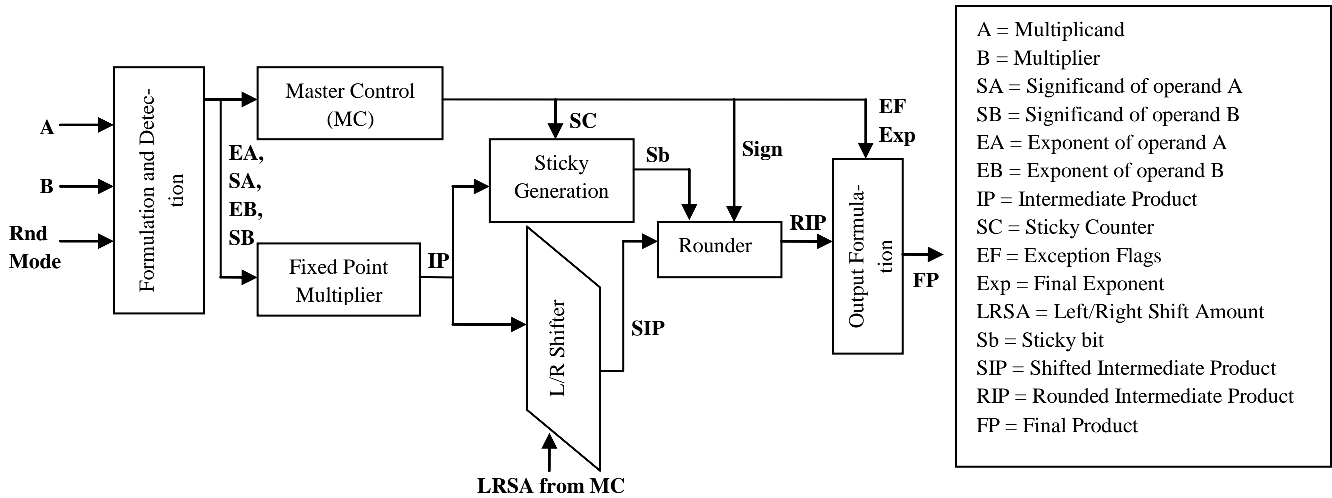

Fig. 1. Block Diagram of DFP Multiplier

The fixed point multiplier design is based on the highly parallel decimal fixed point multiplier presented in [1]. It uses the signed digit recoding technique to generate partial products in parallel. This recoding transforms the multiplier digit set {0. . . 9} into the signed–digit (SD) set {−5. . . 5} to perform the selection of multiples in a similar way as modified Booth recoding.

Decimal multiplicand multiples 2*A, 3A, 4A* and 5*A* are obtained from a few levels of logic using recoding and wired left shifts. The generation of negative multiples is performed by evaluating the 10's complement of positive multiples.

A multiple sets of the multiplicand (A) are generated, concurrently, the multiplier (B) digits are encoded into (n+1) SD radix-10 digits; each recoded digit is used to generate one partial product by selecting one multiple set from the generated sets of the multiplicand. Fig. 2 illustrates a block diagram of the multiplier encoding and the multiple sets selection units.

The generated partial products are reduced using the CSA tree, the main unit in the CSA tree is a 3:2 compressor which takes three partial products and yields two vectors sum and carry, the carry vector must be corrected to be suitable for addition with the sum vector. To reduce the complexity of this correction stage, the partial products are generated in BCD-4221 format.

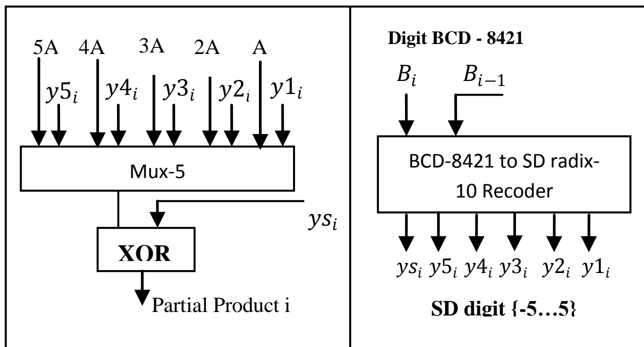

Fig. 2. Multiplier Encoding and Selection Unit

In our proposed design for the DFP multiplier, the output two vectors from the CSA tree are introduced to a novel decimal carry propagation adder to generate the intermediate product. The proposed decimal carry propagation adder is illustrated in Fig. 3.

The carry propagation is based on a Kogge-stone tree that aims to propagate the carry faster. The inputs of the Kogge-Stone tree are Generate and Propagate signals. The Generate signal (Gs) is raised when the sum of the corresponding digits in the input operands is more than 9, and the Propagate signal (Ps) is raised when the sum is exactly 9. As the carry propagation lies in the critical path of the decimal adder, we calculate the Gs and Ps signals early in parallel while computing the sum and sum+1 of each corresponding digits in the input operands. The sum+1 of each corresponding digits is to be selected if the carry from the previous digit is 1. The digits are summed using a four bit binary adder followed by a correction unit to fit the output digit in the range from {0…9}. A straightforward design would have waited for sum digits to

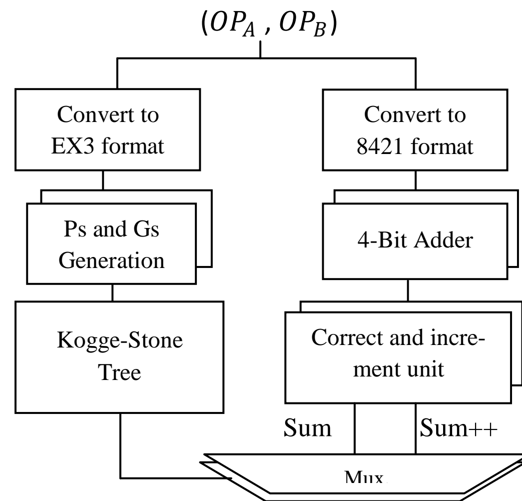

Fig. 3. Decimal Carry Propagation Adder

generate the Gs and Ps signals. In our design, the two input operands are converted to excess3 format and are introduced to a few levels of logic to check if the sum of each corresponding digits is greater than or equal nine to generate the Gs and Ps signals. Then, the Gs and Ps signals are introduced to the Kogge-stone tree. The output of the Kogge-stone tree is a carry vector; each bit in this vector selects the corresponding digit in the output sum. The Kogge-stone tree stages in this design are based on the number of input operand digits not on the number of bits, which improves the latency and reduces the area of the design.

C. Master Control Unit

The exponent of the intermediate product, shift amount and the sticky counter are calculated in the master control unit in parallel with the fixed-point multiplication. The decimal point should be in the middle of the intermediate product which is (2n) length. Thus (n) digits of the intermediate product are to the right of the decimal point. This increases the exponent of the intermediate product (EIP) by (n). The EIP is calculated using (7),

$$EIP = E_a + E_b - Bias + n \qquad (7)$$

The intermediate product result from the fixed point multiplier may need to be left/right shifted to achieve the preferred exponent or to bring the result exponent into the range. The shift amount is determined based on the number of leading zeros in the intermediate product and the difference between the calculated exponent and the minimum and maximum exponent defined in the IEEE Std 754-2008.

The product may need to be left shifted one additional digit after matching the preferred exponent, if a leading zero is detected. Hence, the exponent is updated. Instead of waiting for the intermediate product to count its leading zeros, the latency of the multiplier is improved by determining the output leading zeros based on the leading zeros in the multiplicand and the multiplier. The basic shift amount is calculated using (8),

$$SLA = Min(LZ_a + LZ_b, n) \qquad (8)$$

Where SLA=Shift Left Amount, LZ_a , LZ_b are the leading zeros count in the multiplicand and multiplier respectively.

If the product is a non-zero floating point number with magnitude less than the magnitude of that format's smallest normal number (i.e. the result is a subnormal number), a right shifter is needed to bring the exponent into range even if some precision digits are lost. After shifting the intermediate product, the least (n-2) digits in the intermediate product must be ORed to check if a non-zero digit exist or not, and any shifted digits to the right must be taken into consideration in sticky bit generation. A sticky counter generated from the MC contains the number of digits that must be ORed to generate the sticky bit. To improve the latency of the multiplier, a novel sticky bit generation unit is developed to generate the sticky bit in parallel with the shifter, the sticky counter is used for generating a bit vector of (2n) length; the vector has 1's in bits corresponding to the digits that will be ORed to generate the sticky bit, and 0's in the other bits. The vector is ANDed with the intermediate product then the resultant bits are ORed together to generate the sticky bit. Sticky counter (SC) is calculated early as it depends also on the leading zeros in the multiplier and multiplicand.

$$SC = Max(0,\ n - (LZ_a + LZ_b)) \qquad (9)$$

The sticky counter is decremented twice. This insures that the round and guard digits are not included in the sticky bit generation.

D. Rounding and output formulation

Rounder takes (n+1) digits from the shifted intermediate product (SIP) and the sticky bit. The proposed multiplier supports the five rounding directions listed in the IEEE Std 754-2008 (Round to Nearest Ties to Even (RNE), Round to Nearest Ties Away from Zero (RNA), Round to Positive Infinity (RPI), Round to negative Infinity (RNI), Round Toward Zero (RTZ)). Additionally, it supports two other rounding directions listed in [3]. (Round Away from Zero (RAZ), Round to nearest, Ties Toward Zero (RNZ)). Table I illustrates the rounding scheme used in our DFP multiplier design.

TABLE I
ROUNDING SCHEME

<table>
<tr><th>LSB</th><th>Sticky Bit</th><th>Round Digit</th><th>RNE (+/-)</th><th>RAZ (+/-)</th><th>RPI (+/-)</th><th>RNI (+/-)</th><th>RTZ (+/-)</th><th>RNA (+/-)</th><th>RNZ (+/-)</th></tr>
<tr><td>X</td><td>0</td><td>0</td><td rowspan="3">SIP</td><td>SIP</td><td>SIP</td><td>SIP</td><td rowspan="7">SIP</td><td rowspan="2">SIP</td><td rowspan="4">SIP</td></tr>
<tr><td>X</td><td>1</td><td>0</td><td rowspan="6">SIP+</td><td rowspan="6">SIP+ /SIP</td><td rowspan="6">SIP /SIP+</td></tr>
<tr><td>0</td><td>0</td><td>=5</td><td rowspan="4">SIP+</td></tr>
<tr><td>1</td><td>0</td><td>=5</td><td rowspan="3">SIP+</td></tr>
<tr><td>X</td><td>1</td><td>=5</td><td rowspan="2">SIP+</td></tr>
<tr><td>X</td><td>X</td><td>>5</td></tr>
<tr><td>X</td><td>X</td><td><5</td><td>SIP</td><td>SIP</td><td>SIP</td></tr>
</table>

Legend: SIP =Shifted Intermediate Product, SIP+ = Shifted Intermediate product + 1, LSB=Least Significand Bit of the SIP.

Based on the rounding direction, the product sign, round digit, least significand bit of the least significand bit of SIP and the sticky bit, the rounder selects between the shifted intermediate product truncated to (n) digits and its incremented value.

The output formulation block encodes the significand into DPD encoding format and handles all special values (infinity, Not a Number NaN, overflow and underflow), and generates the exception flags that may be signaled during the multiplication: invalid, overflow, underflow and inexact. The invalid operation exception is signaled when either operand is a signaling NaN or when zero and infinity are multiplied. The default handling of the invalid operation exception involves signaling the exception and producing a quiet NaN for the result, the overflow exception is signaled when a result's magnitude exceeds the largest finite number. Default overflow handling, as specified in IEEE Std 754-2008, involves the selection of either the largest normal number or canonical infinity based on the rounding direction and the raising of the inexact exception. Under default exception handling, the underflow exception is signaled when a result is both tiny and inexact. The system output is in the IEEE Std 754-2008 form.

IV. Verification and Synthesis Results

The multiplier is modeled using RTL VHDL and then it is functionally verified using FPGEN test cases supplied by IBM [4]. Additionally, we generate [7] a large number of random test cases. The results of these random cases are generated using the DecNumber library that implements the general decimal arithmetic specification in ANSI C. This decimal arithmetic library is compliant to IEEE Std 754-2008. The multiplier is synthesized using TSMC 0.18 µm technology. The design is synthesized for a large number of pipeline stages to explore latency, area, and delay tradeoffs. This synthesis was performed using the retiming feature. The synthesis results are given in Table II. The proposed DFP multiplier has a low latency and a small area.Fig. 4 illustrates the relation between the $\boldsymbol{area} \times \boldsymbol{delay}$ product of our design and $\boldsymbol{area} \times \boldsymbol{delay}$ product of the design presented in [3] versus the number of pipelined stages.

TABLE II
Area and Delay for Multiplier Designs

Hardware Design	Delay		Area	
	ns	FO4	µm²	NAND2
Combinational	7.6	84.4	846848.31	84861
1 Stage	7.53	83.6	869870	87169
2 Stages	4.3	47.7	881995	88384
3 Stages	3.2	35.6	946836.5	94881
4 Stages	2.62	29.1	994081.4	99615
5 Stages	2.2	24.4	1160125	118590
6 Stages	1.94	21.5	1187318.6	118980
7 Stages	1.82	20.2	1175210.5	117766
8 Stages	1.65	18.3	1276093	127876
9 Stages	1.6	17.7	1249642	137698
10 Stages	1.52	16.8	1293553.8	146428

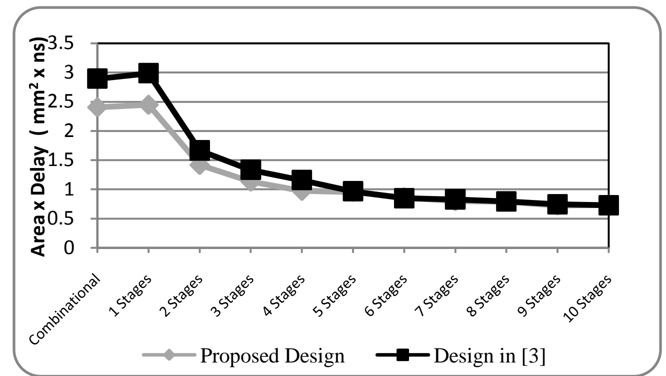

Fig. 4. Area x Delay factor versus No. of pipeline stages

Our combinational design shows about 17% improvement in the $\boldsymbol{area} \times \boldsymbol{delay}$ product over the parallel decimal floating point multiplier presented in [3]. For fair comparison, our results for area and delay are scaled to 0.11 µm technology.

For Decimal128, our decimal floating point multiplier has a delay of 10 ns with 2.4901265 mm^2 area. The proposed floating point multiplier for decimal64 is hardware verified by integrating into NIOS II processor on Altera Cyclone II FPGA development kit. The multiplier is tested with the aid of the NIOS II IDE supplied by Altera and a graphical user interface (GUI) implemented in house.

V. Conclusion

In this paper a decimal fully parallel and pipelined floating point multiplier is presented. Several enhancements are used to improve the latency such as the use of a parallel fixed point multiplier, the generation of the sticky bit in parallel and the use of a fast decimal carry propagation adder. The multiplier is synthesized in 0.18 µm technology and pipelined for different numbers of stages. The multiplier shows very good performance with respect to delay and area. The multiplier is hardware verified through Altera Cyclone II FPGA testing.

References

[1] A. Vazquez, E. Antelo, and P. Montuschi, "A New Family of High–Performance Parallel Decimal Multipliers". Proceedings of the 18th IEEE Symposium on Computer Arithmetic, pages 195-204, June 2007.

[2] M. F. Cowlishaw, "Decimal Floating-Point: Algorithm for Computers", Proceedings of the 16th IEEE Symposium on Computer Arithmetic, pages 104-111, June 2003.

[3] B. Hickmann, A. Krioukov, M. Schulte, and M. Erle, "A Parallel IEEE P754 Decimal Floating-Point Multiplier".

[4] Floating point test suite, IBM Corporation; http://www.haifa.ibm.com/projects/verification/fpgen/ieeets.html

[5] Mark A. Erle, Michael J. Schulte and Brian J. Hickmann, "Decimal Floating-Point Multiplication Via Carry-Save Addition".

[6] "IEEE Standard for Floating-Point Arithmetic" IEEE Std 754-2008, 29 August 2008 (Revision of IEEE Std 754-1985). ISBN 978-0-7381-5753-5.

[7] SilMinds Decimal Tool, SilMinds; http://www.silminds.com/index.php?option=com_content&task=view&id=10&Itemid=37