Little Elm Architecture

© 2018 Brian McMillin - - DRAFT 2018-04-12

Abstract

This paper describes a Single Instruction Multiple Data (SIMD) computer architecture. Novel features include a sequential-read addressing scheme and branchless conditionals. These two features ensure that all reads (code and data) are anticipated and cached. This just-in-time approach ensures that all memory reads are handled in a timely manner.

We introduce the concept of *processing lanes* corresponding to the data access and logic operations performed by each of the parallel processors. These parallel processors are specialized in their functionality and are referred to as *Binary Logic Units* (BLUs). In addition to parallel operations, stepping through a set of very compact states for each lane during each instruction cycle allows the creation of *virtual lanes* that are handled sequentially by each of the BLUs. The use of virtual lanes allows the apparent parallelism to be increased beyond the scope of the physical hardware.

Arithmetic extensions to the basic BLU design allow for compatible floating-point operations to be performed in a power-efficient manner using very minimal additional hardware complexity.

Contents

Abstract
Contents
Background
SIMD Observations
Philosophy

- Memory Addressing
- Memory Reads
- Memory Writes
- Memory Cycle Controller

Branchless Conditionals
Lookup Tables and Case Statements
Memory Organization
Program Loader
Concerning Subroutines
Instruction Set
I/O: Host Setup and Retrieval
Binary Logic Unit (BLU)

Reference Implementation
Concerning Multiply and Divide
Binary Arithmetic Unit (BAU)

Background

SIMD architectures are useful for accelerating the solution to problems that can be expressed as a large number of wholly independent, parallel computations. The same sequence of manipulations is performed simultaneously on different data items.

This requires careful selection of the specific problems that are best suited to parallelism, organization of the data for input, algorithm to be applied, and retrieval of the final results. The Single Instruction aspect shares hardware for instruction sequencing and decode among all parallel processing units.

Programming languages take a concept from the realm of mathematics and describe numeric *values* using names called *variables*. At the most basic level a variable is assigned an *address* in physical memory. Rudimentary hardware implementations always access the specific address when the value is required.

More advanced designs realize that
1. memory addresses become too large and unwieldy
2. memory accesses are too slow
and implement various caching schemes, such as
1. fast memory *caches* and anticipatory read of adjacent memory addresses
2. *registers* to give fast access and shorter addresses to values
3. a *stack* to reduce the addressing overhead
4. delayed or eliminated memory writes of intermediate values

Stack implementations attempt to solve a related problem by arranging to have a Last-In-First-Out access mechanism that makes *intermediate result values* available immediately. This eliminates the explicit addressing from some machine instructions but does not eliminate the address requirements in the general case of LOAD and STORE to specific addresses.

Another concept from the world of mathematics involves vectors and matrices. The idea is to allow several related values to be referred to by a single name, but distinguished by a subscript. This makes describing and understanding an algorithm much easier for the algorithm designer.

In the software world, this subscripting concept in implemented in the form of *arrays*. A fixed, contiguous area of memory is given a name which refers to its base address. An integer *index* is used to select an offset from the base and refers to the value stored in that array element. Hardware could be implemented to facilitate this common indexing operation, however this is rarely done. Instead the index operations are computed by the processor’s arithmetic unit just like program data.

This sharing of arithmetic operations on memory addresses and data values was a wonderful insight from early computer science. Unfortunately, in the modern world, strict adherence to this concept leads to poorly written, unintelligible programs, slow hardware implementations, a great potential for corrupted results and the opportunity for malicious exploits.

When an algorithm is expressed in mathematical terms it often becomes clear which arithmetic operations must be done sequentially and which are ideally performed in parallel. When the same algorithm is expressed in a typical programming language it is turned into a pedantic sequence of iterations over a small set of steps. In order to achieve any semblance of efficienty on modern hardware, the code is then subjected to an Optimization process. This optimization will typically include such things as the assignment of particular values to registers instead of main memory, elimination of redundant computations and loop unrolling to eliminate unnecessary branches. Optimizers for computers with parallel architecture can often detect opportunities for parallelism, despite the fact that the algorithm was expressed in a serial form.

Consider an algorithm designed from the beginning to be implemented on a truly parallel architecture. Many of these circuitous, inefficient steps can be eliminated. In particular, the SIMD implementation allows us to ensure that the algorithm completes in an exact time (number of cycles) and that all data values are accessed in a sequence that can be determined during program compilation. This allows the elimination of run-time address calculations and program branches. Techniques to accomplish this are described for this architecture.

SIMD Observations

1. All parallel processors must operate synchronously using the same decoded instructions. Results may be different among processors *only* due to differences in input data.
2. Conditional execution of specific instructions is possible within individual processors but will not affect the instruction sequence or timing being performed across the set of all processors. This is the basis of **Branchless Conditionals**.
3. A precision simulation of the hardware running a particular algorithm is possible. Algorithm simulation reveals the precise sequence of memory accesses required for all data values and program code. This sequence *must* be the same for all parallel processors.
4. Using simulation it is possible for the compiler to structure a program such that all memory reads are made to sequential addresses. All that is necessary is to ensure that the correct values were previously written to those sequential locations. This is the basis of **Sequential-Read Addressing**.
5. It is possible to perform the same BLU operation on multiple data streams by stepping through a series of BLU states - one for each stream. By reducing the register set and status bits required by each BLU it becomes reasonable to create multiple **Virtual Lanes** with minimal added hardware.
6. Combining selected numbers of Virtual and Physical lanes allows the designer to fine-tune the performance of a particular design and tailor it to certain applications.
7. It is possible to implement **cross-lane communication** by allowing memory write operations into adjacent physical or virtual lanes. The topology of the address space ensures that the lane adjacency is ring-like and that all cross-lane signalling is symmetrical.

Philosophy

It appears that the fundamental concept of a *Random Memory Read* is both flawed and unnecesssary in many cases. The flaw is that the algorithm must wait while desired address is be presented to the memory subsystem, the memory is accessed, and the result returned. The unnecesssary part comes from the observation that the desired value could be equivalently obtained with a sequential read.

The goal here is to reduce complexity and increase speed.

This architecture is implemented using only sequential memory reads and fire-and-forget asynchronous, queued random memory writes. All memory access is to full RAM rows; writes do not require prefetch of the row.

Memory Addressing

Algorithm simulation will identify the precise sequence in which input values will be needed for a computation. Simulation will follow standard optimization steps including common subexpression elimination, loop unrolling, and function inlining.

Instead of referencing a *memory address* assigned to a variable, we view the value as being needed at a particular *time*. This time corresponds to the sequential location of the required memory reference.

Thus we eliminate the correspondence between a program variable and memory address in the hardware implementation.

Whenever a result is computed, the compiler uses its omniscient simulation to anticipate the next time that value could possibly be used. A memory write stores the value in the sequential location that will be accessed at that time.

Eliminating the address computation overhead that is traditionally required to store the value in a particular variable further streamlines the operation. This allows:
1. Elimination of all run-time computation of memory addresses,
2. Fully automatic Register usage optimization,
3. Elimination of speculative memory reads or code execution.

The compiler is acutely aware of the sequence of values and operations that are called for. This architecture allows that knowledge to be conveyed directly to the hardware without the guesswork implemented by traditional designs.

The computed read/write sequence is precisely the same for all processor lanes.

Memory Reads

Memory RAM rows are read based on a sequential counter (the Current Read Register) that can be thought of as corresponding to the time index in the simulation of the algorithm.

The row being read may be either program code or data. These will be interspersed in actual memory so that when the last instruction of the current row is executed, the next memory read will have automatically loaded the next row of instructions. That instruction row will be latched into the instruction word shift register to be used to specify the next instructions. The next row read will be the program data to be used next.

Frequently, consecutive instructions will need access to the same input value, or a specific instruction will not actually need an input value. In order to support these cases efficiently, each instruction contains a control bit that indicates when to step to the next input value. The compiler will ensure that there are no conflicts between stepping to the next input value and loading the next row of instructions.

Memory Writes

Computed values will be required at some point in the future. This future time corresponds to the actual memory address that will be used as input at that time. A memory write will store the value into that future time slot.

Thus, all memory reads are purely sequential but memory writes are random.

In the SIMD architecture all memory operations are made to physical memory RAM rows. This means that the write operations always change the contents of an entire RAM row. This eliminates the need for read-before-write used by architectures that allow selective modification of words within a RAM row.

It is expected that rows to be written will be placed into a (short) write queue. This will allow consecutive instructions to initiate write operations without stalling processor operation to wait for the write to complete.

All instruction OpCodes have a single “address” which represents the time in the future that the computed value will be needed.
1. This may be zero in which case no actual write will occur and the accumulator value will simply be used by the next instruction.
2. The “address” may be a small number which is interpreted as a reference to a slot in the memory read cache. This can be viewed as a register operation which will make the value available as needed but will not cause a physical memory cycle or delay in program execution.
3. The “address” may be a number larger than the size of the read cache. This will cause the output value to be placed into the RAM write queue for subsequent use. The actual memory address will be computed as the sum of the Current Read Register and the “address” field of the OpCode. This eliminates absolute addressing and shortens the required bit fields. The required memory write cycle will occur under control of the memory cycle controller at some future time.

Memory Cycle Controller

In the conceptual implementation RAM will be accessed strictly with alternating Read and Write cycles occurring with every machine instruction. Reads will be from addresses contained in a continuously incrementing Current Read Register. Writes will place the results (accumulator value) of the current OpCode into the “address” specified in the instruction.

Further, when the last OpCode in the current program row is executed an additional memory Read cycle would be inserted to load the next program instruction row.

Obviously performing a memory read and write cycle with every instruction would be highly inefficient and unnecessary in most cases. The compiler/simulator’s advance knowledge of the timing of data requirements allows the compiler to insert read and write “hints” into each instruction, thus explicitly controlling the

behavior of the memory controller.

Actual implementations will include a fast access Read Queue analgous to a register set on other architectures. This forms the RAM read-ahead cache which is also accessible by short-distance writes. Longer-distance writes would use a separate Write Queue that would actually initiate physical memory write cycles.

The general rules for each memory access cycle would be:
1. Perform the next Write if the Write Queue is full
2. Perform the next Read if there is an empty slot in the Read Queue
3. Perform the next Write if there is anything in the Write Queue

The host (external) interface is used for program and data load. It would cause the insertion of RAM row data and physical addresses into the Write Queue. This allows for Initial Program Load, as well as concurrent load of the next Application Program or data during program execution.

Branchless Conditionals

Many architectures (particularly microcontrollers) implement conditionals in the form of an instruction that tests a state and then inhibits the execution of the next instruction if a condition is satisfied. In the present case, we take this concept further. Blocks of instructions can be enabled or disabled. By using a conditional counter we allow nesting of control structures. The management of these conditions is controlled by a bit field in every instruction.

For deterministic algorithms backward branches (loops) exist only to make the code more compact: they are never required. The present architecture eliminates them completely, thus provably ensuring fully deterministic behavior.

Conditional execution is implemented using a single counter **Cond**. The operation sequence allows IF…THEN…ELSE…END control structures to be correctly evaluated in each lane. By incrementing and decrementing a simple counter nested control structures can be created.

Two bits from each instruction word are used to express the IF, ELSE and END operations. These condition code bits are always evaluated in each lane for each instruction. They test that lane's **Toggle** bit and set that lane's **Cond** counter, independent of the operation of other lanes.

Rules for BLU and memory operations are:
1. Latch BLU computation result into **Acc** only if **Cond** is zero.
2. Write **Acc** to the current write address if **Cond** is zero, otherwise write the current input value to the current write address.

Lookup Tables and Case Statements

The fundamental rule for the present architecture is that all memory reads must be from sequential addresses. This makes the common programming technique of using lookup tables problematic. In any case, a SIMD architecture must replicate the lookup tables for each of the parallel cores and allow them to

be accessed independently. Further, these independent memory accesses must not cause interference or side-effects that would impact the deterministic timing of the operations.

Similarly, program control structures such as CASE statements perform a one-of-many selection of operations within the algorithm.

In keeping with the architecture's design philosophy we provide mechanisms to select the desired values or instructions from the sequential stream of all possible options. In particular, see the **ARRAY** and **CASE** instructions described later.

It is expected that the data lookup tables used here will be relatively small. Selection of elements from large datasets will have been performed by the host (external) control processor prior to setup and initiation of the algorithm.

Memory Organization

Conceptually the physical memory organization of a program might look like this for a four-lane, non virtual implementation with four OpCodes per word:

```
Row 00: |Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|
Row 01: |    Data    |    Data    |    Data    |    Data    |
Row 02: |    Data    |    Data    |    Data    |    Data    |
Row 03: |    Data    |    Data    |    Data    |    Data    |
Row 04: |Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|
Row 05: |    Data    |    Data    |    Data    |    Data    |
```

Every memory row is either OpCodes or data and they are interspersed such that the OpCodes immediately preceed the data that they reference. When the last OpCode in a row is executed, the next-up memory row will actually be the next row of program instructions.

For an implementation with four physical lanes and two virtual lanes the memory layout would look like this:

```
Row 00: |Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|
Row 01: |  Data V0   |  Data V0   |  Data V0   |  Data V0   |
Row 02: |  Data V1   |  Data V1   |  Data V1   |  Data V1   |
Row 03: |  Data V0   |  Data V0   |  Data V0   |  Data V0   |
Row 04: |  Data V1   |  Data V1   |  Data V1   |  Data V1   |
Row 05: |  Data V0   |  Data V0   |  Data V0   |  Data V0   |
Row 06: |  Data V1   |  Data V1   |  Data V1   |  Data V1   |
Row 07: |Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|
Row 08: |  Data V0   |  Data V0   |  Data V0   |  Data V0   |
Row 09: |  Data V1   |  Data V1   |  Data V1   |  Data V1   |
```

Thus, the data for all virtual lanes are stored in blocks of consecutive memory rows. Addressing of data from each of the virtual lanes is handled transparently during each logical instruction cycle. This concept is extended to whatever number of virtual lanes are included in the particular implementation.

The Data operands used within a function may be (1) input data to the function, (2) constant values or (3)

previously computed results. After loading the physical memory and prior to function execution the memory rows might look like this:

```
Row 00: |Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|
Row 01: |    Data    |    Data    |    Data    |    Data    |
Row 02: |    ....    |    ....    |    ....    |    ....    |
Row 03: |    ....    |    ....    |    ....    |    ....    |
Row 04: |Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|
Row 05: |    Data    |    Data    |    Data    |    Data    |
```

Here the "...." values are Don't Care placeholders that will be filled by intermediate results as the function executes. Frequently these intermediate values will make up the bulk of the actual memory usage. Further it would be convenient for a complete program to be stored in a compact block without requiring (1) lots of null values, or (2) addresses to be associated with each row. The second point is important if we want to be able to load a program into the BLU in a streaming fashion from the external (host) interface.

Program Loader

In order to accomplish this monolithic-load we expect the use of a program loader on the front of the actual program. The loader will consist of the instructions necessary to unpack the code+data rows into correct positions for actual execution. This will have the additional advantage of allowing frequently-used constant data to occur only once during the load and yet be correctly placed in multiple locations as required for execution.

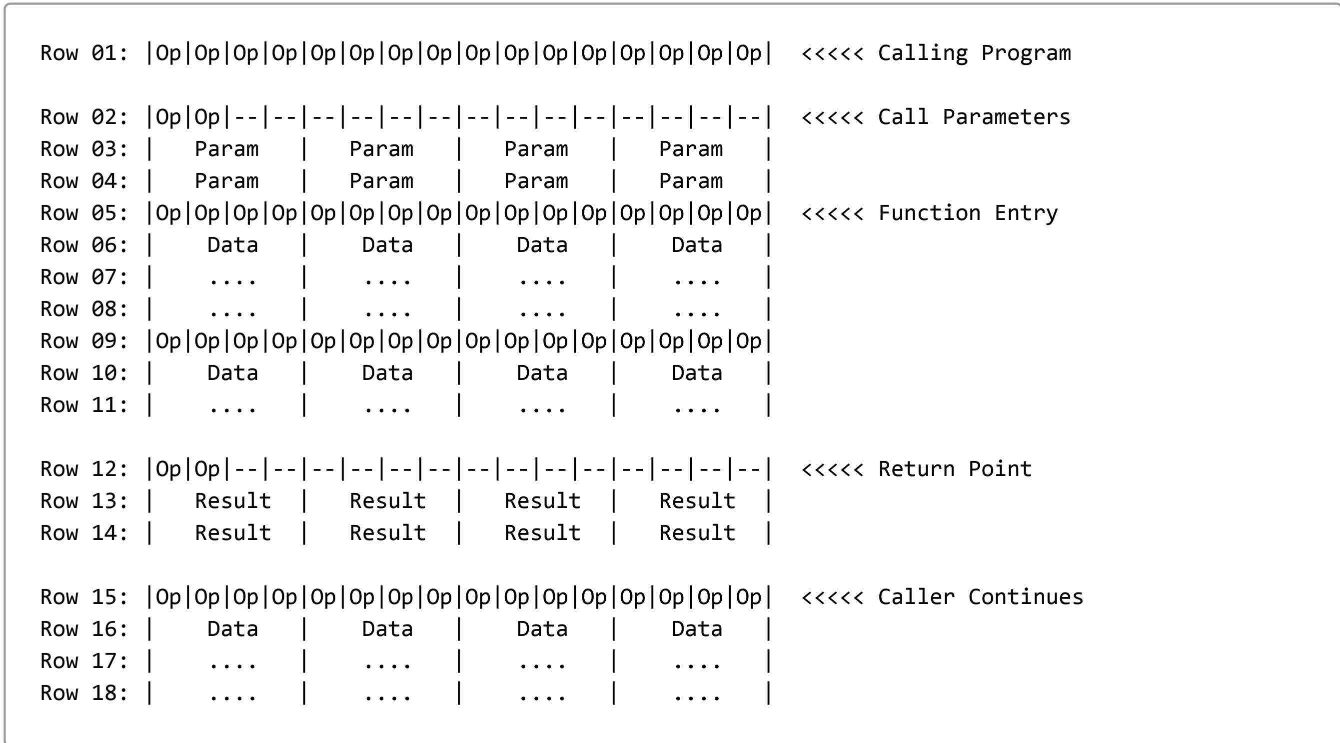
```
Row 00: |Op|Op|Op|Op|--|--|--|--|--|--|--|--|--|--|--|--|  <<<<< Loader
Row 01: |Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|    --|
Row 02: |    Data    |    Data    |    Data    |    Data    |      | Compact
Row 03: |Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|      | Code+Data
Row 04: |    Data    |    Data    |    Data    |    Data    |    --|

Row 05: |Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|  <<<<< Program execution begins here
Row 06: |    Data    |    Data    |    Data    |    Data    |
Row 07: |    ....    |    ....    |    ....    |    ....    |
Row 08: |    ....    |    ....    |    ....    |    ....    |
Row 09: |Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|
Row 10: |    Data    |    Data    |    Data    |    Data    |
```

In the example the loader in RAM Row 00 copies the next four RAM rows into the correct locations in rows 05 to 10. The "–" OpCodes represent CODE instructions, the first of which actually loads RAM Row 05 (the first instructions in the actual function) for execution. This provides a compact method of loading a program, even though the actual execution of that program may use extravagant amounts of memory for intermediate results. Note also that all aspects of the memory accesses are designed to be relocatable: All addressing is relative and uses positive offsets.

Concerning Subroutines

It is important to be able to package standardized “library functions” in a convenient form so that optimized solutions to common problems are available to the programmer. Library functions will have very specific, determininstic behavior in this architecture. The size and input/output interface will be well defined. Conceptually, a library function will be similar to the code block illustrated above, but with added interface elements similar to this:



```
Row 01: |Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|  <<<<< Calling Program

Row 02: |Op|Op|--|--|--|--|--|--|--|--|--|--|--|--|--|--|  <<<<< Call Parameters
Row 03: |   Param    |   Param    |   Param    |   Param    |
Row 04: |   Param    |   Param    |   Param    |   Param    |
Row 05: |Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|  <<<<< Function Entry
Row 06: |    Data    |    Data    |    Data    |    Data    |
Row 07: |    ....    |    ....    |    ....    |    ....    |
Row 08: |    ....    |    ....    |    ....    |    ....    |
Row 09: |Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|
Row 10: |    Data    |    Data    |    Data    |    Data    |
Row 11: |    ....    |    ....    |    ....    |    ....    |

Row 12: |Op|Op|--|--|--|--|--|--|--|--|--|--|--|--|--|--|  <<<<< Return Point
Row 13: |   Result   |   Result   |   Result   |   Result   |
Row 14: |   Result   |   Result   |   Result   |   Result   |

Row 15: |Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|Op|  <<<<< Caller Continues
Row 16: |    Data    |    Data    |    Data    |    Data    |
Row 17: |    ....    |    ....    |    ....    |    ....    |
Row 18: |    ....    |    ....    |    ....    |    ....    |
```

The sequence of events is as follows:
1. The Calling Program computes a set of parameters and places them in memory slots preceding the Function Entry point.
2. The Call Parameters code is part of the library function and is responsible for placing the parameters into appropriate slots within the Function Body itself.
3. The Row labeled Function Entry and following are the actual body of the function. The function computes the desired results and stores them in slots following the Row labeled Return Point.
4. The Function completes and hits the code Row labelled Return Point, which is properly considered part of the Calling program.
5. The instructions beginning at Return Point distribute the Result values to the correct parts of the Calling program.
6. Execution continues with the Calling program.

This sequence separates Calling and Called code and makes for independent modules, even though the functions themselves are rendered as in-line code. In many cases this will simplify design, debugging and simulation by giving standardized or frequently-used functions a consistent appearance.

Production code, however, will undoubtedly use an intelligent linker to compact the code and eliminate the redundant data movement of Parameters and Results described here.

Instruction Set

We envision the instruction set to be a minimal set of very fast operations. The goal is to keep the footprint area and power consumption of each BLU implementation to a minimum. This enables all logic modules to be aligned with the memory cache bits when laid out on the chip.

The following table represents a candidate list of BLU machine instructions and OpCodes. Selection of instructions to be included in a final design will be made on the basis of utility and gate count.

There are basically four different instruction formats.

Format	Description
`ccnn 0000 00zz iiii`	Non-Write and Control Instructions
`ccnn 0000 zzbb bbbb`	Bit Test/Clear/Set Instructions
`ccnn iiii 1111 eeee` `eeee eeee eeee eeee`	Binary operations with long write addresses
`ccnn iiii dddd dddd`	Binary operations with short write addresses

The full set of OpCodes is represented by **zzzz iiii**. Not all of these 256 OpCodes are implemented. Unimplemented OpCodes yield undefined results.

Sixteen OpCodes of the form **0000 iiii** are selected to be applicable to both the Short-Write and Long-Write instructions. This allows common sequences of the form compute…store to be specified in a single instruction.

In addition, 192 instructions of the form **xxbb bbbb** are reserved to directly test, clear and set individual bits in **Acc**.

	0	1	2	3	4	5	6	7	8	9	A	B	C
0	LOAD	PUT	PUTW	PUTE	PUTN	PUTS	ADD	NEG	NOT	AND	OR	XOR	R
1	CODE	HALT	END	PARITY	ZERO	LOG	COUNT	DEC	REVERSE	ELSEIF	CASE	BREAK	LF
2	--	--	--	--	--	--	--	--	--	--	--	--	-
3	MUL	DIV	REM	--	--	--	--	--	COMP	COMP	COMP	COMP	CO
4-7	TEST	TEST	TEST	TEST	TEST	TEST	TEST	TEST	TEST	TEST	TEST	TEST	TE
8-B	CLEAR	CLEAR	CLEAR	CLEAR	CLEAR	CLEAR	CLEAR	CLEAR	CLEAR	CLEAR	CLEAR	CLEAR	CL
C-F	SET	SET	SET	SET	SET	SET	SET	SET	SET	SET	SET	SET	S

Control Instructions

`zzzz iiii`	Instruction	Description
`0001 0000`	CODE	Latch next Code Row. No change to **Acc**
`0001 0001`	HALT	Terminate algorithm. Signal completion to host

`0001 0010`	END	Set **Cond** to zero, closing all conditionals
`0001 1011`	BREAK	Set **Cond** to 11 1111.
`0001 1001`	ELSEIF	Decrement **Cond** only if greater than one.
`0001 1101`	OUTPUT	Queue the current Write for external (host) interface

Bit Instructions

`zzzz iiii`	**Instruction**	**Description**
`01bb bbbb`	TEST `b`	Copy the value of bit `b` to **Toggle.** No change to **Acc**
`10bb bbbb`	CLEAR `b`	Set bit `b` in **Acc** to 0
`11bb bbbb`	SET `b`	Set bit `b` in **Acc** to 1

Data Copy Instructions

`zzzz iiii`	**Instruction**	**Description**
`0000 0000`	LOAD	Copy **Next** to **Acc**
`0001 1111`	ARRAY	Decrement **Acc.** If zero, Increment **Cond** and copy **Next** to **Acc**
`0000 0001`	PUT	No change to **Acc**
`0000 0010`	PUTW	Write to Physical Lane on West, No change to **Acc**
`0000 0011`	PUTE	Write to Physical Lane on East, No change to **Acc**
`0000 0100`	PUTN	Write to Virtual Lane on North, No change to **Acc**
`0000 0101`	PUTS	Write to Virtual Lane on South, No change to **Acc**

Arithmetic Instructions

`zzzz iiii`	**Instruction**	**Description**
`0000 0110`	ADD	Integer ADD **Next** and **Acc**

Single Operand Instructions

`zzzz iiii`	**Instruction**	**Description**
`0000 0111`	NEG	Integer Twos Complement Negate **Acc**

`0000 1000`	NOT	Logical NOT **Acc**
`0001 0011`	PARITY	Set **Toggle** from **Acc.** No change to **Acc**
`0001 0100`	ZERO	Set **Toggle** if **Acc** is zero. No change to **Acc**
`0001 0101`	LOG	Set **Acc** to number of highest bit set in **Acc**
`0001 0110`	COUNT	Set **Acc** to total number bits set in **Acc**
`0001 1110`	INC	Increment **Acc.** Set **Toggle** if result is negative
`0001 0111`	DEC	Decrement **Acc.** Set **Toggle** if result is negative
`0001 1000`	REVERSE	Reverse the order of all 64 bits in **Acc**
`0001 1010`	CASE	If **Acc** is zero set **Cond** to zero. Decrement **Acc.**

Dual Operand Instructions

`zzzz iiii`	**Instruction**	**Description**
`0000 1001`	AND	Logical AND **Next** and **Acc**
`0000 1010`	OR	Logical OR **Next** and **Acc**
`0000 1011`	XOR	Logical XOR **Next** and **Acc**
`0000 1100`	ROT	Bitwise Rotate Left **Next** bits in **Acc**
`0000 1101`	MASKL	Bitwise Mask Left **Next** bits in **Acc**
`0000 1110`	MASKR	Bitwise Mask Right **Next** bits in **Acc**
`0000 1111`	EXTEND	Bitwise Extend bit **Next** leftward in **Acc**
`0001 1100`	LFSR	Linear Feedback Shift Register using **Next** and **Acc**

`zzzz iiii`	**Instruction**	**Description**
`0010 ----`	UNUSED	16 Unused OpCode Block
`0011 ----`	UNUSED	16 Unused OpCode Block (reserved for BAU)

The **cc** field is the condition code used to enable branchless conditional execution.

Condition Code	**Operation**	**Description**
0 0		No change to **Cond**

0 1	IF	Increment **Cond** if **Toggle** is zero
1 0	ELSE	Invert the low-order bit of **Cond**
1 1	END	Decrement **Cond** if it is non-zero

All instructions latch BLU result into **Acc** only if **Cond** is zero.

In general expect instructions to write the current **Acc** value to the write address. There are exceptions, so, to be specific:
1. If write Address is zero, do not write
2. If OpCode is PUTx and **Cond** is non-zero, write **Next** instead.
3. If **Cond** is zero, write **Acc** to write address.

The **nn** field is the Next Input code used to control selective stepping through the sequential memory reads and the sequential read cache.

Next Input Code	Operation	Description
0 0	REP	Repeat the current Input Value
0 1	REG	Step to the next cache value
1 0	RAM	Step to the next RAM value
1 1	ALL	Step both Cache and RAM indexes

The **dddd dddd** field is the destination write address. This 8-bit value is relative to the current read address. Zero would be the current input value (represented by **Next**); in reality a zero value is an indicator to suppress the write operation altogether.

The eight bit **dddd dddd** value would give a range up to 255 values in the future. We wish to extend this timescale to at least 2^{19}. This is accomplished by reserving addresses of the form **1111 dddd** to indicate *long addresses* in which the last four bits are pre-pended onto the 16-bits of the next instruction register value to form a 20-bit relative address. This long address is referred to as **eeee eeee eeee eeee eeee**.

Thus, the allowed range of **dddd dddd** addresses is 0 to 239.

I/O: Host Setup and Retrieval

A host (external) control processor writes contents to RAM to initialize code and data prior to operation. This may also be used to load the next algorithm/data during the execution of the previous.

Random writes from the host (external) control processor fit in with the general architecture by simply adding RAM rows and addresses to the existing write queue.

Computation results may be read from the RAM by the host (external) control processor. Alternatively, the SIMD algorithm may issue Output instructions to write full RAM rows to an external device.

The use of a specific OUTPUT instruction complements the architectural design but requires a ready and waiting control processor to handle this data at a moment's notice. This may pose an unacceptable burden to the off-chip interface. A sequential FIFO queue to buffer pending OUTPUT data may be used to mitigate this issue.

Binary Logic Unit (BLU)

Full consideration must be given to the tradeoff between BLU Complexity (chip real estate), parallelism, power consumption and speed. In particular, it may be necessary to limit the instruction set due to these constraints. Therefore, the availability of certain of the more exotic instructions described here should be considered implementation dependent. The compiler/assembler is expected to make suitable adjustments to the resulting code.

. . . T B D . . .

The BLU is a two-input, 64-bit combinatorial logic processor. All instruction OpCodes complete in one machine cycle. The operation result is (conditionally) latched into the Accumulator **Acc**. The accumulator value is always used as one of the two inputs to the next operation performed by the BLU. An auxiliary output bit from the BLU is called **Toggle**; this bit represents a state that can be tested for controlling conditional operations.

Reference Implementation

We describe an implementation of 64-bit SIMD processors attached to a 64MB, 1024-bit wide DRAM interface. There are 16 parallel **lanes** under the control of one set of instruction decode and sequencing logic.

Parameters that define the implementation are:

Value	Item	Description
64	**Bits**	Word size in bits
1024	**Row**	RAM Row Length in bits
64	**Ram**	Total RAM size in MB
19	**Addr**	RAM Row Address Bits
2^{19} = 524,288	**Rows**	Number of RAM rows
16	**Lanes**	Parallel processing units

16	**VLanes**	Virtual lanes processed serially
256	**Cache**	Rows stored in fast-read cache

Each BLU lane (virtual or physical) has an internal state defined by

Bits	Register	Description
64	**Acc**	Accumulator
1	**Toggle**	Auxiliary BLU Status Bit
6	**Cond**	Conditional Execution Nesting Counter

Standard instructions are 16-bits wide, packed 64 per RAM row.
A Standard Instruction may be followed by an addional 16-bit Long Address value. Instructions with appended Long Addresses may not cross a RAM row boundary. If necessary, adjust boundaries by inserting a **CODE** OpCode as the last instruction in the RAM row.

The Instruction Control Unit implements the following shift register

Bits	Registers	Description
16	64	Instructions in code shift register

Each instruction cycle shifts in the next 16-bit instruction. If the instruction decodes as requiring a long address the next 16-bit word is used and an extra shift cycle occurs. With each shift, the value of a **CODE** OpCode is shifted in to the last place in the shift register. This eliminates the need for any form of instruction counter or index register and associated logic.

The execution of a **CODE** OpCode loads the instruction shift register, in parallel, from the contents of the **Next** RAM Row.

Concerning Multiply and Divide

Thus far the architectural discussion has centered on bitwise operations in a compact, parallel-operation environment. Frequently a co-processor such as this will be applied to scientific, neural net or graphical problems that require extended precision arithmetic computations. We now consider the minimum requirements for these operations and suggest compact, power-efficient extensions to support them.

The IEEE 754 Standard for Binary Floating Point Arithmetic is one of the cleverest, most widely adopted standards in existence. Its use allows a high degree of performance optimization for all kinds of higher mathematical functions. In order to achieve these performance benefits, specialized hardware is required. Specifically, the standard includes special-case bit configurations indicating
1. Signed Zero values
2. Signed Infinite values

3. Denormalized values
4. NAN values
5. Ordinary signed floating-point values

Handling every combination of these configurations used as input to every supported operation, in a conformant manner, is quite slow if a software implementation is required. Further, accurate handling of result rounding is not trivial. The hardware implementation is not compact, but can be made quite fast - especially when consecutive, pipelined operations can obscure the latency of individual instructions.

Specialized hardware can also be quite beneficial in other cases. For example modern Multiply-Accumulate (MAC) units implement fused multiply-add which significantly reduces the number of roundings. Thus, higher precision results are achieved than would even be possible with a (realistic) software implementation.

Programming languages bridge the gap between the designers of a software algorithm and the actual hardware that will implement it. Traditionally the languages and its compilers will implement data types that resemble those supported by the physical hardware. This becomes problematic when a variety of different hardware capabilities are involved. For example, a programmer would like for his algorithm to "just work" - even though some implementations might include floating-point hardware and some might not. The solution to problems such as this typically involves the use of libraries of hand-crafted routines that ensure standardized results for operations no matter what features the underlying hardware might provide.

Unfortunately, this "library solves the problem" approach is insufficient because the programming languages themselves are poorly defined. Even such critical features as the number of bits in an integer are "implementation dependent" as far as the language definition is concerned. This leads to the use of generalized consensus in place of rigor. "Everybody knows" that an **int** is 32-bit twos complement. Except in older code. Except in embedded systems. Except on GPUs. Except when . . . . *ad infinitum*.

Legacy programming languages implement poorly conceived, incomplete and frequently inappropriate data type declarations. Hardware designers make trade-offs relating to speed, power consumption and complexity that cause a wide variety of different features to be implemented. The combination of language and hardware incompatibilities means that it is literally not possible for an algorithm designer to express his actual intentions.

Furthermore, in high-performance applications the algorithm designer may have very specific goals. The fact that these goals cannot be expressed explicitly limits not only the performance but the ability of hardware designers to recognize possible improvements.

The hardware designer tries to implement features that will correspond to the expectations of the programming language and the programmers. Hardware designs typically target these general expectations (or the performance requirements of very specific benchmarking software for marketing reasons).

This leads to a situation where high-performance algorithm designers are forced to target specific hardware implementations with manually crafted, assembly-level code. Thus, high-level languages, features and compilers play essentially no part in the process - except to act as spoilers by leading to naive expectations on the part of neophyte software, hardware and marketing engineers.

Things like the (otherwise wonderful) IEEE 754 Standard lead to a bloated and wasteful one-size-fits-all mentality when carried to extremes. Once a successful FPU or MAC design is created, the temptation is to increase performance simply by putting thousands of them on a chip.

Processor arrays such as these are typically found in GPUs. While they may be able to achieve spectacular performance for their intended algorithms, small changes can lead to surprises. For example, GPU performance on single-precision operands is typically eight times better than double-precision. Graceful scaling is replaced with stepwise penalties. Much effort, complexity, power consumption and risk is associated with creating hardware elements that are rarely if ever used in the real world.

In the software world, this mentality causes programmers to pick a favorite numerical representation to use - whether it is warranted or not. Rules-of-thumb take precedence over rational analysis of algorithms, scaling and precision. And, as we have seen, the style of data type declarations actively prevent the languages and compilers from providing meaningful assistance.

Binary Arithmetic Unit (BAU)

The present architecture takes a very simplistic approach to providing floating-point operations. We already implement powerful instructions for the manipulation of binary bit fields. These allow us to import and export data in rather arbitrary representations. Thus, we need not feel constrained to actually implement specific hardware standards if we can accomplish our goals more efficiently and maintain interface compatibility.

Internally all our operations are performed on 64-bit words, but this does not mean that we are constrained to only 64-bit precision. We implement integer multiply and divide operations on a bit-at-a-time basis using a single 64-bit hardware adder. This allows us significant advantages

- Speed scales inversely with precision
- Power consumption scales linearly with precision
- Hardware utilization is maximized
- Hardware complexity is minimized
- Precision scaling as required by the specific algorithm
- Arbitrary mantissa and exponent sizes are supported

Most importantly, the choice of precision can be made in each individual instance within the program. The algorithm designer is given considerable flexibility and is no longer as constrained by arbitrary decisions made *a priori* by the hardware design. Speed, resolution and power consumption can be directly matched to the design requirements.

It is true that more specialized hardware can be expected to perform faster, but the design presented here is expected to be best-in-class from the standpoints of power consumption and hardware complexity. The thought process for the designer should be viewed more along the lines of using microcode instructions to build a result, as opposed to simply accepting results from standardized FPU modules.

It is well understood that more sophisticated techniques (such as base-8 Booth Recoding) can significantly improve the speed of hardware multiply implementations. Division can also be accelerated with digit recurrence algorithms which converge linearly or functional iteration (using multiplies) which converge

quadratically. Pipelining and parallelism within the core can also increase throughput in many cases. These approaches are appropriate in a speed-at-all-cost design. The present architecture takes a more nuanced path, eschewing raw speed in favor of a compact, low-power, adaptable implementation.

Floating point representations including manipulation of signed values, exponents, exponent offsets, significands and rounding modes are handled as software features on an as-needed basis. Acceleration is provided for multiplication and division of non-negative binary integers.

In order to support unsigned integer Multiply and Divide operations, each BLU lane (virtual or physical) has an extended internal state. The extended state is defined by the following table. A Binary Logic Unit (BLU) with these extended capabilities is referred to as a Binary Arithmetic Unit (BAU).

Bits	Register	Description
64	**Acc**	Accumulator
1	**Toggle**	Auxiliary BLU Status Bit
6	**Cond**	Conditional Execution Nesting Counter
1	**MulDiv**	Indicator Bit for Multiply (set) or Division (reset)
64	**AccX**	Multiplier or Divisor
64	**AccY**	Multiplicand or Dividend

The addition of the single-bit **MulDiv** indicator and two 64-bit registers to the internal state requirements of each BLU is all that is necesssary to support the following instructions

`zzzz iiii`	Instruction	Description
`0011 0000`	MUL	Set **Next** into **AccX**, **Acc** into **AccY**, set **MulDiv** and clear **Acc**
`0011 0001`	DIV	Set **Next** into **AccX**, Twos Complement of **Acc** into **AccY** and clear **MulDiv** and **Acc**
`0011 0010`	REM	Copy remainder into **Acc**
`0011 0011`	UNUSED	Unused OpCode
`0011 01--`	UNUSED	4 Unused OpCode Block
`0011 1jjj`	COMPUTE `j`	Compute next 2^{j} bits into product or quotient

The technique begins with the use of a **MUL** or **DIV** operation to set the operands into the appropriate registers, clear the accumulator for the upcoming results and set or clear the **MulDiv** indicator. Then repeated **COMPUTE** instructions are used to build the result to the desired precision. Controlled by the **MulDiv** indicator, the **COMPUTE** operations embody the standard shift-and-conditional-add or shift-and-conditional-subtract concept. In this case, however, the subtract used for the Division-mode operation is handled by the adder and uses a twos-complement value initialized in the **DIV** instruction. At the completion of each bitwise step the partial product or partial quotient is in the **Acc**. When all required bits have been computed the final result is therefore in the **Acc**. In the case of division, the remainder can be accessed with the **REM** instruction.

Using a single **COMPUTE** instruction per bit of result would naturally lead to long consecutive sequences of **COMPUTE** OpCodes in the instruction code stream. This matches the conceptual requirements and does not (significantly) affect the speed, but it does result in much unnecessary data movement as each virtual lane is loaded, processed and saved, thus having a serious impact on power consumption. This issue is alleviated by configuring the **COMPUTE** OpCode to include the three-bit **jjj** field to control a repetition counter. The **jjj** value initializes a bit in a 6-bit OpCode Repetition Counter, thus enabling 2^{j} consecutive operations on a single virtual lane with a single OpCode.