IMPLEMENTATION OF FPGA-BASED RISC FOR LNS ARITHMETIC BY SOFTWARE & HARDWARE

Asst. Lecturer N. H. Abbas
Dept. of Elect. College of Eng. – University of Baghdad
Baghdad- Iraq

ABSTRACT

Field Programmable Gate Arrays (FPGAs) have some difficulty with the implementation of floating-point operations. In particular, devoting the large number of slices needed by floating-point multipliers prohibits incorporating floating point into smaller, less expensive FPGAs. An alternative is the Logarithmic Number System (LNS), where multiplication and division are easy and fast. LNS also has the advantage of lower power consumption than fixed point. The problem with LNS has been the implementation of addition. There are many price/performance tradeoffs in the LNS design space between pure software and specialised-high-speed hardware. This paper focuses on a compromise between these extremes. and on a small RISC core design (loosely inspired by the popular ARM processor) in which only 4 percent additional investment in FPGA resources beyond that required for the integer RISC core more than doubles the speed of LNS addition compared to a pure software approach. This approach shares resources in the data path of the non-LNS parts of the RISC so that the only significant cost is the decoding and control for the LNS instruction. The preliminary experiments suggest modest LNS-FPGA implementations, like the algorithms under consideration, are more cost effective than pure software and can be as cost effective as more expensive LNS-FPGA implementations that attempt to maximise speed.

الخلاصة

ترتيبات بوابة برمجة المجال (FPGAs) عندها بعض الصعوبة بتطبيق عمليات النقطة الطائفة. بصورة خاصة، يكرس عدد كبير للشرائح التي يحتاجها المضروبون فيهم للنقطة الطائفة ويحرم دمج النقطة الطائفة في ترتيبات بوابة برمجة المجال الى اصغر واقل كلفة . خيار اخر هو نظام العدد اللوغاريتمي (LNS) حيث الضرب والقسمة تتمان بسهولة وبسرعة . نظام العدد اللوغاريتمي (LNS) له فائدة اخرى هي استهلاك الواطيء للقدرة مقارنتا بالنقطة الطائفية . المشكلة مع نظام العدد اللوغاريتمي (LNS) هي في بناء عملية الجمع هناك العديد من المقايضات بين السعر والاداء في نظام العدد اللوغارتمي (LNS) وهذا النظام له تصحيحي بين البرامج الصافية واجهزه خاصة سريعة . هذه المنشورة تركز على المساومة بين النهايات التي كتبت عن RISC core الصغير في تصحيحي الخاص (يعمل بشكل طليق بمعالج ARM شعبي) التي استثمار 4% اضافي في مصادر ترتيب بوابة برمجة المجال عن المطلوبة لتصميم RISC core السرعة تتضاعف لجمع في نظام العدد اللوغارتمي (LNS) مقارنة مع طريقة برنامج صافي. هذه الطريقة

التي اخترناها تحصل فيها مشاركة بالمصادر في طريق البيانات لغير جزء من نظام العدد اللوغارتمي RISC
ان الكلفة مهمة في فك الجفرة والسيطرة على ايعاز ال LNS. في تجاربي الاولية اقترحت بناء
FPGAs – LNS معتدل الذي هو اكثر فعالية للكلفة عنه عن البرنامج الصافي واذا مارفعنا الكلفة فانه سوف
تحصل على سرعة اعلى .

KEY WORDS
Addition, ARM, Interpolation, Logarithmic Number System, Low-power Arithmetic, RISC, Verilog.

INTRODUCTION

The Logarithmic Number System (LNS) uses inexpensive hardware for multiplication: an adder [Pal 2000]. This is possible because the sum of logarithms is the logarithm of the product: log(x)+log(y)=log(x y). LNS can be more cost effective and less power-hungry than fixed- or floating-point for multiply-intensive signal-processing applications, including sound and screen computations (constrained by the limited resources of portable communication devices like WAP phones) where moderate accuracy is acceptable [Kadlec].

However, multiplication is not the only arithmetic operation such multimedia applications require. There is usually about an equal mix of addition and multiplication. One approach would be to convert to the logarithmic format only for multiplication, and convert back to conventional fixed-point for the summation [Pan 1999]. This has two drawbacks: two conversions, each requiring a look-up table (LUT), are required at each multiplication, and the resulting fixed-point representations often requires more bits (and correspondingly more power for transmission). In fact, logarithmically-based formats, such as µ-law encoding, have been used in telecommunications for decades because of the compression they afford compared to fixed-point methods like PCM.

For the multimedia and signal-processing applications are interested in, the number of input values given to an algorithm is much smaller than the number of additions and multiplications performed on these values. For example, it might have O($n^{3/2}$) computations for O(n) inputs and outputs. Thus, it is desirable from a power-consumption standpoint to keep data in the more compressed logarithmic format during addition as well as during multiplication, and only convert to fixed-point at the end of the computation.

The problem is that LNS addition also requires LUTs [Kadlec, Waz 1995]. Yet FPGAs, the central component in reconfigurable computing, are rich in LUTs [Sto 1988]. Three ways to implement LNS are listed by increasing speed (and cost):

1- Software running on LUT-based RISC;
2- Hybrid software with some LUT-based hardware dedicated to LNS; and
3- LUT-based hardware dedicated to LNS [Kadlec].

This paper will discuss such FPGA design alternatives using LNS arithmetic. For design reuse between these alternatives, it synthesize the FPGA aspects of the design from a high-level (C-like) notation, known as implicit-style Verilog, using a tool called VITO [Arn 1997] to create the hardware state machines automatically.

It is investigate in this paper the implementation of a conventional CPU inside the FPGA together with some unconventional hardware for LNS. This paper describes using an FPGA to implement a RISC core inspired by a subset of the Advanced RISC Machine's ARM microprocessor [arm]. Rather than simply emulating the ARM, this core is intended to be a platform for an experiment measuring the cost-effectiveness of LNS arithmetic. Thus, it named this project the ARM Work-alike Experiment (AWE) The ARM has been the subject of other academic-design experiments [Woo 1997] and has compiler tools available; also ARM is popular in many multimedia system-on-chip applications where LNS may be useful. Such applications typically need a large memory space for software and data, and thus must assume the logarithm tables required by LNS can fill in what otherwise might be wasted space in a large fixed-size memory chip. AWE is presently targeted for the Virtex-300-FPGA-based VW-300 board from Virtual Computer Corporation. This excellent board has a 1MBx16 external RAM. Since LNS tables occupy an insignificant fraction of this external memory, relatively modest FPGA resources yield numeric speed-up compared to conventional techniques. Putting tables onto the FPGA instead accelerates operations further but at significant-LUT cost. Thus, LNS offers a range of tradeoffs for reconfigurable computing not possible with conventional arithmetic techniques.

AWE reconfigures the meaning of some instructions to assist with LNS implementation. Some flexibility appeared because the processor is implementing only as the configuration of an FPGA. The only constraint is that the instructions under consideration reconfigure should not be ones that are commonly generated by the compiler. For example, the Add-with-Carry (ADC) instruction of the ARM instruction set is infrequently used. It is possible to replace the ADC instruction with a sequence of a few other instructions in the rare instances in which ADC is required. Thus, one option for introducing LNS into our system would be to reconfigure the opcode of the ADC instruction to implement logarithmic multiplication, reducing signed LNS multiply from five cycles to one using insignificant FPGA resources. The ARM instruction set also includes coprocessor instructions, and it will focus on whether it is cost effective to reconfigure this opcode to implement logarithmic addition.

The essential idea with LNS is to convert values into logarithms once and keep them in this representation throughout the entire computation. For example, when a positive value X is input, it is converted into $x = \log_b(X)$. I use capital letters for variables that describe values perceived by the end user, and lower case for the LNS representation. LNS multiplication and division are easy, involving only the addition or subtraction of the logarithmic representation, with some special cases to deal with signs and overflow. (These cases are why reconfiguring the ADC instruction may be desirable.) These special cases are ignored since they have been covered elsewhere [Arn Aug. 1992, King 1971]. Given the LNS representations, $x = \log_b(X)$ and $y = \log_b(Y)$ of the positive values, X and Y, the representation of the product can be formed simply as $x + y$. The difficult part of LNS is the implementation of addition. LNS addition involves the following steps:

1- Obtain $z = y - x$, which corresponds to $\log_b(Z) = \log_b(Y / X)$.
2- Approximate $s_b(z) = \log_b(1 + b^z)$, which corresponds to $\log_b(1+Z) = \log_b(1+ Y / X)$.
3- Obtain $y = x + s_b(z)$, which corresponds to $\log_b(X\,(1+ Y / X)\,) = \log_b(X + Y)$.

The benefit of this algorithm is that it only needs one lookup from a table (step 2), instead of the three lookups for the more natural approach, $\log_b(X + Y) = \log_b(b^x + b^y)$.

Two facts affected early LNS implementations [King 1971]: 1) practical results for many applications can be achieved using low-precision LNS, and 2) prices were low enough that direct-memory lookup could approximate $s_b(z)$ for such low-precision systems. Since its memory requirements grow exponentially with word length, high precision cannot be obtained with direct-memory implementation.

Over one hundred papers [xlns] have described variations on LNS techniques, with many showing how to approximate $s_b(z)$ at lower cost than direct lookup. The most common improvement is that the size of the s_b table can be cut in half [King 1971] (without affecting accuracy) by interchanging y and x so that z is positive because $s_b(-z) + z = s_b(z)$. Since

$$\mathop{\mathrm{Lim}}_{z \to \infty} \quad s_b(z) = z ,$$

thus $s_b(z) \approx z$ for large z, the entire domain of z need not be tabulated [Tay 1988]. Also, interpolation [Arn june1992, Lew 1990, Lew 1994] can increase precision possible with smaller table size than direct lookup.

Power consumption and battery life are important issues in the design of large FPGA systems. Recently, Palourias showed that LNS-based circuits can consume less power than a comparable fixed-point (scaled integer) representation since, on average, the high-order bits of LNS-words exhibit less switching activity [Pal 2000].

LNS is most naturally compared [Arn Aug.1992] against floating-point arithmetic, which is typically larger and more accurate than fixed-point arithmetic. The goal of this project is to implement LNS arithmetic on the AWE in a 32-bit format that is roughly as precise as the 32-bit single-precision floating-point format of the IEEE-754 standard (23 bits of precision). It is considered both a software LNS implementation on a version of the AWE that lacks any special hardware devoted to LNS, and a hardware implementation of LNS for an alternate version of the AWE that consumes only modest additional FPGA resources.

It is also considered how aggressively the designer should pursue high-speed hardware solutions for LNS arithmetic by comparing the modest LNS hardware to a more sophisticated (and expensive) LNS design in the literature [Kadlec] implemented in the same FPGA family as our design.

AWE INSTRUCTION SET ARCHITECTURE

This section describes the non-LNS aspects of the AWE core. The AWE is a 32-bit microprocessor that supports a subset of the ARM's instruction set. It must be chose this subset to be large enough to run the benchmark programs i were interested in and to enable faithful emulation of those ARM instructions that it did not implement in hardware. Like the later versions of the ARM, the AWE supports a full 32-bit address space. (Early versions of the ARM supported a 26-bit address space with the processor's state in the high bits.) The AWE has sixteen general-purpose registers, of which R15 acts as the program counter. Unlike the ARM, the AWE does not have additional registers used for supervisor mode, but instead saves the processor's state in memory. The following describes the binary-compatible instructions of the AWE that behave identically to those of the ARM and describes those ARM instructions not implemented in hardware on the AWE.

The primary class of instructions for the AWE is the data-processing instructions. Like any RISC processor, these instructions operate on two operands with the result going into a third register. For example,

```
ADD R1,R2,R3
AND R4,R5,15 ROR 2
SUB R6,R7,R8 LSL 7
```

There are 16 such AWE instructions, either involving only an addition/subtraction or a Boolean operation. The last operand can be a (possibly rotated) 8-bit immediate value or a register (possibly shifted/rotated a fixed distance). Unlike the ARM, the AWE does not support shifting or rotating by a variable distance, but, as explained below, the AWE has provisions for software emulation of ARM instructions not implemented in hardware.

The second class of instructions on the AWE is the multiply/accumulate instruction:

```
MUL R1,R2,R3
MLA R1,R2,R3,R4
```

The latter instruction is the only AWE instruction implemented in hardware that processes four operands. Like early versions of the ARM, the AWE only multiplies unsigned 32-bit values producing only the low-order 32 bits of the product.

The third class of instructions on the AWE is for relative branch instructions:

```
B  label
BL label
```

The branch-and-link instruction (BL) saves a return address in R14. The ARM lacks a halt, but it is useful for testbenches. I have defined a branch back to itself (eafffffe) as the halt for the AWE.

The final class of instructions on the AWE is the load/store instructions:

```
 LDR R1,[R2,4]
 STR R3,[R4],-4
```

The AWE supports pre- and post-increment and decrement modification of the index register by an unsigned 12-bit constant. The AWE also supports pre-indexed addressing without modification of the indexed register. Unlike the ARM, the AWE does not support modification of the index register by another register.

The AWE supports conditional execution of instructions, based on four bits of the program-status register (negative, zero, carry and overflow). These bits are optionally set by data-processing or multiply instructions. The sixteen conditions supported include signed and unsigned inequality.

The AWE does not support multi-register transfer, swap or supervisor mode instructions. The non-INS version of the AWE does not support the coprocessor instructions and raises an exception if a program attempts to execute such an instruction.

Although the AWE does not support special supervisor mode instructions, it does have a primitive supervisor mode used for unimplemented instruction traps and external interrupts. The way in which the AWE supervisor mode processes interrupts is completely different from the way the ARM supervisor modes process interrupts.

In the ARM, an interrupt causes a subset of the registers to be switched for a bank of supervisor registers. For "FIQ" interrupts, R8-R14 are switched, with R14 containing the return address. In the other four interrupt modes, R13-R14 are switched. So, in total, there are 16+(14-8+1)+4*(14-13+1) = 31 ARM registers, of which only 16 are available to the software at any instant. Although this makes the ARM well suited for context switching, the complexity of this scheme makes the hardware realisation of this on an FPGA undesirable.

Instead, the AWE uses a minimalist technique borrowed from the classically elegant PDP-8 [Bell 1971]. On that machine, an interrupt causes the return address to be saved at a fixed location in memory and execution to proceed from the location following the return address with the interrupt flag disabled. Interrupts can only occur when the interrupt flag is enabled. The interrupt flag provides a semaphore that controls writing to that fixed location. The PDP-8 returns from the interrupt service routine by turning the interrupt flag back on and doing an indirect jump through the fixed location in memory.

In the AWE, an interrupt causes the program counter, R15, to be saved at a fixed location in memory and execution to proceed from the following location. Because the AWE implementation is pipelined, the value of the R15 at that moment is somewhat offset from the correct return address, but the correct address can be computed from the information saved in memory. The AWE instruction set includes load instructions with relative addressing (pre-indexed R15 without modification). When R15 is loaded by such an instruction, the effect is identical to a jump indirect. For example, if the following AWE code is located so that the label UR15 is at the address where the hardware saves the user's R15 and ISR is the label where the hardware resumes execution after the exception:

```
UR14              ;saved user R14
UR15              ;AWE saves user R15 here
ISR  STR R14,[R15,-16] ;save user R14 in UR14
  LDR R14,[R15,-16] ;get UR15 into R14
  SUB R14,R14,12    ;adjust ret addr for pipe
  STR R14,[R15,-24] ;ret addr to UR15

  LDR R14,[R15,-32] ;restore UR14 into R14
  LDR R15,[R15,-32] ;indirect jump to UR15
             ;LDR R15 supervisor off
```

Execution proceeds at the label ISR; the interrupt will be processed; and the user's R14 wil saved by the software in UR14. Of course, a realistic service routine would have more details a place indicated by the ellipsis (which must be empty for the offsets to be correct here). To exit user's state is restored (in this case, just the restoration of UR14 into R14 is shown). The fina to resume execution of the user's code is the LDR R15. The AWE has a feature of the LDR R instruction not present on the ARM: the LDR R15 instruction turns supervisor mode off on AWE. This feature causes no problems with a user-mode program having an LDR R15 instru This feature does mean that LDR R15 can only be used in AWE supervisor mode for the purpos returning to AWE user mode, as shown above.

Because this return address scheme is non-reentrant. AWE interrupts (and unimplem instruction traps) can only occur in user mode. All supervisor software must be restricted to th instructions supported by the hardware. In order to support ARM supervisor mode softwa should be possible to write a small kernel that runs ARM supervisor modes under AWE user m

VERILOG CODING

The design was done in the implicit style of Verilog [Arn 1999], which allows easy codin register transfers in an Algorithmic State Machine (ASM). It has a register file with two read and one write port. The register file is simply declared as reg[31:0]r[15:0]. At present, chosen a pipeline depth of 3 stages (instruction fetch, instruction decode, execution) as was do early versions of the ARM . For example, the execution stage for the following:

```
    ADD R1,R2,R3
    SUB R4,R1,1
```

will cause the Verilog non-blocking assignment, r[1] <= `NCLK r[2]+r[3], to execute in one (reading r[2] and r[3] in the same cycle that the sum is written back into r[1]). Then, in the cycle (when r[1] contains the sum) the non-blocking assignment r[4] <= `NCLK r[1]-1 will exe again causing two reads and one write. It is may decide to increase the pipeline depth in or increase the clock frequency, but my initial experiments suggest that a depth of 3 stages is usa to at least 25MHz. (The B instruction is natural [Arn 1999] for a pipeline depth of 3, althoug versions of the ARM, such as the StrongArm [Mon 1997], went to a pipeline depth of 5 in or operate above 200MHz.)

The starting point for our design of the AWE was the tiny textbook example of an ARM [Arn 1999]. That example was intended to be an illustration of the concepts of pipelined exe and it only supports ADD, SUB, MOV and B instructions and the N bit in the program register. That example does not implement the logical, compare, shift, load, store, multi subprogram instructions. That example assumes that the program counter is physically part register file and that the ARM could be regarded as a Harvard architecture. The load, multiply and subprogram instructions use multi-cycle implementation on the AWE (as on the ARM). My reconfigured LNS instruction also uses a multi-cycle implementation.

Like the ARM, the AWE is a Princeton architecture, with the same memory used to sto programs and data. I made the implementation choice that there is only one port to the memory. Because of the one-port memory, LDR and STR instructions on the (as on the ARM) take multiple cycles (one to fetch the instruction, another to calculate the ef address and a third to access the data). R15 is not the actual program counter on the AWE. instructions that modify R15 (such as the LDR R15 above) cause the AWE to copy the ne from R15 back into a separate program counter in an extra state that only occurs for R15 similar way, the BL instruction takes an extra cycle to save the return address in R multiply/accumulate on the AWE performs the multiply of two register operands and inserting an appropriate ADD instruction in the pipeline to fetch the fourth operand. In the implementation is similar to microcode. The features of the AWE listed in the require multiple states to implement. The ease with which the implicit style allows design of a complex-state machine is an important factor. Unlike a pure multi-cycle implementation, the AWE must enter and leave these special states aware of the contents of the pipeline, and this complicates the state machine considerably.

The other version of the AWE that is augmented with an LNS-addition instruction also uses multi-cycle implementations, similar in complexity to the integer multiply/accumulate. It was possible to integrate this quickly into the AWE because of the convenience of the implicit style of Verilog.

SOFTWARE IMPLEMENTATION

Linear interpolation computes $s_b(z_H) + c(z_H) \cdot z_L$ as an approximation for $s_b(z)$, where $c(z_H)$ is the slope of an interpolation line and $s_b(z_H)$ is obtained from a table in RAM. Here i split z into two disjoint components so that $z = z_H + z_L$, where z_H is the high portion of z used to access the table and (note that $0 \leq z_L < \Delta = 2^{-N}$) is the low portion which is multiplied by the slope. The division between z_H and z_L occurs N bits after the radix point. It is do not considered partitioning [Bell 1971, Arn 1999], which is a more complicated form of interpolation in which Δ varies depending on z.

There are several alternative forms of interpolation, which differ in how $c(z_H)$ is defined and in how much accuracy it can guarantee for the result . For example, choosing $c(z_H) = s_b'(z_H)$ gives $2N + 3$ bits of accuracy. Instead, it will be use Lagrange interpolation, which gives $2N + 5$ bits. The linear-Lagrange approach computes $c(z_H)$ as $(s_b(z_H + \Delta) - s_b(z_H)) / \Delta$. Thus a choice of $N = 9$ gives 23 bits of accuracy, which is roughly what IEEE-754 provides, leaving 23 - 9 = 14 bits for z_L. (Lewis Lew 1994] argues for better-than-floating-point accuracy, which can be achieved with larger table and guard bits.) Here is the AWE code for the LNS addition algorithm without using any LNS-specific instructions:

```
SUBS  R2,R2,R1       ;R2=z=y-x
ADDMI R1,R1,R2        ;if(x>y){ R1=y
RSBMI R2,R2,0x00     ;      z=|z| }
CMP   R2,0xcf ROR 12  ;if z is big
BCS   L               ; skip interpolate
MOV   R4,R2 LSR 14   ;R4=zH=z>>14
SUB   R2,R2,R4 LSL 14 ;R2=zL=z-(zH<<14)
ADD   R6,R3,R4 LSL 2  ;R6=addr + (zH<<2)
LDR   R5,[R6],0x004   ;R5 = sb(zH)
LDR   R4,[R6],0x000   ;R4 = sb(zH+Dt)
SUB   R4,R4,R5 ;c(zH)=(sb(zH+Dt)-sb(zH))/Dt
MUL   R6,R4,R2        ;R6 = c(zH)*zL
ADD   R2,R5,R6 LSR 14 ;R2 = sb(z)
ADD   R0,R1,R2        ;R0 = min(x,y)+sb(z)
```

Assuming R1 contains x, R2 contains y and R3 contains the starting address of the $s_b(z_H)$ table, the conditional instructions (ADDMI and RSBMI) put the absolute value of their difference (z) into R2 and the smaller of the two of them into R1. The compare and branch instructions avoid interpolation when z is outside of the domain in which the function needs to be tabulated. The following seven instructions implement the interpolation formula. The final ADD combines the interpolated approximation for $s_b(z)$ with the minimum of x and y, forming the logarithm of the sum of X and Y. On the AWE, the LDR instruction takes three cycles, and the 32-bit integer multiply takes 36 cycles. (In order to simplify its implementation, the AWE does not exit early on multiplication in the way the ARM does—the testing of the 32-bit word would slow the cycle time in our FPGA implementation.) The other eleven instructions are single cycle. The total time for the LNS-addition software on the AWE is 53 cycles.

FPGA IMPLEMENTATION

An advantage of an FPGA is that its functionality can be reconfigured to optimise operations that are important for the application at hand. In this case, the LNS-addition algorithm (including interpolation) can be transformed from the above software into equivalent Verilog, making LNS addition part of the instruction set of the AWE. This has the potential to speed up the operation. FPGA implementation allows some steps that were done sequentially in §4 to proceed in parallel. For example, the summing of $s_b(z_H)$ to the minimum of x and y occurs simultaneously with the fetching of $s_b(z_H + \Delta)$. Also, the first three instructions are reduced to one or two cycles in the hardware implementation as shown in the following implicit Verilog code:

```
...
else if (ir1[27:24] == 4'b1110)
 begin
  ir2 <= `CLK{12'hf05,ir1[19:0]};
  //NOPed SUB --same ops as LADD
  ...
  @(posedge sysclk) `ENS;
  t <= `CLK `PC;
  minreg <= `CLK ir1[3:0];  //Y
  maxreg <= `CLK ir1[19:16];//X
  z <= `CLK aluout; // X-Y
  ir2 <=`CLK{12'hf06,ir2[19:0]};//RSB
  if (aluout[31]) //msb from SUB
   begin  //Y>X, use RSB aluout
    @(posedge sysclk) `ENS;
     maxreg <=`CLK ir1[3:0];  //Y
     minreg <=`CLK ir1[19:16];//X
     z <= `CLK aluout; //Y-X
   end
 ...
 end
```

Here, `ENS indicates Entering a New State, ir1 is the instruction register for the decode stage of the pipeline (ir1[19:16] points to the register that contains X and ir1[3:0] points to the register that contains Y), ir2 is the instruction register for the execute stage of the pipeline (ir2[25:20] determines which data-processing operation the AWE's ARM-compatible ALU performs: 05 is subtract, X-Y, and 06 is reverse subtract, Y-X), aluout is the output from that ALU, and minreg and maxreg are bit pointers to registers that contain min(X, Y) and max(X, Y), respectively. The above code inserts SUB and RSB instructions into the instruction register to obtain z. It takes an extra cycle when the roles of X and Y need to be interchanged in order to make z positive. Together with Verilog shown above, it takes seven or eight cycles outside of the multiplication for LADD to compute. Since z_L only needs 14 bits, the multiply in the interpolation can stop after 14 cycles. Thus the total time for the Logarithmic-Add instruction (LADD) is either 21 or 22 cycles.

Unlike the actual coprocessor instructions of the ARM, LADD on the AWE (coded in the coprocessor group, 1110) accesses the processors' general-purpose registers. (A few additional internal registers that are not accessible to the programmer, like minreg and maxreg, are also used.) To simplify the design of LADD, it was assumed that the destination is a different register than the registers that contain X or Y, and it was also assumed that either of operand registers may be used by LADD as a scratchpad. LADD chooses the one that contains the larger of the operands (pointed to by maxreg) as the scratchpad, leaving the minimum value unchanged for use at the end of the logarithmic addition algorithm. (Such assumptions are possible because this is an FPGA-RISC implementation, where optimisations may be shared between the processor and its software—a luxury not possible for conventional processors.)

Both designs were synthesised for the Virtual Computer Corporations' VW-300 board, which uses the Xilinx Virtex-300 FPGA. This FPGA has 3,072 logic slices.

Table 1. Comparison of Implementations
(Assuming these are placed and routed in the same V300 chip)

	AWE no LNS	AWE with LNS	LNS/noLNS ratio	HSLA (ALU only)	HSLA ALU + AWE
MHz	27	25	0.92	17	17
Cycles	53	21-22	0.4	8	8
States	19	27	1.4	n/a	n/a
Slices	2,471	2,560	1.04	2,325	4,796
FPGA utilisation	80 %	83 %	1.04	75 %	won't fit
Flip-flops	784	850	1.08	n/a	n/a
LUTs	3,651	3,875	1.06	n/a	n/a
Gates	35,114	37,045	1.05	n/a	n/a

(n/a = not available)

The Cycles indicate how many clock cycles are required to perform the logarithmic addition operation. The States are the total number of states in the state machine that controls the hardware. LUTs are the internal lookup tables used as the basic component of the FPGA. A slice consists of LUTs plus other logic and flip-flops. The equivalent gates are those reported by the Xilinx synthesis tool, and should be viewed as only a hypothetical estimate of the complexity of the design.

The 53 cycles for the software implementation does not include one cycle to initialise R3 to contain the address of the table. This cycle is not needed in the hardware implementation because, unlike a commercial VLSI processor, an FPGA processor can be resynthesised to customise the table address for a particular software program. This is one of the advantages of the reconfigurable approach—A CISC instruction like LADD need not be quite so complex because i can make some simplifying assumptions.

The LNS-addition aspect of the AWE shares many resources with its non-LNS-aspects. The marginal cost of implementing LNS addition is only 2,560 - 2,471 = 89 slices because of this resource sharing. These slices are mostly devoted to implementation of the extra states of the algorithm.

The present design does not implement subtraction. Although for the same accuracy, subtraction takes more of the external memory than addition[10], the algorithmic complexity of subtraction is similar to addition. It can thus estimate that at most another 89 slices would be required for subtraction, yielding a total of 2,560 + 89 = 2,649 slices

COMPARISON WITH OTHER LNS FPGAS

There are a few other reports in the literature of FPGA implementations for LNS arithmetic with which the AWE might be compared. Wazlowski et al. [Waz 1995] report much more limited-precision use of LNS than that proposed here in a re-configurable platform specialised for hidden-Markov speech recognition.

Kadlec et al. [Kadlec] report a 32-bit LNS ALU, with comparable precision to the design considered here. It is based on a design promoted by the HSLA project [col 2000], and like my design, has the logarithm tables residing off-chip. Kadlec synthesized this for a larger member of the same family of FPGAs used here, and thus can be compared to my design. (A more recent version of Kadlec's design uses Virtex-E part, and thus cannot be compared directly to my design.) The available data for Kadlec's original design [Kadlec] is shown in the right column in Table 1 above. The clock frequency is roughly two-thirds of that in my design. Kadlec appears to use a significant portion of the resources in a fast integer multiplier for quadratic interpolation, and yet given the limitations of the FPGA, is only able to achieve 8-cycle operation. It should be remembered that Kadlec only implements an ALU—there is no processor mentioned to control its operation. A fairer comparison is one between my AWE and Kadlec's ALU plus a processor. Since he reports no processor, let us assume that he is using a processor of the same size as the non-LNS AWE. Since his ALU and processor stand alone from each other, this combination would require 2,325 + 2,471 = 4,796 slices.

The LNS AWE can achieve 25/21.5 = 1.16 MFLOPs using no more than 2,560 + 89 = 2,649 (including the estimate for subtraction) slices. Kadlec's ALU with a processor could achieve 17/8 = 2.125 MFLOPs using no more than 4,796 slices. A reasonable figure of merit to compare Kadlec against the LNS AWE is MFLOP/slice. This is roughly $4.4 \cdot 10^{-4}$ for either system. Thus, Kadlec is no more cost effective than the LNS AWE. In contrast, my non-LNS AWE with software LNS has a lower figure of merit: $2 \cdot 10^{-4}$. Thus i conclude that it pays to move from software to hardware reconfiguration (which is done easily within my V300 FPGA), but there is no additional gain in developing a system as complex as Kadlec (which would require a larger, more expensive FPGA).

CONCLUSIONS

In this paper the results shown that a modest investment in FPGA resources, on top of what is required for a minimal integer-RISC processor, allows significant improvement in the implementation of LNS arithmetic. For the particular example of 32-bit LNS, an increase of about 4 percent of the FPGA's resources allows a speedup of about 2.5 for logarithmic addition. This improvement is possible because a significant amount of the resources required can be shared with the non-LNS RISC core. In contrast, an earlier attempt [Kadlec] to make a faster LNS ALU comes at a much higher FPGA cost. Since the justification for LNS must be stated in terms of cost-effectiveness, this preliminary experiment with AWE suggests that a faster LNS implementation [Kadlec] is no more cost effective than an economical implementation (like mine). Since my design takes half the FPGA resources (in a similar RISC-processor context), my design can be used on smaller, less expensive FPGAs, such as the Virtex-300 used in our experiment.

Were able to conduct this experiment rapidly because of the convenience of the implicit style of Verilog, which allows efficient multi-cycle state machines to be coded in a natural algorithmic form. The enhanced preprocessor described in the following appendix (VITO 1.4) enables us to use implicit Verilog to produce a one-hot state machine that is accepted by a conventional synthesis tool (in my case, Xilinx's WebPack). VITO is available for download [Arn 1997], as is WebPack [Xil 1999].

APPENDIX. ENHANCEMENTS TO VITO

In order to synthesize the AWE, i had to extend the semantics of my VITO preprocessor beyond the previously published [Arn 1997, Arn 1999] specifications to cope with memories. As an example of this extension, let's consider something much simpler than the AWE. Here is a two-state nonsensical machine specified in implicit Verilog (the macros `ENS and `CLK are explained elsewhere [Sto 1986]):

```
   reg [31:0] a;
```

```
reg [31:0] data1, data2;
always
 begin
  @(posedge clk) `ENS;
   a <= `CLK data1;
  @(posedge clk) `ENS;
   a <= `CLK data2;
 end
```

The labels on the left correspond to wires in a one-hot controller. The previous version (1.2) of VITO creates a one-hot controller and a corresponding datapath that implements the algorithm specified in implicit-style Verilog. Here the controller has two states, one for each @(posedge clk)`ENS. The datapath has only the one register, a, in this example. For the above implicit Verilog source code, all versions of VITO (including the improved version 1.4 described here) translate this into a one-hot controller, with two outputs, whose names are based on the statement numbers of the original Verilog (s4 and s6 here), as shown in **Fig. (1):**

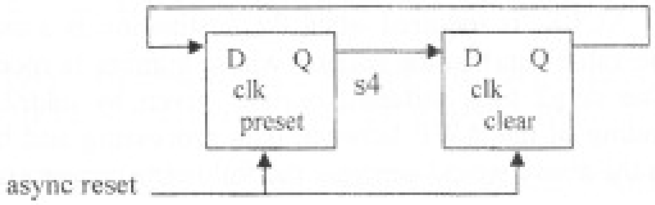

Fig (1). A two-state one-hot controller.

The asynchronous reset makes sure that the flip flop for the starting state contains a one in the first clock cycle and the other flip flop(s) contain zero(s). (An additional flip flop involved in the reset is not shown.) Control statements, such as if or while, would cause the corresponding one-hot controller to be more complicated. The outputs of the controller are used to tell the datapath what to do. The difference between older versions of VITO and the new version used here is in the datapath. VITO 1.2 generates the datapath by extracting all the non-blocking assignments (sorted by destination). These then specify continuous assignment(s) to wire(s) (whose names derive from the concatenation of "new_" to the destination register):

```
wire [31:0] new_a;
assign new_a = s4 ? data1 : s6 ? data2 : a;
```

This corresponds to a series of two input multiplexors, as shown in **Fig. (2).**

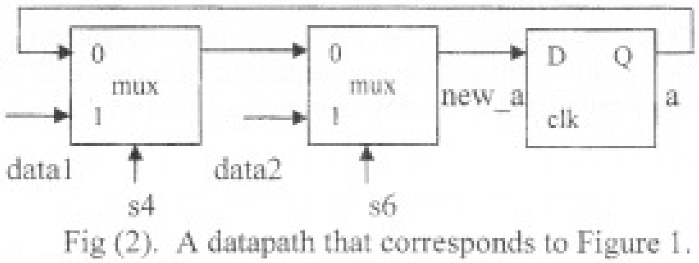

Fig (2). A datapath that corresponds to Figure 1.

The wire (new_a) is the Q input to the destination register (whose D output is a in this case):

```
reg [31:0] a;
always @(posedge clk)
```

```
  a <= new_a;
```

Although this works adequately for simple designs up to the complexity of accumulator-based general-purpose computers [Arn 1999], this datapath-generation technique is not powerful enough to handle the Verilog coding for the register file of a RISC processor in a correct fashion. For example, different addresses may be used to access the register file in different states:

```
   reg [31:0] r[15:0];
   reg [31:0] data1, data2;
   reg [3:0] addr1,addr2;
s1: always
s2:  begin
s3:   @(posedge clk) `ENS;
s4:     r[addr1] <= `NCLK data1;
s5:   @(posedge clk) `ENS;
s6:     r[addr2] <= `NCLK data2;
s7:  end
```

For simulation, a different macro, `NCLK, is required when the destination is a memory, such as r[addr1]. The first state assigns the value data1 to the register whose number is specified by addr1. The second state assigns the value data2 to a different register given by addr2. For instance, situations like this occur in the coding of the AWE between data-processing and branch-and-link instructions. Using VITO 1.2 with the above would generate the following erroneous code:

```
 wire [31:0] new_r[15:0];
 assign new_r[addr1]=s4 ? data1 : m[addr1];
 assign new_r[addr2]=s6 ? data2 : m[addr2];
```

This is illegal since Verilog does not allow an array of wires. In order to overcome this restriction, we developed a new version (1.4) of VITO that generates the datapath in a new way:

```
 reg [31:0] r;
 ...
 always @(posedge clk)
  begin
   r[addr1] <= s4 ? data1 : r[addr1];
   r[addr2] <= s6 ? data2 : r[addr2];
  end
```

The semantics of the non-blocking assignment allow these separate assignments to be grouped together into a single always block. This coding style is compatible with the IEEE P1364.1 Verilog synthesis standard, and should be synthesizable by any commercial tool that accepts only explicit style.

REFERENCES

M. Arnold, T. Bailey, and J. Cowles, (1992), Comments on 'An architecture for addition and subtraction of long word length numbers in the logarithmic number system, IEEE Trans. Comput., 41, pp. 786-788, June.

M. Arnold, T. Bailey, J. Cowles, and M. Winkel, (1992), Applying features of IEEE 754 to sign/logarithm arithmetic, IEEE Trans. On Comput., 41, pp.1040-1050, Aug..

M. Arnold and J. Shuler, (1997), A preprocessor that converts implicit style Verilog into one-hot igns, 6th International Verilog HDL Conference, Santa Clara, CA, pp. 38-45, March 31-April 3,. e www.verilog.vito.com for more recent versions.

M. Arnold, (1999), Verilog Digital Computer Design: Algorithms into Hardware, PTR Prentice Hall, Upper Saddle River, NJ,.

C. G. Bell and A. Newell, (1971), Computer Structures: Readings and Examples, ch. 5, McGraw-Hill, New York, NY,.

J. N. Coleman, E. I. Chester, C. I. Softley, and J. Kadlec, (2000), Arithmetic on the European garithmic Microprocessor, IEEE Trans. Comput., 49, no. 7, pp. 702-715, July.

King 1971] N. Kingsbury and P. Rayner, (1971), Digital Filtering Using Logarithmic Arithmetic, Electron.Lett.,7, pp.56-58, Jan.

Kadlec et al., LNS ALU core for FPGA, http://www.utia.cas.cz/idealist-east/vilach/sld001.htm

J. M. Lewis, (1990), An architecture for addition and subtraction of long word length numbers in e logarithmic number system, IEEE Trans. Comput., 39, pp. 1325-1336, Nov..

J. M. Lewis, (1994), Interleaved memory function interpolators with application to accurate LNS ithmetic units, IEEE Trans. Comput., 43, pp. 974-982, Aug..

Montanaro, et al., (1997), A 160-MHz, 32-b, 0.5-W CMOS RISC microprocessor, Digital chnical Journal, 9, No. 1,. See also www.intel.com/design/strong/.

Paliouras and T. Stouraitis, (2000), Logarithmic number system for low-power arithmetic," TMOS 2000: International Workshop on Power and Timing Modeling, Optimization and ulation, Gottingen, Germany, 13-15 September, pp. 285-294,.

e Programmable Logic Data Book, Xilinx, San Jose, (1999), See www.support.xilinx.com for rmation on WebPack.

Stouraitis, (1986), Logarithmic Number System Theory, Analysis, and Design, PhD Dissertation, versity of Florida, Gainesville, pp. 122-124,.

F. J. Taylor, R. Gill, J. Joseph, and J. Radke, (1988), A 20 Bit logarithmic number system cessor, IEEE Trans. Comput., C-37, pp. 190-199,

Wazlowski, A. Smith, R. Citro, and H. Silverman, (1995), Performing log-scale addition on a ributed memory MIMD multicomputer with reconfigurable computing capabilities, Proceedings the 1995 International Conference on Parallel Processing, pp. III-211 - III-214,.

Woods, P. Day, S. B. Furber, J. D. Garside, N. C. Paver, and S. Temple, (1997), AMULET1: synchronous ARM microprocessor, IEEE Trans. on Comput., 46, No. 4, pp. 385-398, April.

] www.arm.com.

] www.xlnsresearch.com.

Pan et al., (1999), A 32b 64-matrix parallel CMOS processor, IEEE International Solid-State its Conference, San Francisco, pp. 15-17, Feb.