An Asynchronous IEEE Floating-Point Arithmetic Unit

Joel R. Noche*
Affiliation when work was started and completed:
Department of Electrical and Electronics Engineering
College of Engineering, University of the Philippines, Diliman
joel.noche@up.edu.ph
Present affiliation:
Department of Mathematics and Natural Sciences
College of Arts and Sciences, Ateneo de Naga University, Naga City, Camarines Sur
jrnoche@adnu.edu.ph
Date submitted: July 29, 2005; Date accepted: May 11, 2006

Jose C. Araneta
(deceased)
Department of Electrical and Electronics Engineering
College of Engineering, University of the Philippines, Diliman

ABSTRACT

An asynchronous floating-point arithmetic unit is designed and tested at the transistor level using Cadence software. It uses CMOS (complementary metal oxide semiconductor) and DCVS (differential cascode voltage switch) logic in a 0.35 μm process using a 3.3 V supply voltage, with dual-rail data and single-rail control signals using four-phase handshaking.

Using 17,085 transistors, the unit handles single-precision (32-bit) addition/subtraction, multiplication, division, and remainder using the IEEE 754-1985 Standard for Binary Floating-Point Arithmetic, with rounding and other operations to be handled by separate hardware or software. Division and remainder are done using a restoring subtractive algorithm; multiplication uses an additive algorithm. Exceptions are noted by flags (and not trap handlers) and the output is in single-precision.

Previous work on asynchronous floating-point arithmetic units have mostly focused on single operations such as division. This is the first work to the authors' knowledge that can perform floating-point addition, multiplication, division, and remainder using a common datapath.

Key words: Asynchronous logic circuits, floating point arithmetic, calculation times

**Corresponding author*

INTRODUCTION

Asynchronous circuits, digital logic circuits that do not use a global clock signal, have attracted attention this past decade due to their potential advantages over synchronous circuits (Hauck, 1995), (van Berkel et al., 1999), (Sutherland & Ebergen, 2002). Asynchronous circuits automatically adapt to changing physical conditions, operating faster when the temperature is lower or when the supply voltage is higher. They consume power only when and where performing computations. They allow robust mutual exclusion of signals, making them ideal for handling external inputs. They have better noise and electromagnetic compatibility properties. They exhibit no clock skew, and can thus be designed modularly. They are thus ideal for portable, low-power, wireless applications that are activated by external signals. Asynchronous circuits have been used for some parts of a digital hearing aid (Nielsen & Sparsø, 1999) and a pager (Kessels & Marston, 1999), among others.

Some applications require accurate calculations to be made quickly. Although real numbers can be handled by integer arithmetic hardware (Grehan, 1988), implementing the format known as floating-point in hardware greatly improves performance. A binary floating-point standard proposed by IEEE (1985) is widely adopted, enabling software developers to create easily-portable, highly reliable code. The standard defines four different formats; the one with the least number of bits (32) is called single-precision. Floating-point numbers are represented as $(-1)^S \times F \times 2^E$, where S is the sign bit, F is the significand, and E is the exponent. If the floating-point number is normalized, (i.e., $1 \leq F \leq 2$), then the most significant bit of the significand is always 1 and can be removed to save space (packed). For single-precision, the result is a 23-bit fraction *f*. The signed exponent is encoded as an unsigned number *e* called the exponent field using a bias representation, with a bias of 127 for single-precision (i.e., $E = e - 127$). The standard also defines special quantities: denormal numbers (values which are less than the smallest normalized values), 'Not a Number's (results of invalid operations), positive and negative zeroes, and positive and negative infinities.

Previous work on asynchronous floating-point arithmetic units have mostly focused on single operations such as division (Williams & Horowitz, 1991), (Matsubara & Ide, 1997), (Won & Choi, 2000). The work described in this paper is the first to the authors' knowledge that can handle IEEE floating-point addition/subtraction, multiplication, division, and remainder using a common datapath. To achieve this goal with the available computing resources, we chose to use single-precision arithmetic, with rounding and other operations to be handled by separate hardware or software. To minimize circuit size, division and remainder are done using a restoring subtractive algorithm and multiplication uses an additive algorithm (Noche, 2003). Thus, the architecture is 'serial.' Exceptions are noted by flags (and not trap handlers) and the output is in single-precision.

The datapath uses dual-rail DCVS (differential cascode voltage switch) logic, and the control unit uses CMOS (complementary metal oxide semiconductor) logic. A transistor-level design of the unit using a 0.35 μm process and a 3.3 V supply voltage is designed and tested using Cadence software (IC4.46 package for Sun Solaris 5.8). Virtuoso Schematic Editing and Virtuoso Symbol Editing are used to create the transistor-level schematics. Testing is done using the Affirma Analog Circuit Design Environment.

MATERIALS AND METHODS

Basic Operation

The unit has the following inputs: two 32-bit data inputs (the operands), four arithmetic control (request) signals (one for each operation), four rounding mode control signals, and five flag reset signals. The outputs are: a 32-bit data output (the result), a signal acknowledging correct receipt of the operands *(ain)*, a signal indicating that the output is ready *(aout)*, five flags, and the signals acknowledging their resets.

The external system first makes the operands active, then waits for *ain* to become active. (When the external system later makes the data inputs inactive, *ain* will become inactive.) It then requests the operation to be performed by making the corresponding arithmetic

control signal active. The unit processes the data and any exceptions set the corresponding flags. After the result is computed, *aout* becomes active. The external system then makes the arithmetic control signal inactive. This deactivates the unit, making the result and *aout* inactive. It is now ready for the next set of operands and the next arithmetic operation.

The flags are always active and once set, they remain set until explicitly cleared. They can only be set by the unit, and can only be reset by the external system.

Overview of the Datapath

Figure 1 shows a block diagram of the unit's datapath, with the control unit and some control signals hidden for simplicity. The complete circuit schematics are in Noche (2003). Operands *X* and *Y*, and result *Z* are each composed of a sign bit, an exponent field, and a fraction (e.g., $X = S_X, e_X, f_X$).

The main building blocks of the datapath are the registers and the adders. SR latches are used in the control unit and also for the exception flags. Registers for the exponent calculations do not require any shifting, so SR latches (with completion signals) are also used there. Although one 9-bit register *E* is enough for the addition, multiplication, and division operations, the remainder operation requires two additional 9-bit registers e_1 and e_2 in its initial operand normalization. Significand calculations involve left and right shifting, so significands are stored in shift registers (modified versions of those in Kishinevsky et al. (1994)). *P* is a 25-bit bidirectional shift register connected to *A*, a 24-bit bidirectional shift register with an additional round bit. *B* is a 25-bit shift-left register.

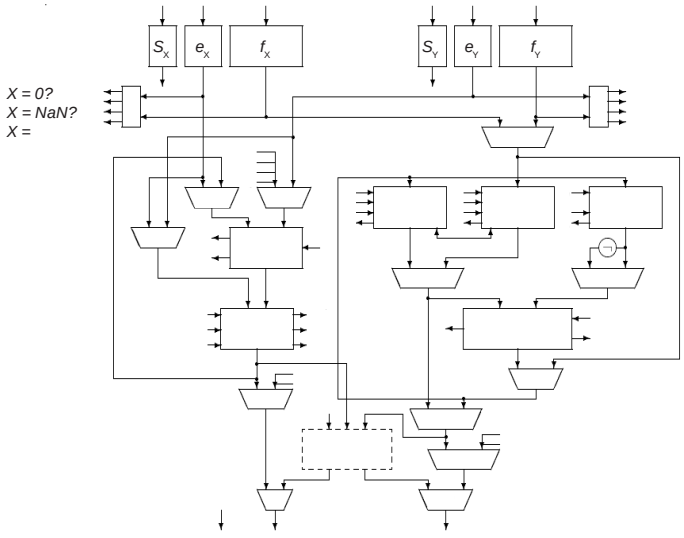

Figure 1. Block diagram of the datapath

A 9-bit carry look-ahead (CLA) adder (Ruiz, 1998), (Ruiz, 2000) is used for exponent calculations, and a 25-bit CLA adder handles all the required significand calculations for all operations. Special signals indicate if the result is zero or -1, and these were used in certain cases as completion signals. For example, the unit's remainder algorithm checks for the case where $e_X - e_Y = -1$.

DCVS multiplexers are used to select the inputs to the registers and adders. The multiplexers do not have completion signals because the blocks their outputs are connected to work correctly whether the outputs are early or late. When two dual-rail signal paths merge into one and both will never be active at the same time, OR gates are used instead of multiplexers to minimize the number of control signals needed. For example, the output of the 25-bit adder is connected to OR gates because the control signals to enable the 25-bit adder and multiplexer 5 will never be high at the same time.

Control Building Blocks

The rest of the building blocks are used in the control unit. These perform handshaking, counting, and conditional branching. The C-element is a basic asynchronous circuit building block, a device whose output changes to a value (logic 0 or 1) only when all its inputs are that value. Aside from C-elements (Shams et al., 1998), SR-latches, and Boolean logic gates, most of the control circuitry of the unit uses building blocks well-suited for four-phase signaling and dual-rail data: decide, do, twice, and thrice. The last three are described by their signal transition graphs (Kondratyev et al., 1998) and implemented as complex CMOS gates using the software PETRIFY (Cortadella et al., 1997). PETRIFY is also used to create the control circuitry for the shift registers, details of which are in Noche (2003).

The decide Building Block

The decide building block has two variants: the decide early and the decide late. Their symbols are shown in Figure 2. Whenever the request *on* becomes active, the first of the inputs *ti* or *fi* to go high decides which of the outputs (*t* or *f*) will go high. Only one of the outputs can go high at any time, and it goes and remains high only when *on* is active. Once the 'decision' is 'made,' it cannot be changed or taken back. For example, if *ti* and *on* go high, then *t* goes high and remains high even if *ti* goes low, or *fi* goes high, or both (as long as *on* remains active).

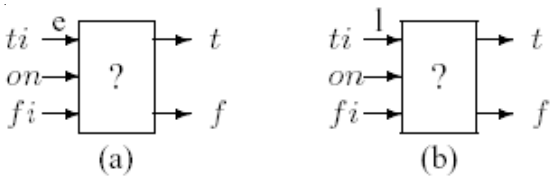

Figure 2. Symbols of (a) decide early, (b) decide late

Transistors driven by input signals that are the last to change are placed nearer the outputs to improve performance. The decide early block assumes that the data input becomes active and valid before *on* becomes active; decide late assumes that the data input becomes active and valid after *on* becomes active. Figure 3 shows the transistor-level schematics of both versions. The inputs *ti* and *fi* (and their corresponding outputs *t* and *f*) are interchangeable.

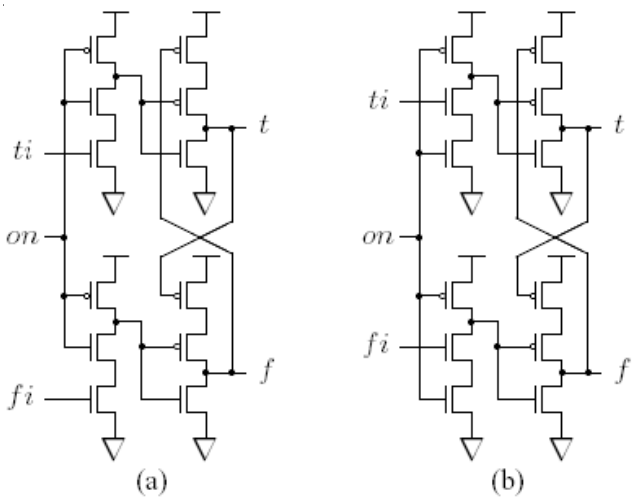

Figure 3. Transistor-level schematics of (a) decide early, (b) decide late

The do Building Block

The do building block has two variants: the do unique and the do guarded. Their symbols are shown in Figures 4a and 4b. The do guarded block is a do unique block with an attached C-element as shown in Figure 4c. Figure 5 shows the STG of the do unique building block. The do unique building block is the same as the Q-element of Martin (1990).

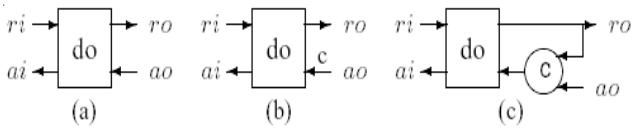

Figure 4. Symbols of (a) do unique, and (b) do guarded; (c) structure of do guarded

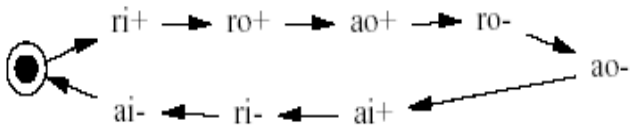

Figure 5. STG of do unique

The twice and thrice Building Blocks

The symbols and STGs of the twice and thrice building blocks are shown in Figures 6, 7, and 8.

A Control Circuit Example

An example illustrates how these building blocks are used to implement algorithms. The algorithm for division is as follows. The sign of the result is the exclusive-or of the operand signs. Special cases (those with operands or results of zero, infinity, or 'Not a Number') are handled first. For ordinary cases, the second exponent is subtracted from the first and the bias is added to this. Register *A* is set to zero. Significand F_X is placed in register *P*, and F_Y is placed in *B*. The

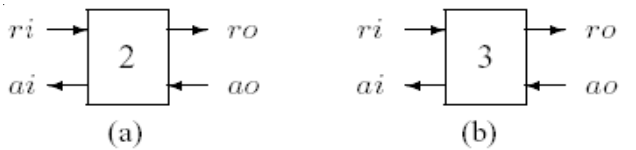

Figure 6. Symbols of (a) twice, (b) thrice

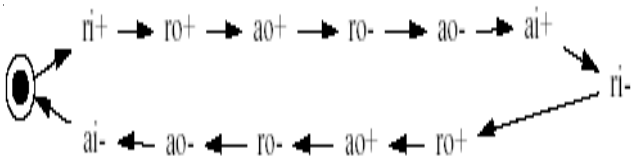

Figure 7. STG of twice

operands are then normalized. Thus, register *B* is shifted left (and the preliminary exponent is incremented accordingly) until its most significant bit is 1. The contents of *P* are shifted left (and the preliminary exponent is decremented accordingly) until the significand there is normalized. The connected registers *P* and *A* are then shifted left 24 times. For each iteration, if the difference *P* - *B* is positive, then it is written to *P* and the rightmost bit of *A* (bit 0) is set to 1. This step is done 25 times. After this, if the significand in *A* is not normalized, then *A* is shifted left once and the preliminary exponent decremented. If the difference *P* - *B* is positive, then it is written to *P* and bit 0 of *A* is set to 1. The result (with the significand in *A*, bit 0 of *A* as the guard bit, and the round bit deduced from *P*) is then passed to the rounding unit.

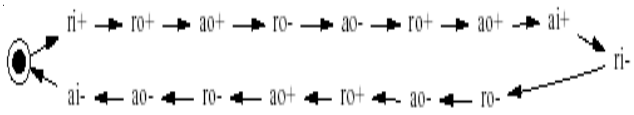

Figure 8. STG of thrice

Figure 9 shows a part of the division algorithm and Figure 10 shows one implementation. The example shows sequential and parallel operations; while-loop, for-to-do-next, and if-then constructs; and how subroutines are handled. As line number *n* in the algorithm is reached, signal *nextn* goes high. When the algorithm is completed, all the *nextn* signals go low in succession.

Note that the circuitry generating signals *did*2 and *did*4 are the same (a 2-input C-element whose inputs are *aB* and *cE*). These two C-elements can be replaced by a single C-element generating the signal *did*2*or*4, which replaces all instances of *did*2 and *did*4. If so, then a transition on *did*2*or*4 would be interpreted as transitions on both *did*2 and *did*4, resulting in incorrect behavior. The do building blocks generating the signals *do*2 and *do*4 should thus be do guarded blocks, so only the block that made the request would be acknowledged.

```
    ⋮
 1  E ← e_X − e_Y; A ← 0; P ← 0, F_X;
 2  E ← E + 127; B ← 0, F_Y;
 3  while (bit(24, B) = 0)
 4      shiftleft B, 0; E ← E + 1;
 5  loop
 6  while (bit(24, P) = 0)
 7      shiftleft P, 0; E ← E − 1;
 8  loop
 9  for dstep ← 1 to 24 do
10      calculate P − B;
11      if (P − B ≥ 0) then
12          P ← P − B; bit(0, A) ← 1;
13      fi
14      shiftleft P, A, 0;
15  next
    ⋮
```

Figure 9. Sample algorithm

RESULTS AND DISCUSSIONS

The unit uses variable-latency algorithms that are implemented using variable-latency circuits. Completion times are thus expected to be data dependent Because limited resources prevented the implementation of the rounding unit, the completion times reported here do not include the time needed to correctly round the result. The algorithms for the four operations and for rounding are in Noche (2003).

What Affects Completion Times

Arithmetic operations start by unpacking denormals and checking for 'Not a Number's. This takes the same time t_{un} for all operations. Next, cases with operands or results of zero or infinity are checked. The different operations have different special cases, so this step takes different times: t_{as} for addition, t_{ms} for multiplication, t_{ds} for division, and t_{rs} for the remainder

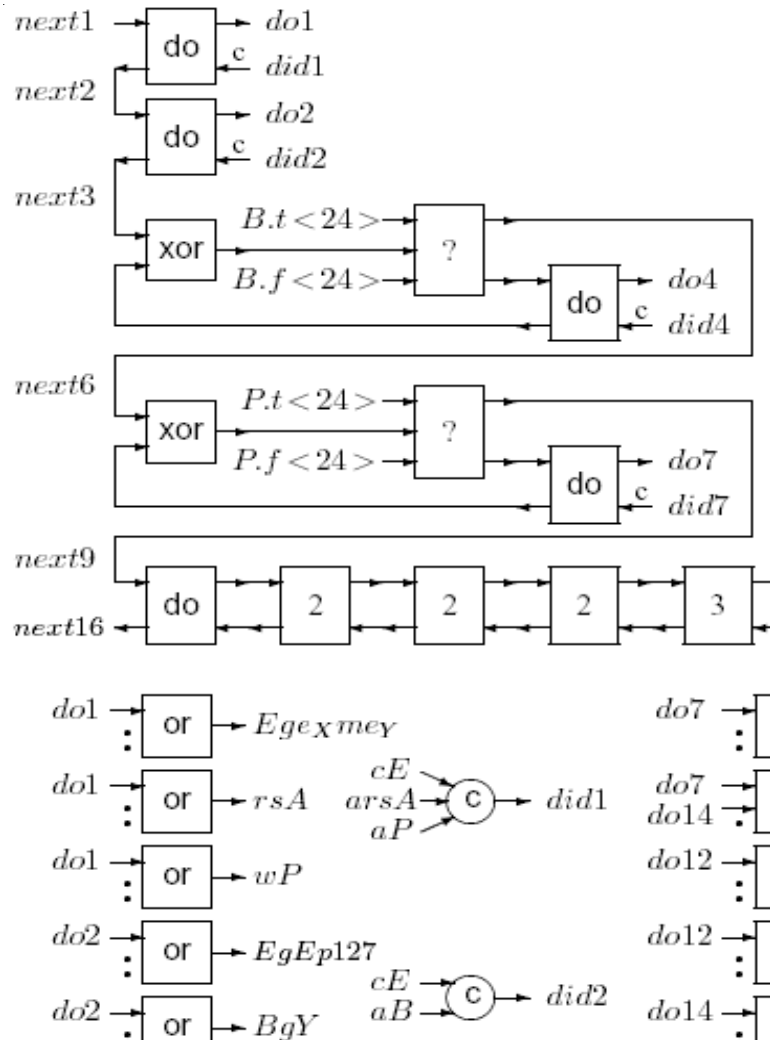

Figure 10. Implementation of sample algorithm

operation. The completion times of special cases are around 8 to 9 ns.

For addition, the next step, the determination of the larger operand, takes a time t_{ad}. Adjusting the significand so that the exponents are equal takes a time dependent on the exponent difference: xt_{aa}, where $x = |e_X - e_Y|$. The last steps take a time t_{al} which is small if the result is zero, and large if the result is negative. But its effect on the completion time is negligible when compared with the effect of the exponent difference. Thus, t_{al} can be approximated as a constant equal to its average value.

For multiplication, the next step is a significand addition for every bit in the first operand's significand that is a 1. This step takes a time $n_X t_{mad}$, where n_X is the number of 1's in F_X. There is also a fixed time taken for shifting the significands, t_{mss}. The last steps take a variable time t_{ml} depending on the result. But since it has little effect on the completion time, it can be treated as a constant.

For division, the next step is to normalize the operands if they are denormal. This takes a time st_n, *where s* is the number of shifts to normalize the operands. There is then a fixed loop, where a register write occurs when the temporary significand of the first operand is greater than or equal to the second operand's significand. This condition is noted by a 1 in the unrounded significand of the result F_Z'. This takes a time $n_Z t_{dsb}$, where n_Z is the number of 1's in F_Z'. Shifting the significands takes a fixed time t_{dss}. The last steps take a variable time t_{dl} depending on the result. Its effect on the completion time is small, and it can be treated as a constant to simplify matters.

For the remainder operation, the next step, normalizing the operands if they are denormal, takes a time st_n. The remaining steps are more complicated, with many conditional branches. When $e_X < e_Y$, the calculation is quick and this can be treated as a special case. When $e_X > e_Y$, the algorithm may go through a loop that executes a shift up to $e_X - e_Y$ times. However, if the significands are multiples of each other, this loop is exited. One easy way to quantify 'being multiples of each other' is to use the difference $r_X - r_Y$, where r_X is the number of bits from the most significant 1 bit of F_X to the least significant 1 bit of F_X, and r_Y is defined similarly for F_Y. If each iteration in the loop takes a time trsb, this step cannot take longer than $(e_X - e_Y)\, t_{rsb}$, but can take a shorter time if $r_X - r_Y < e_X - e_Y$. Thus, this step takes a time zt_{rsb}, where z is either $e_X - e_Y$ or $r_X - r_Y$, whichever is the smaller positive number. When $e_X = e_Y$, the completion time is t_{rl} and depends highly on the significands. The effect of t_{rl} is not negligible, but may be ignored for simplicity. Note that operands having $e_X > e_Y$ also pass through the circuitry for $e_X = e_Y$.

Table 1 shows that the completion times are mostly functions of the number of shifts or of additions. Using improved adders or shifters will greatly shorten the completion times.

Experimental Results

op	detailed	approximate
+	$t_{un} + t_{as} + t_{ad} + xt_{aa} + t_{al}$	$a_a x + b_a$
×	$t_{un} + t_{ms} + n_X t_{mad} + t_{mss} + t_{ml}$	$a_m n_X + b_m$
÷	$t_{un} + t_{ds} + st_n + n_Z t_{dsb} + t_{dss} + t_{dl}$	$a_d n_Z + b_d + c_d s$
rem	$t_{un} + t_{rs} + st_n + zt_{rsb} + t_{rl}$	$a_r z + b_r + c_r s$

Table 1
Expressions for completion times

There are 2^{32} different single-precision floating-point values. Thus there are $2^{32} \times 2^{32} \approx 1.8 \times 10^{19}$ different possible test vectors for each operation. Due to time constraints, only 248 test vectors were simulated. The simulated temperature was 25 ºC and each output was connected to a 5 fF capacitive load. These are typical simulation conditions; higher temperatures and larger loads would lengthen the completion times. The test inputs were all active and valid from the start. A reset pulse was applied to reset all flags, and then the arithmetic control signal was set high. Once *aout* went high, the test circuit set the arithmetic control signal low, causing *aout* to go low. The time from when the arithmetic control signal went high to when *aout* went low is the completion time recorded for the test vector. The selection of test vectors and the results of the simulations are in Noche (2003).

Figure 11 shows the completion time t (in ns) for ordinary case addition as a function of $|e_X - e_Y|$. Figure 12 shows t for ordinary case multiplication as a function

of the number of 1's in the first operand's significand. Figure 13 shows *t* for ordinary case division (no denormal operands) as a function of the number of 1's in the result's unrounded significand (including the guard, round, and sticky bits). Figure 14 shows *t* for ordinary case division as a function of the number of shifts needed to normalize both operands (the test vectors here all have one 1 in F_Z'). Figure 15 shows *t* for ordinary case remainder (no denormal operands, $e_X \geq e_Y$) as a function of *z* (defined in the previous section). Figure 16 shows *t* for ordinary case remainder as a function of the number of shifts needed to normalize both operands (if denormal) (the test vectors here all have $X = Y$ so that $z = 0$). The least squares line is shown for each graph.

Estimation of Completion Times

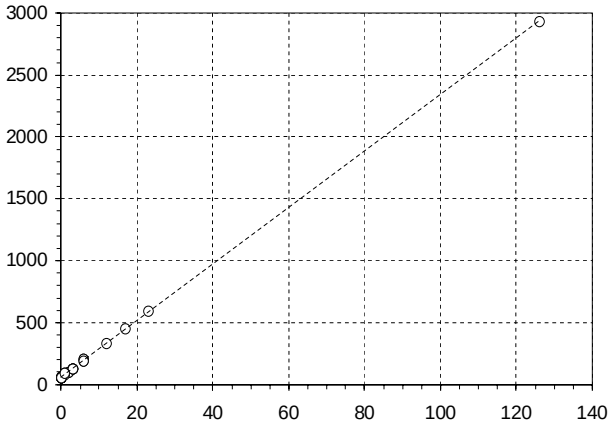

Figure 11. Addition completion times (ordinary cases) in ns as a function of |eX - eY|

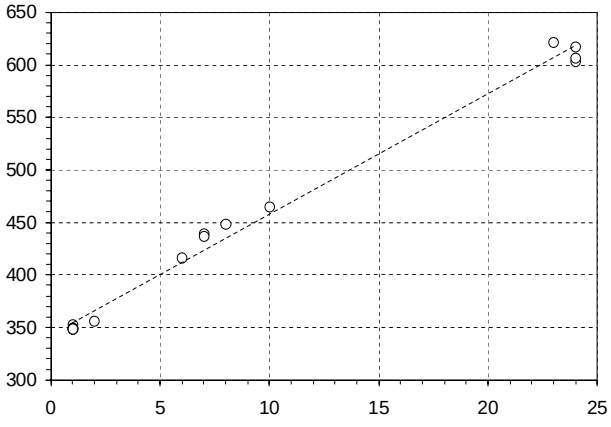

Figure 12. Multiplication completion times (ordinary cases) in ns as a function of the number of 1's in FX

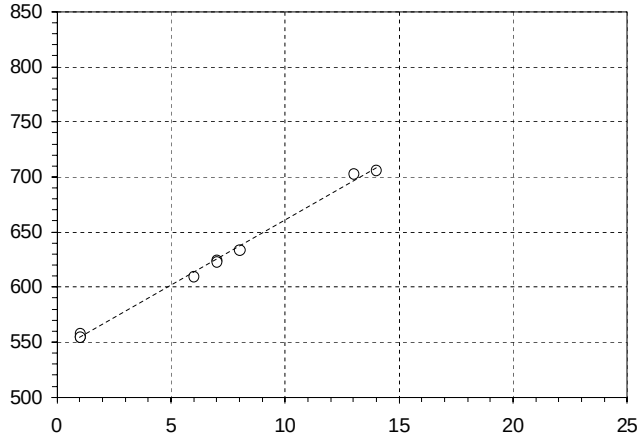

Figure 13. Division completion times (ordinary cases, without denormal operands) in ns as a function of the number of 1's in FZ'

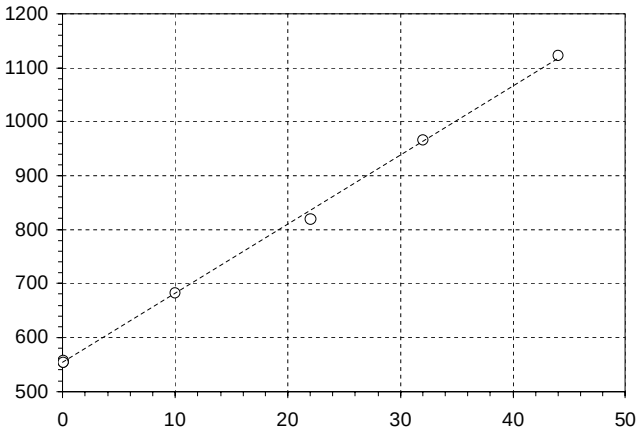

Figure 14. Division completion times (ordinary cases, with one 1 in FZ') in ns as a function of the number of shifts to normalize operands

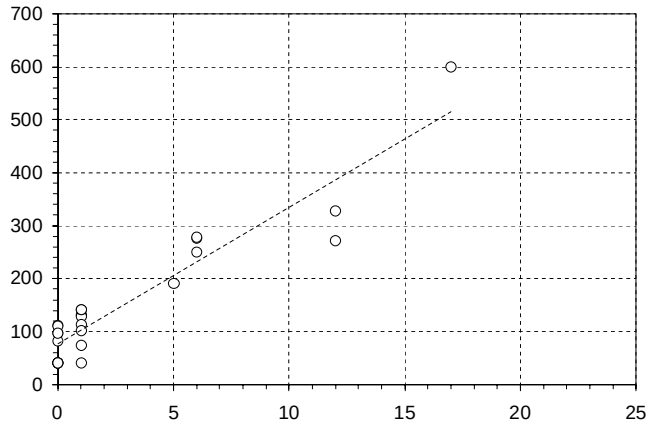

Figure 15. Remainder completion times (ordinary cases, w/ o denormal operands, eX >= eY) in ns as a function of z

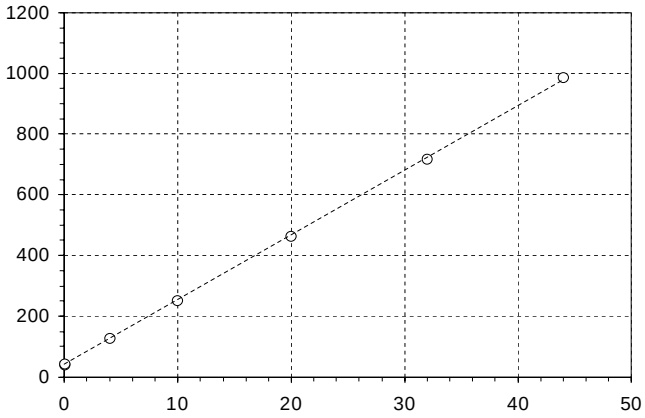

Figure 16. Remainder completion times (X = Y) in ns as a function of the number of shifts to normalize operands

Table 2 shows the predicted completion times *t* and the standard error in the predicted *t* based on the simulation results. For single precision, the absolute value of the exponent difference ranges from $x = 0$ to 254. The approximate range for addition completion times would thus be from 59.0 ns to 5850.2 ns. Cases where x is large are quite rare. In Oberman (1996), ten applications from the SPECfp92 benchmark suite yielded the following distribution for double-precision addition and subtraction operations: around 23% of them had $x = 0$ and around 20% had $x = 1$; 52% of the operations had $x < 3$ and around two-thirds had $x < 6$. Using this distribution, the average addition completion time would be around 127.4 ns.

op	*t* (ns)	standard error
+	$22.8x + 59.0$	5.0
×	$11.5n_x + 342.5$	10.1
÷	$11.8n_z + 542.2 + 12.8s$	5.3
rem	$26.4z + 70.6 + 20.4s$	42.1

Table 2
Predicted completion times

The number of 1's in a single-precision significand ranges from $n_x = 0$ to 24. Multiplication completion times might thus range from 342.5 ns to 618.5 ns. For division, n_z can vary from 1 to 26, while s can vary from 0 to 44, resulting in an estimated range of 554.0 ns ($nZ = 1$, $s = 0$) to 1107.0 *ns* ($n_z = 24$, $s = 22$) for the completion times. For the remainder operation, $0 \le z \le 23$ and $0 \le s \le 44$, leading to a best-case estimate of 70.6 ns ($z = 0$, $s = 0$), and a worst-case estimate of 1126.6 ns ($z = 23$, $s = 22$) for the completion times. Note that the predicted remainder completion time has a relatively large standard error.

While these variable latency algorithms can result in very long completion times, in other cases the times are much shorter. For example, operations with zero operands finish very quickly in this work. This could prove useful in some applications. For example, taking advantage of the relative occurrence of zero-valued discrete cosine transform coefficients in compressed video led to fewer operations and reduced power consumption in (Xanthopoulos & Chandrakasan, 1999).

Power and Energy Consumption

The power and energy consumptions of this work for a few test cases are shown in Table 3.

Test vector	(ns)	(mW)	(nJ)
4195835 + 3145727	79.0	4.08	0.32
4195835 × 3145727	465.1	4.07	1.89
4195835 ÷ 3145727	703.1	3.87	2.72
4195835 rem 3145727	101.5	4.19	0.43

Table 3
Completion times, power consumption, and energy consumption of a few test cases

CONCLUSIONS

An asynchronous single-precision floating-point arithmetic unit is designed and tested at the transistor level using Cadence software. Building blocks well-suited for four-phase handshaking and dual-rail data are used to implement the algorithms. A serial architecture is chosen to keep the design small: only 17,085 transistors are used. Provision for a rounding unit is included, which enables the unit to follow the IEEE 754-1985 Standard for Binary Floating-Point Arithmetic. Due to limited time and resources, the

transistor-level design of the rounding unit is left for future work.

Previous work on asynchronous floating-point arithmetic units have mostly focused on single operations such as division. This is the first work to the authors' knowledge that can perform floating-point addition, multiplication, division, and remainder using a common datapath. The algorithms used in this work are designed to minimize area (and possibly cost) requirements. While current designs focus on improving speed, the recent trend toward mobile devices might make area-efficient designs more attractive.

ACKNOWLEDGMENTS

This study was granted financial support by the Office of the Vice Chancellor for Research and Development-University of the Philippines, Diliman under Grant No. 00010.1 NSET. Louis Alarcón and Anastacia Ballesil provided useful information during the revision of this manuscript. J.R.N. thanks the family of J.C.A. for their support, and the anonymous referees for their suggestions.

REFERENCES

Cortadella, J., M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, 1997. Petrify: A tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *IEICE Transactions on Information and Systems* E80-D(3): 315-325.

Grehan, R., 1988. Floating-point without a coprocessor. *BYTE* 13(9): 313-319.

Hauck, S., 1995. Asynchronous design methodologies: An overview. *Proceedings of the IEEE* 83(1): 69-93.

IEEE Standard for Binary Floating-Point Arithmetic, 1985. New York: ANSI/IEEE Std. 754-1985.

Kessels, J. and P. Marston, 1999. Designing asynchronous standby circuits for a low-power pager. *Proceedings of the IEEE* 87(2): 257-267.

Kishinevsky, M., A. Kondratyev, A. Taubin, and V. Varshavsky, 1994. Concurrent Hardware: The Theory and Practice of Self-timed Design. Chichester, John Wiley & Sons: 368 pp.

Kondratyev, A., J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Taubin, 1998. The use of Petri nets for the design and verification of asynchronous circuits and systems. *Journal of Circuits, Systems, and Computers* 8(1): 67-118.

Martin, A., 1990. Programming in VLSI: From communicating processes to delay-insensitive circuits. In Hoare C. (ed.) Developments in Concurrency and Communication. Addison-Wesley, UT Year of Programming Series: 1-64.

Matsubara, G. and N. Ide, 1997. A low power zero-overhead self-timed division and square root unit combining a single-rail static circuit with a dual-rail dynamic circuit. In *Proceedings of the Third International Symposium on Advanced Research in Asynchronous Circuits and Systems*, Eindhoven, The Netherlands: 198-209.

Nielsen, L. and J. Sparsø, 1999. Designing asynchronous circuits for low power: An IFIR filter bank for a digital hearing aid. *Proceedings of the IEEE* 87(2): 268-281.

Noche, J., 2003. An asynchronous single-precision floating-point arithmetic unit. M.S. thesis, University of the Philippines at Diliman.

Oberman, S., 1996. Design issues in high performance floating point arithmetic units. Technical Report CSL-TR-96-711, Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University, Stanford, California.

Ruiz, G., 1998. Evaluation of three 32-bit CMOS adders in DCVS logic for self-timed circuits. *IEEE Journal of Solid-State Circuits* 33(4): 604-613.

Ruiz, G., 2000. Addition to "Evaluation of three 32-bit CMOS adders in DCVS logic for self-timed circuits". *IEEE Journal of Solid-State Circuits* 35(10): 1517.

Shams, M., J. Ebergen, and M. Elmasry, 1998. Modeling and comparing CMOS implementations of the C-element. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 6(4): 563-567.

Sutherland, I. and J. Ebergen, 2002. Computers without clocks. *Scientific American* 287(8): 46-53

van Berkel, C., M. Josephs, and S. Nowick, 1999. Applications of asynchronous circuits. *Proceedings of the IEEE* 87(2): 223-233.

Williams, T. and M. Horowitz, 1991. A zero-overhead self-timed 160-ns 54-b CMOS divider. *IEEE Journal of Solid-State Circuits* 26(11): 1651-1661.

Won, J.-H. and K. Choi, 2000. Low power self-timed floating-point divider in 0.25μm technology. In *Proceedings of the 26th European Solid-State Circuits Conference*, Stockholm, Sweden.

Xanthopoulos, T. and A. Chandrakasan, 1999. A low-power IDCT macrocell for MPEG-2 MP@ML exploiting data distribution properties for minimal activity. *IEEE Journal of Solid-State Circuits* 34(5): 693-703.