RISC-V/SYMPL HYBRID ISA COMPUTE ENGINE

New hybrid ISA genetically splices the instruction pipeline of a RISC-V to the instruction pipelines of one or more *SYMPL* Compute Engines, giving you the best of both worlds in a single package.

Overview

By re-purposing certain previously never-used RISC-V instructions and making the ***SYMPL*** Compute Engine (SCE) instruction pipeline immediately responsive to them when such re-purposed RISC-V instructions enter the RISC-V instruction pipeline, the result is an exceedingly powerful hybrid ISA compute engine.

With these re-purposed RISC-V instructions and the fused instruction pipelines, the RISC-V now has the ability to access, on-the-fly, any memory location, any register (including PC, Status Register, Auxiliary Register, etc.), any floating-point, logical or integer arithmetic operator input or result buffer, including pushing dual operands, etc., that the SCE can access and with the same amount of ease.

To maximize the power of the RISC-V/SCE hybrid architecture, the RISC-V now has a programmable instruction-REPEAT capability with *sourceA, sourceB* and *destination* auto-post-modify indirect addressing capability, allowing the RISC-V, with a single instruction, to push or pull entire single or dual vectors of 1, 2, 4, 8, 16, 32, 64 or 128-byte elements each, into or out of the SCE memory/register/operator space into or out of the RISC-V data memory space.

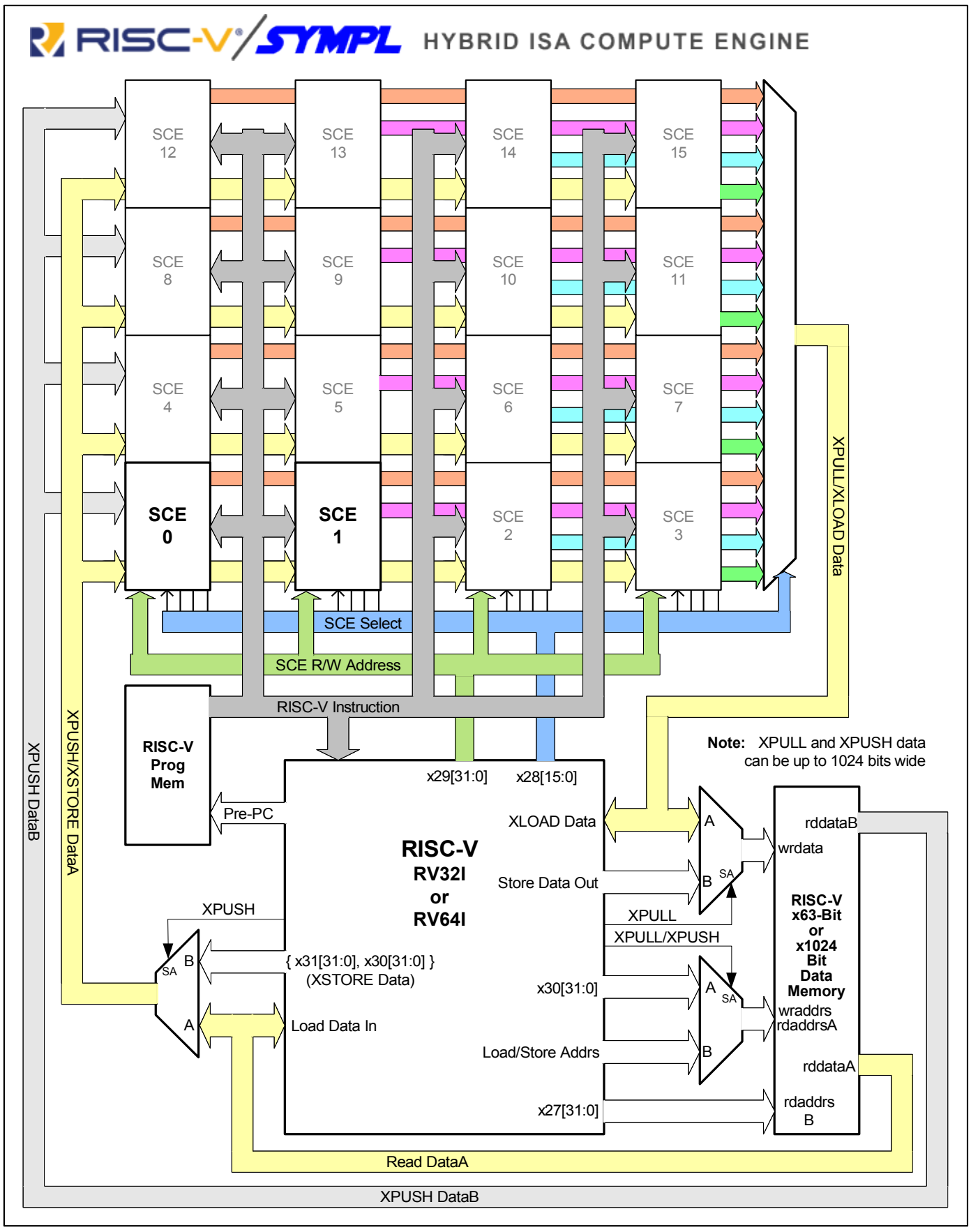
Designed for implementation in Xilinx® Kintex® Ultra brand FPGAs, an evaluation Verilog RTL source code model of this new hybrid ISA is available for free download at the ***SYMPL*** Compute Engine repository at GitHub: www.github.com/jerry-D

Because the ***SYMPL*** Compute Engine has the architectural infrastructure in place to implement in hardware the entire repertoire of mandated IEEE® 754-2008 floating-point operations, including all computational and non-computational operators, convertFromDecimalCharacter, convertToDecimalCharacter, etc., by fusing the instruction pipeline of the RISC-V to the instruction pipelines of up to qty. (16) SCE cores, the RISC-V/SCE hybrid now has the capability of being fully IEEE 754-2008 compliant in hardware. As such, the RISC-V/SCE now has the ability, for example, to compute directly with floating-point decimal character sequences up to 28 decimal digits in length without having to explicitly convert them to binary format before entering the computational stream.

Unlike the previous ***SYMPL*** Strap-on-Booster (SOB) implementation, this RISC-V/SCE hybrid ISA implementation does not require a multi-port SRAM parameter/data buffer thru which parameters and data are passed between RISC-V and SCE, in that the instruction pipeline of the RISC-V is genetically spliced to the instruction pipelines of the SCE(s), giving the RISC-V complete control of, and immediate access to, all computational and control resources of every SCE fused/grafted/genetically-spliced to it, making SCE resources RISC-V resources using its own re-purposed instructions.

For a list and description of the re-purposed RISC-V instructions, *see* next Page of this *Preliminary Information* sheet.

Preliminary Information



SYMPL is presently an alias used by an individual developer.
RISC-V/SCE Hybrid ISA Compute Engine IP is presently not available for commercial use.

Brief Description

From the block diagram above, it can be seen that the RISC-V and any SCE(s) fused to it fetch instructions from the same RISC-V program memory. In this instance, the RISC-V is a RV32-I configured as a Harvard memory model, having separate program and data address/data buses. In parallel with that, the SCE simultaneously fetches instructions from its own program memory space, carrying out its intended task. Thus, whenever a RISC-V hybrid instruction appears on the RISC-V instruction bus, logic in the SCE gives priority to the RISC-V hybrid instruction and executes the RISC-V instruction instead. Once completed, the SCE will re-fetch the instruction preempted by the RISC-V instruction and continue executing the program in its own program memory space until another RISC-V hybrid instruction comes along and preempts it again.

For more information on the SYMPL Compute Engine architecture and programming model, refer to the .pdf documentation located at the RISC-V/SYMPL Hybrid ISA Compute Engine repository here: www.github.com/jerry-D

It should be understood that the SCE does not employ “opcodes” to carry out operations. Instead, all operators, including PC, Status Register, Auxiliary Registers, Stack Pointer, etc., are all memory-mapped. Thus, an application need only know where these resources reside in the memory map to read or write from/to them.

It should be further understood that the SCE instruction set, that is, the instruction set that one programs to carry out an application, has not changed. Rather, the instruction pipeline of the RISC-V has been fused to the instruction pipeline of the SCE. Applications written using the original SCE instruction set continue to function the same. The difference now is that RISC-V hybrid instructions fetched by the RISC-V from its program memory space are simultaneously executed by the SCE and have priority over the original SCE instructions the SCE fetches from its own program memory.

Theory of Operation

To understand how the newly re-purposed RISC-V hybrid instructions work, it is important to understand that the SCE does not employ opcodes in its instruction set to carry out operations. Instead, it fundamentally only knows how to do one thing and one thing only—MOVE—but it does it very, very well. The ***SYMPL*** Compute Engine is a very efficient pure “mover” architecture, in that it can move multiple operands up to 128 bytes each, simultaneously, every clock cycle. All of its instructions comprise at least one source address,*SrcAddrsA,* and a destination address, *DestAddrs*. For operations involving more than one operand, the SCE instruction also comprises a second source address, *SrcAddrsB*. Within this instruction, each source address and the destination address fields also include three *Size* bits and a *Signal* bit. There are a few other bits in the instruction that signal rounding mode or neural network activation mode and/or accumulate modes. For more information on the SCE instruction bit fields, please refer to the .pdf documentation and readme file located at the RISC-V/SYMPL Hybrid ISA Compute Engine repository here: www.github.com/jerry-D

Thus, to access any resource within the SCE's memory/register/operator space, all the RISC-V logic needs to do is provide the SCE with source and destination address and size information, which the SCE logic can use to assemble, on-the-fly in real-time, a SCE instruction that either reads or writes to the target SCE's memory/register/operator location using the address information, signals and modes specified by the contents of certain RISC-V registers. Where an SCE operation is to be repeated with the same instruction, logic has been added to the RISC-V to freeze its PC, decrement its REPEAT counter until it reaches “1” and increment the source and address pointers provided to the SCE, such decrementing and incrementing occurring every clock cycle.

In the implementation shown in above block diagram, the RISC-V/SCE hybrid employs the RISC-V registers x31, x30, x29, x28 and, optionally, x27, for providing the address, size and mode information the SCE(s) require to instantly assemble a hybrid instruction the SCE is to carry out. For implementing a REPEAT counter internal to the RISC-V and only for use with hybrid instructions that are repeatable, a portion of RISC-V register x31 is used for this purpose as explained in the following section.

RISC-V Hybrid Instruction Descriptions

As mentioned previously, certain never-used original RISC-V instructions have been re-purposed for carrying out operations over the gene-spliced/grafted/fused RISC-V/SCE pipelines. The original RISC-V instructions that have been re-purposed for this are as follows, with the re-purposed operations all being implied:

Machine Code	RISC-V Original Instruction	Re-Purposed Hybrid Instruction
0000C093	XORI x1 , x1 , 0	XLQU
00014113	XORI x2 , x2 , 0	XLDU
0001C193	XORI x3 , x3 , 0	XLWU
00024213	XORI x4 , x4 , 0	XLHU
0002C293	XORI x5 , x5 , 0	XLBU
00034313	XORI x6 , x6 , 0	XSQ
0003C393	XORI x7 , x7 , 0	XSD
00044413	XORI x8 , x8 , 0	XSW
0004C493	XORI x9 , x9 , 0	XSH
00054513	XORI x10, x10, 0	XSB
0005C593	XORI x11, x11, 0	XPUSH
00064613	XORI x12, x12, 0	XPULL
0006C693	XORI x13, x13, 0	XPUSHD
00074713	XORI x14, x14, 0	XFRCBRK
0007C793	XORI x15, x15, 0	XCLRBRK
00084813	XORI x16, x16, 0	XFRCRST
0008C893	XORI x17, x17, 0	XCLRRST
00094913	XORI x18, x18, 0	XSSTEP
0009C993	XORI x19, x19, 0	XSTATUS

As can be seen from the above instruction list, original RISC-V xori rd, rs1, 0 instructions, where rd and rs are in the range from x1 to x19, inclusive, and *rd* and *rs1* are the same register, have been re-purposed into the implied hybrid instructions shown. Furthermore, it should be understood that whenever a re-purposed hybrid instruction appears on the RISC-V program instruction bus, both the RISC-V and SCE simultaneously executes it.

The re-purposed hybrid instructions are described as being implied because whenever one of these instructions require an address or data field, they always use the same RISC-V registers to provide this information.

XLQU

XLQU loads RISC-V (RV64I-only) register pair {x31[63:0], x30[63:0]} with the 128-bit value from SCE memory/register/operator space pointed to by the contents of RISC-V register x29. One of the lower 16 bits of x28 specifies from which SCE the data is read, with bit 0 corresponding to SCE(0), bit 1 corresponding to SCE(1), bit 2 corresponding to SCE(2) and so forth. This instruction is only available for RV64I versions of the RISC-V. The XLQU instruction is not repeatable.

XLDU

XLDU loads RISC-V register pair {x31[31:0], x30[31:0]} with the 64-bit value from SCE memory/register/operator space pointed to by the contents of RISC-V register x29. One of the lower 16 bits of x28 specifies from which SCE the data is read, with bit 0 corresponding to SCE(0), bit 1 corresponding to SCE(1), bit 2 corresponding to SCE(2) and so forth. This instruction is available for both RV32I and RV64I versions of the RISC-V. For RV64I versions, the values loaded into registers x31 and x30 are zero-extended to 64 bits. The XLDU instruction is not repeatable.

XLWU

XLWU loads RISC-V register x30[31:0] with the zero-extended 32-bit value from SCE memory/register/operator space pointed to by the contents of RISC-V register x29. One of the lower 16 bits of x28 specifies from which SCE the data is read, with bit 0 corresponding to SCE(0), bit 1 corresponding to SCE(1), bit 2 corresponding to SCE(2) and so forth. This instruction is available for both RV32I and RV64I versions of the RISC-V. The XLWU instruction is not repeatable.

XLHU

XLHU loads RISC-V register x30[31:0] with the zero-extended 16-bit value from SCE memory/register/operator space pointed to by the contents of RISC-V register x29. One of the lower 16 bits of x28 specifies from which SCE the data is read, with bit 0 corresponding to SCE(0), bit 1 corresponding to SCE(1), bit 2 corresponding to SCE(2) and so forth. This instruction is available for both RV32I and RV64I versions of the RISC-V. The XLHU instructions is not repeatable.

XLBU

XLBU loads RISC-V register x30[31:0] with the zero-extended 8-bit value from SCE memory/register/operator space pointed to by the contents of RISC-V register x29. One of the lower 16 bits of x28 specifies from which SCE the data is read, with bit 0 corresponding to SCE(0), bit 1 corresponding to SCE(1), bit 2 corresponding to SCE(2) and so forth. This instruction is available for both RV32I and RV64I versions of the RISC-V. The XLBU instructions is not repeatable.

XSQ

XSQ stores the 128-bit contents of RISC-V (RV64I) register pair {x31[63:0], x30[63:0]} into SCE memory/register/operator space pointed to by the contents of RISC-V register x29. One of the lower 16 bits of x28 specifies to which SCE the data is written, with bit 0 corresponding to SCE(0), bit 1 corresponding to SCE(1), bit 2 corresponding to SCE(2) and so forth. This instruction is only available for RV64I versions of the RISC-V. The XSQ instruction is not repeatable.

XSD

XSD stores the 64-bit contents of RISC-V register pair {x31[31:0], x30[31:0]} into SCE memory/register/operator space pointed to by the contents of RISC-V register x29. One of the lower 16 bits of x28 specifies to which SCE the data is written, with bit 0 corresponding to SCE(0), bit 1 corresponding to SCE(1), bit 2 corresponding to SCE(2) and so forth. This instruction is available for both RV32I and RV64I versions of the RISC-V. The XSD instruction is not repeatable.

XSW

XSW stores the 32-bit contents of RISC-V register x30[31:0] into SCE memory/register/operator space pointed to by the contents of RISC-V register x29. One of the lower 16 bits of x28 specifies to which SCE the data is written, with bit 0 corresponding to SCE(0), bit 1 corresponding to SCE(1), bit 2 corresponding to SCE(2) and so forth. This instruction is available for both RV32I and RV64I versions of the RISC-V. The XSW instruction is not repeatable.

XSH

XSH stores the 16-bit contents of RISC-V register x30[15:0] into SCE memory/register/operator space pointed to by the contents of RISC-V register x29. One of the lower 16 bits of x28 specifies to which SCE the data is written, with bit 0 corresponding to SCE(0), bit 1 corresponding to SCE(1), bit 2 corresponding to SCE(2) and so forth. This instruction is available for both RV32I and RV64I versions of the RISC-V. The XSH instruction is not repeatable.

XSB

XSB stores the 8-bit contents of RISC-V register x30[7:0] into SCE memory/register/operator space pointed to by the contents of RISC-V register x29. One of the lower 16 bits of x28 specifies to which SCE the data is written, with bit 0 corresponding to SCE(0), bit 1 corresponding to SCE(1), bit 2 corresponding to SCE(2) and so forth. This instruction is available for both RV32I and RV64I versions of the RISC-V. The XSB instruction is not repeatable.

XPUSHD (push dual operand)

XPUSHD simultaneously pushes as *operandA* the contents of the location in RISC-V three-port data memory pointed to by the address value contained in x30[31:0] and the contents of the location in RISC-V data memory pointed to by the address value contained in x27[31:0], into the SCE operator input location pointed to by the address value contained in RISC-V register x29[31:0]. If the REPEAT value represented by bits [11:0] of RISC-V register x31 is not already zero, then this REPEAT value in x31 is decremented by 1. While the REPEAT value is not zero, the RISC-V instruction fetch address is frozen until the REPEAT value reaches 0x00000001.

The lower 16 bits of RISC-V register x28 specify to which target cores the XPUSHD pertains, which can be any, some or all of them, meaning that the XPUSHD instruction can be used to push dual operands into more than one target simultaneously.

The 3-bit source *Size* specifier for both *operandA* and *operandB* reads from RISC-V three-port data memory are contained in bits [22:20] of x31. The 3-bit destination *Size* specifier for the actual push into the SCE operator specified by x29 is contained in bits [14:12] of x31.

Bit [23] of x31 functions as a general-purpose *Signal* bit for the source, to signal hardware (if any) to perform an intermediate step during the simultaneous read of *operandA* and *operandB* from RISC-V three-port data memory. If no such intermediate hardware is present in an implementation of a specified target, this bit should be left in its default state of “0”. Bit [15] of x31 functions as a general-purpose *Signal* bit for the destination, to signal hardware (if any) to perform an intermediate step during the simultaneous push of *operandA* and *operandB* into the SCE operator specified by x29. If no such intermediate hardware is present in an implementation of a specified target, this bit should be left in its default state of “0”.

After the simultaneous push of *operandA* and *operandB* into the specified operator (which can reside as a stand-alone operator outside any SCE), the SCE destination address specified by RISC-V register x29 is automatically post-incremented by the encoded amount specified by bits [19:16] of RISC-V register x31. Simultaneous with this, the *operandA* and *operandB* source addresses specified by the contents of RISC-V registers x30 and x27 (respectively) are also automatically post-incremented by the encoded amount specified by bits [27:24] of RISC-V register x31. The automatic post-modify increment amount encodings are shown in the following table:

Code	Increment Amount
0000	1
0001	2
0010	4
0011	8
0100	16
0101	32
0110	64
0111	128
1000	0

Note that an increment amount of 0 is useful for situations, for example, where it is necessary to keep a source and/or destination address constant while using using the XPULL or XPUSH hybrid instruction. A good example of this is in scenarios where the application running on the RISC-V needs a block of pseudo-random floating-point numbers to initialize a neural network weight vector for training. With the RISC-V/SCE hybrid ISA this is easy to do. Simply load up the REPEAT value (minus 1) into RISC-V register x31[11:0], along with the appropriate source and destination *Size* bit fields of x31 as described above, then specifying a post-increment amount of 0 for the SCE PRNG source address specified by RISC-V register x29 and specifying an appropriate post-increment amount (in bytes) for the RISC-V data memory destination address specified by RISC-V register x30 and then execute the XPULL instruction. Once executed, the RISC-V will have pulled the specified quantity of pseudo-random numbers from the SCE PRNG and written them into consecutive locations in RISC-V data memory specified by x30.

It is also important to note that the SCE program memory is presently implemented using a 64-bit SRAM that can only be accessed on 8-byte boundaries. Hence, each instruction occupies exactly 1 (64-bit) location rather than 8 (byte) locations. Thus, to perform a XPUSH of a program into SCE program memory space using the REPEAT counter, the encoded destination address increment amount contained in RISC-V register x31 must be set to “0000”. It should also be noted that a given SCE program memory can only be accessed for reads and writes by the RISC-V by setting bit [31] of RISC-V register x29. To access SCE data memory, registers or operators, bit [31] of RISC-V register x29 must remain “0”.

XPUSH

XPUSH is similar to XPUSHD, except it reads a single data value/operand instead of dual operands. XPUSH pushes the contents of the location in RISC-V two or three-port data memory pointed to by the address value contained in x30[31:0] into the SCE operator input, register, program memory or data memory location pointed to by the address value contained in RISC-V register x29[31:0]. If the REPEAT value represented by bits [11:0] of x31 is not already zero, then this REPEAT value in x31 is decremented by 1. While the REPEAT value is not zero, the RISC-V instruction fetch address is frozen until the REPEAT value reaches 0x00000001.

The lower 16 bits of RISC-V register x28 specify to which target cores the XPUSH pertains, which can be any, some or all of them, meaning that the XPUSH instruction can be used to push dual operands into more than one target simultaneously.

The 3-bit source *Size* specifier for the read from RISC-V two or three-port data memory are contained in bits [22:20] of x31. The 3-bit destination *Size* specifier for the actual push into the SCE operator specified by x29 is contained in bits [14:12] of x31.

Bit [23] of x31 functions as a general-purpose *Signal* bit for the source, to signal hardware (if any) to perform an intermediate step during the read of *operandA* /data from RISC-V two or three-port data memory. If no such intermediate hardware is present in an implementation of a specified target, this bit should be left in its default state of "0". Bit [15] of x31 functions as a general-purpose *Signal* bit for the destination, to signal hardware (if any) to perform an intermediate step during the push of *operandA* into the SCE operator specified by x29. If no such intermediate hardware is present in an implementation of a specified target, this bit should be left in its default state of "0".

After the simultaneous push of *operandA* or data into the specified SCE operator/register/memory location (which can reside as a stand-alone operator outside any SCE), the SCE destination address specified by RISC-V register x29 is automatically post-incremented by the encoded amount specified by bits [19:16] of RISC-V register x31. Simultaneous with this, the data source address specified by the contents of RISC-V registers x30 is also automatically post-incremented by the encoded amount specified by bits [27:24] of RISC-V register x31. The automatic post-modify increment amount encodings are shown in the following table:

Code	Increment Amount
0000	1
0001	2
0010	4
0011	8
0100	16
0101	32
0110	64
0111	128
1000	0

Note that an increment amount of 0 is useful for situations, for example, where it is necessary to keep a source and/or destination address constant while using using the XPULL hybrid instruction. A good example of this is in scenarios where the application running on the RISC-V needs a block of pseudo-random floating-point numbers to initialize a neural network weight vector for training. With the RISC-V/SCE hybrid ISA this is easy to do. Simply load up the REPEAT value (minus 1) in RISC-V register x31, along with the appropriate source and destination *Size* bit fields of x31 as described above, then specifying a post-increment amount of 0 for the SCE PRNG source address specified by RISC-V register x29 and specifying an appropriate post-increment amount (in bytes) for the RISC-V data memory destination address specified by RISC-V register x30 and then execute the XPULL instruction. Once executed, the RISC-V will have pulled the specified quantity of pseudo-random numbers from the SCE PRNG and writen them into consecutive locations in RISC-V data memory.

It is also important to note that the SCE program memory is presently implemented using a 64-bit SRAM that can only be accessed on 8-byte boundaries. Hence, each instruction occupies exactly 1 (64-bit) location rather than 8 (byte) locations. Thus, to perform a XPUSH of a program into SCE program memory space using the REPEAT counter, the encoded destination address increment amount contained in RISC-V register x31 must be set to "0000". It should also be noted that a given SCE program memory can only be accessed for reads and writes by the RISC-V by setting bit [31] of RISC-V register x29. To access SCE data memory, registers or operators, bit [31] of RISC-V register x29 must remain "0".

XPULL

XPULL is similar to the XPUSH instruction, except in reverse. It pulls SCE data or program memory, register or operator result buffer contents from the SCE memory/register/operator location specified by RISC-V register x29[31:0] and writes such data into RISC-V two or three-port data memory location specified by the contents of RISC-V register x30[31:0]. If the REPEAT value represented by bits [11:0] of x31 is not already zero, then this REPEAT value in x31[11:0] is decremented by 1. While the REPEAT value is not zero, the RISC-V instruction fetch address is frozen until the REPEAT value reaches 0x00000001.

The lower 16 bits of RISC-V register x28 specify from which target core the XPULL instruction pertains. Thus, for the XPULL instruction, the value in x28 must be in one-hot form, meaning only one bit can be set to "1" at a time, with the other remaining bits cleared to "0".

The 3-bit destination *Size* specifier for the read from RISC-V two or three-port data memory are contained in bits [22:20] of x31. The 3-bit source *Size* specifier for the actual push into the SCE operator specified by x29 is contained in bits [14:12] of x31.

Bit [23] of x31 functions as a general-purpose *Signal* bit for the destination, to signal hardware (if any) to perform an intermediate step during the write of *operandA* /data to RISC-V two or three-port data memory. If no such intermediate hardware is present in an implementation of a specified target, this bit should be left in its default state of "0". Bit [15] of x31 functions as a general-purpose *Signal* bit for the source, to signal hardware (if any) to perform an intermediate step during the pull of *operandA* or data from the SCE operator, register/memory location specified by x29[31:0]. If no such intermediate hardware is present in an implementation of a specified target, this bit should be left in its default state of "0".

After the pull/read of *operandA* or data from the specified SCE operator/register or memory location (which can reside as a stand-alone operator outside any SCE), the SCE source address specified by RISC-V register x29 is automatically post-incremented by the encoded amount specified by bits [19:16] of RISC-V register x31. Simultaneous with this, the RISC-V data memory destination address specified by the contents of RISC-V registers x30 is also automatically post-incremented by the encoded amount specified by bits [27:24] of RISC-V register x31. The automatic post-modify increment amount encodings are

shown in the following table:

Code	Increment Amount
0000	1
0001	2
0010	4
0011	8
0100	16
0101	32
0110	64
0111	128
1000	0

Note that an increment amount of 0 is useful for situations, for example, where it is necessary to keep a source and/or destination address constant while using using the XPULL hybrid instruction. A good example of this is in scenarios where the application running on the RISC-V needs a block of pseudo-random floating-point numbers to initialize a neural network weight vector for training. With the RISC-V/SCE hybrid ISA this is easy to do. Simply load up the REPEAT value (minus 1) in RISC-V register x31, along with the appropriate source and destination *Size* bit fields of x31 as described above, then specifying a post-increment amount of 0 for the SCE PRNG source address specified by RISC-V register x29 and specifying an appropriate post-increment amount (in bytes) for the RISC-V data memory destination address specified by RISC-V register x30 and then execute the XPULL instruction. Once executed, the RISC-V will have pulled the specified quantity of pseudo-random numbers from the SCE PRNG and writen them into consecutive locations in RISC-V data memory.

It is also important to note that the SCE program memory is presently implemented using a 64-bit SRAM that can only be accessed on 8-byte boundaries. Hence, each instruction occupies exactly 1 (64-bit) location rather than 8 (byte) locations. Thus, to perform a XPUSH of a program into SCE program memory space using the REPEAT counter, the encoded destination address increment amount contained in RISC-V register x31 must be set to “0000”. It should also be noted that a given SCE program memory can only be accessed for reads and writes by the RISC-V by setting bit [31] of RISC-V register x29. To access SCE data memory, registers or operators, bit [31] of RISC-V register x29 must remain “0”.

XFRCBRK

XFRCBRK forces a hardware breakpoint on the target SCEs specified by 16 bits of RSIC-V register x28, with bit 0 corresponding to SCE(0), bit 1 corresponding to SCE(1), bit 2 corresponding to SCE(2) and so forth. Once set, the force break bit in the specified target SCEs will remain set until explicitly cleared by the RISC-V using the XCLRBRK instruction.

It is important to note that upon device reset, the force-break bit of all SCEs are automatically set as default. The main reason for this is due to the fact that upon power-on reset, each SCE's program memory will have nothing in it. Thus, by forcing a hardware breakpoint on power up, the SCEs will remain in a break state until the RISC-V pushes a program into them to execute.

Once a program or thread is pushed into a given SCE by the RISC-V, the RISC-V can then use the XCLRBRK instruction to clear the hardware break of one or more SCEs simultaneously. Always remember to specify which SCEs the XCLRBRK pertains using RISC-V register x28.

Once the hardware break is cleared, the specified SCE must be explicitly stepped out of the break state by using the XSSTEP instruction, otherwise the PC will remain where it is, which, upon reset, is always 0x00000100.
It should be noted that, whether at a breakpoint or not, the RISC-V can modify any one, some or all SCE program counters at will. Meaning that if an application requires one, some or all SCEs to begin executing from some location other than 0x00000100, all it has to do is use the XSW instruction to push a desired start address into the target SCEs specified by RISC-V register x28.

Further, it should be understood that a given SCE need not be at a software or hardware breakpoint before the RISC-V is able to read or write to any of the SCE's memory locations, registers or operators, meaning that the RISC-V can access the SCE at will, on-the-fly and in real-time, just as easily as the RISC-V access its own registers and memory.

XCLRBRK

As mentioned above, the XCLRBRK instruction clears the force hardware break bit of the target SCE(s) specified by RISC-V x28. As stated above, once the hardware break is cleared, the specified SCE must be explicitly stepped out of the break state by using the XSSTEP instruction, otherwise the PC will remain where it is, which, upon reset, is always 0x00000100.
It should be noted that, whether at a breakpoint or not, the RISC-V can modify any one, some or all SCE program counters at will. Meaning that if an application requires one, some or all SCEs to begin executing from some location other than 0x00000100, all it has to do is use the XSW instruction to push a desired start address into the target SCEs specified by RISC-V register x28.
Further, it should be understood that a given SCE need not be at a software or hardware breakpoint before the RISC-V is able to read or write to any of the SCE's memory locations, registers or operators, meaning that the RISC-V can access the SCE at will, on-the-fly and in real-time, just as easily as the RISC-V access its own registers and memory.

XFRCRST

When fetched and executed by the RISC-V, XFRCRST forces a hardware reset on the target SCE(s) specified by RISC-V register x28. Once set, the force reset bit for the specified SCE(s) will remain set until either a external system reset or the RISC-V fetching and executing the XCLRRST instruction.

XCLRRST

When fetched and executed by the RISC-V, XCLRRST clears the hardware reset bit of the target SCE(s) specified by RISC-V register x28.

XSSTEP

Once a hardware breakpoint is encountered by one or more SCEs, the XSSTEP instruction can be used to single-step the target SCEs specified by RISC-V register x28 one instruction at a time. Execution of the XSSTEP instruction produces a one-shot, single-step pulse that the specified target SCE(s) respond to by advancing the PC to the next instruction. As mentioned above, the XSSTEP instruction is also used to step a specified SCE out of a hardware breakpoint once encountered. For more information on hardware breakpoints, refer to the XFRCBRK section above.

XSTATUS

The XSTATUS instruction, once executed by the RISC-V, simply copies the 7 status bits of the target SCE specified by X28 into RISC-V register pair {x31[31:0], x30[31:0]}, with the most significant bits being padded with “0”. These 7 status bits are described as follows:

Bit	Name	Description
x30[0]	frcReset	this is the force-reset bit set by the XFRCRST instruction
x30[1]	frcBreak	this is the force h/w break bit set by the XFRCBRK instruction
x30[2]	broke	this bit, when set, indicates that a h/w breakpoint was encountered
x30[3]	swbrkDetect	when set, indicates that a software breakpoint has been encountered
x30[4]	stepCmplt	when set, indicates that an issued single-step has been completed
x30[5]	done	reflects the state of the target SCE Status Register “Done” bit
x30[6]	IRQ	reflects the state of the target SCE Interrupt Request input

RISC-V/SCE Hybrid ISA Demonstration Program

The following program demonstrates usage of the new RISC-V/SCE hybrid instructions. The exemplary Verilog RTL model of the RISC-V/SCE hybrid Instruction Set Architecture Compute Engine depicted in the first Page of this *Preliminary Information* sheet comprises a RISC-V (RV32I) core gene-spliced to qty. (2) ***SYMPL*** Compute Engine cores, SCE(0) and SCE(1), forming a RISC-V/SCE hybrid Compute Engine, with such cores being able execute the same hybrid instructions.

SCE(0) and SCE(1) are identical in all respects, except SCE(1) does not include the SYMPL Fused Universal Neural Network (FuNN) operator or the human-readable Pseudo-Random Number Generator that SCE(1) has. The purpose of SCE(1) being included in the instant implementation is to demonstrate the RISC-V hybrid core's ability to push programs and/or data into two or more SCEs, set or clear hardware breakpoints, single-step, launch and multiple SCE cores simultaneously.

The Verilog test bench that is included with the distribution package available for free download at the **SYMPL** Compute Engine repository here, www.github.com/jerry-D, begins by first loading the RISC-V hybrid's program memory that the RISC-V executes to orchestrate the processing that the SCEs are to perform. Next, the test bench loads the RISC-V hybrid's data memory with the program that the SCEs are to execute, followed by the test bench loading the data file that SCE(0) is to process. While SCE(1) does not include a FuNN operator, this same data file is simultaneously pushed into it as well, mainly for the purpose of demonstrating simultaneous loads by the RISC-V hybrid.

The program that is to be pushed into both SCEs simultaneously the RISC-V hybrid is a simple neural network classification routine that SCE(0) executes to classify qty. (16) objects using SCE(0)'s on-board FuNN operator. SCE(1) executes the same program, but since it has no FuNN operator with which to perform the required operations, SCE(1) will eventually get trapped in a loop that attempts to read a result when there is none.

Once the test bench pushes both the program and the data files into the SCEs, it releases RESET, allowing the RISC-V hybrid core to run its program. Once those things are accomplished, the test bench will sit and wait for an indication that the RISC-V/SCE hybrid Compute Engine has completed processing, at which point the test bench writes formats and writes the results to two files in the Vivado simulation directory for that project. The two file names are “assayPullForm.txt” and “randomNumbers.txt”.

The file assayPullForm.txt” is a formatted, human-readable report showing the results of the neural network routine that SCE(0) computed using its on-board FuNN operator. The file “randomNumbers.txt” is an unformatted, is a list of qty. (256) human-readable floating-point pseudo-random floating-point numbers in decimal character sequence format that the RISC-V hybrid pulled directly from SCE(0)'s on-board PRNG.

RISC-V Launching SCEs

During RESET active, the SCE's force-break bit is automatically set to active, thus when RESET is released, the force-break in each SCE remains set, forcing each SCE to enter a continuous hardware break state. The main purpose of this is to allow an opportunity for the RISC-V hybrid to push a program and data into the SCEs, in that upon power-on reset, the SCE program memories have nothing in them for the SCEs to execute. Once programs and data have been pushed into the SCE's, the RISC-V hybrid then simultaneously clears both SCEs' force-break bit by executing the XCLRBRK hybrid instruction immediately followed by the XSSTEP hybrid instruction. This short sequence clears both SCEs' force-break bits, thereby allowing both SCEs begin executing their own programs now residing in their own program memories. The XSSTEP instruction following the XCLRBRK instruction is necessary because, while XCLRBRK does clear the SCE's force-break bit, the SCE will remain in a hardware break state until it is explicitly “stepped” out of that state by the RISC-V hybrid using the XSSTEP instruction.

The first part of the SCE neural network classification program initializes various SCE registers, which in this case, is necessary because the SCE just came out of RESET state. Once these few registers have been initialized, the SCEs then encounter a software breakpoint residing at location 0x0000010A in the SCEs' program memory. The RISC-V hybrid's routine anticipates this and therefor waits for this to happen by polling SCE(0)'s status bits using the XSTATUS hybrid instruction.

Once the RISC-V hybrid detects that the software breakpoint has been encountered, it then simultaneously modifies both SCE program counters to point to the entry point of the SCEs' neural network classification routine, which is location 0x0000010C in SCE program memory. The RISC-V hybrid also simultaneously pushes into both SCEs' Auxillary Register r (AR4), the number of objects SCE(0) is to classify, which, in the case is “16”, in that SCE(0) classification program expects that its AR4 contain the number of objects to classify.

After the RISC-V hybrid simultaneously modifies both SCE's program counter, the SCEs immediately begin executing the neural network classification program using the data that the RISC-V simultaneously pushed into their data memories. While the SCEs are busy performing the classification routine, the RISC-V is continually polling SCE(0) status bits for a “Done” condition. When the classification routine is completed by SCE(0), it sets is “Done” bit to signal this fact, at which point the RISC-V detects this and begins pulling results from SCE(0)'s output buffer located within SCE(0)'s data memory and storing it in RISC-V data memory space using the XPULL hybrid instruction.

Finally, after the RISC-V has pulled the results from SCE(0) output buffer, the RISC-V enters a “forever” loop of NOPs, which the test bench detects.

Here is the exemplary program the RISC-V hybrid executes to orchestrate processing by more than one SCE:

```
;-----------------------------------------------------------------------------------------------------------
; these 5 instructions tell SCE(0) PRNG to start generating random floating-point numbers less than 2.0
;-----------------------------------------------------------------------------------------------------------
start:      mvi     x28, 0x00000001    ;specify to which SCE the following operations concerns
            lui     x29, PRNG          ;point to SCE psuedo-random number generator
            srli    x29, x29, 12       ;shift x29 right by 12 bits to do this immediate address load
            mvi     x30, 37            ;load x30 with code for desired PRNG range of numbers to generate
            XSW                        ;store 37 to PRNG range control register.
;-----------------------------------------------------------------------------------------------------------
; these 6 instructions push the SCE program into both SCEs simultaneously
;-----------------------------------------------------------------------------------------------------------
            mvi     x28, 0x00000003    ;point to both SCE(0) and SCE(1) cores
            lui     x29, 0x80000       ;point to first location of SCE program memory
            lui     x31, 0x03303       ;load x31 with size and increment amounts for each individual push
            ori     x31, x31, 0x170    ;load REPEAT counter in x31[11:0] with length of SCE program (minus 1)
            mvi     x30, 0x00000000    ;the SCE program resides starting at 0x00000000 in RISC-V data memory
            XPUSH                      ;push specified program from RISC-V data memory into SCE(0) and (SCE(1)
;-----------------------------------------------------------------------------------------------------------
;these 6 instr push qty. 16 object X and weight vectors (total of 4096 bytes) into SCE(0) and SCE(1) at same time
;-----------------------------------------------------------------------------------------------------------
            lui     x29, inbuf         ;place 20-bit pointer to SCE input buffer in upper 20 bits of x29
            srli    x29, x29, 12       ;shift x29 right by 12 bits to do this immediate address load
            lui     x31, 0x03333       ;load x31 with size and increment amounts for each individual push
            ori     x31, x31, 511      ;load repeat counter (lower 12 bits of x31) with repeat amount (minus 1)
                                       ;at least one instruction must follow load of repeat counter before XPUSH
            lui     x30, 0x00001       ;input data resides beginning at location 0x1000 in RISC-V data memory
            XPUSH                      ;push 4096-byte data block into SCE
;-----------------------------------------------------------------------------------------------------------
;these two instr launch SCE(0) and SCE(1) simultaneously
;-----------------------------------------------------------------------------------------------------------
            XCLRBRK                    ;clear the force hardware breakpoint automatically set on power-up reset
            XSSTEP                     ;step out of the h/w breakpoint and allow SCE to initialize itself
;-----------------------------------------------------------------------------------------------------------
; these 5 instr cause RISC-V to wait for SCE(0) to hit software breakpoint at SCE location 0x0000010A
;-----------------------------------------------------------------------------------------------------------
wait0:      mvi     x28, 0x00000001    ;specify to which SCE the following operations concerns
            XSTATUS                    ;copy specified SCE status bits into x30[7:0]
            andi    x30, x30, 8        ;swbrk status bit is bit x30[4]
            mvi     x31, 8             ;load x31 with constant 8, ie, set bit x31[3] to "1"
            bne     x30, x31, wait0    ;goto wait0 if x30[4] is not set
;-----------------------------------------------------------------------------------------------------------
; these 8 instr initialize SCE(0) AR4 and PC prior to launch
;-----------------------------------------------------------------------------------------------------------
            lui     x29, AR4           ;get address of SCE Auxiliary Register 4
            srli    x29, x29, 12       ;shift it right 12 bits for proper alignment
            mvi     x30, 16            ;load x30 with the number of objects to classify
            XSW                        ;store number objects directly into AR4 of SCE(s) specified by x28

            lui     x29, PC            ;get address of SCE program counter
            srli    x29, x29, 12       ;shift it right 12 bits for proper alignment
            mvi     x30, 0x10C         ;load x30 with the entry point to the routine
            XSW                        ;store entry point directly into PC of SCE(s) specified by x28
            nop                        ;do nothing for a few clocks, giving time for SCE to clear its DONE bit,
            nop                        ;indicating it is now busy. alternatively, you can manually test the bit
            nop                        ;but in this instance I'm being too lazy to code it
            nop
            nop
```

```
;-------------------------------------------------------------------------------------------------------------
;these 4 instr wait for SCE(0) to  hit software breakpoint at location 0x0000010A in program memory
;indicating it has completed the task
;-------------------------------------------------------------------------------------------------------------
wait1:      XSTATUS                 ;copy specified SCE status bits into x30[7:0]
            andi    x30, x30, 32    ;poll swbrk status bit x30[5]
            mvi     x31, 32         ;load x31 with constant 32, ie, set bit x31[5] to "1"
            bne     x30, x31, wait1 ;goto wait0 if x30[5] is not set, ie, SCE is still busy (ie, not done yet)
;-------------------------------------------------------------------------------------------------------------
;now pull the 4096-byte result out of SCE(0) and store it into RISC-V data memory
;-------------------------------------------------------------------------------------------------------------
            lui     x29, outbuf     ;place 20-bit pointer to SCE output buffer in upper 20 bits of x29
            srli    x29, x29, 12    ;shift x29 right by 12 bits to do this immediate address load
            lui     x31, 0x03333    ;load x31 with size and increment amounts for each individual push
            ori     x31, x31, 1279  ;there are 1280 8-byte words to pull out of SCE output buffer
            lui     x30, 0x02008    ;store result data starting at 0x2008 in RISC-V data memory
            srli    x30, x30, 12
            XPULL                   ;pull 4096-byte result data block from SCE
;-------------------------------------------------------------------------------------------------------------
;these 7 instructions demonstrate how the RISC-V has the ability to borrow any of the SCE's
;resources, in that any resource of the SCE is also a resource of the hybrid RISC-V
;-------------------------------------------------------------------------------------------------------------
            lui     x29, PRNG       ;load RISC-V x29 with pointer to SCE(0) pseudo-random number generator
            srli    x29, x29, 12    ;shift x29 right by 12 bits to do this immediate address load
            lui     x31, 0x0338B    ;load x31 with size and increment amounts for each individual push
                                    ;in this case signal "text" for decimal char sequence
ori     x31, x31, 255       ;256 8-byte decimal character floating point representations please
lui     x30, 0x04820        ;store generated numbers immediately above classification results
srli    x30, x30, 12        ;pull the 256 random numbers from SCE(0) PRNG and store them in
XPULL                       ;RISC-V data RAM starting at location 0x00004820
```

Example SYMPL FuNN eNNgine Classification Routine

For completeness, here is an example of an actual 16-input by 16-object classification routine that uses the FuNN eNNgine. It employs a single input layer with TanH activation. The output layer comprises a SoftMax activation layer followed by a HardMax activation layer. The SoftMax activation comprises three steps: 1) exponentiation of all 16 nodes/cells in the TanH layer, 2) summation of all exponentials, 3) division of each of the 16 exponentials by the summation. As the final step in the classification computation, all 16 results of the SoftMax layer are simultaneously pushed back into the neural network with HardMax activation set. All 16 results of the HardMax are simultaneously computed and stored in the final layer with only one of the results being a literal “1.0” to indicate maximum probability over the remaining 15 other nodes, which will all be “0.0”.

In summary, the following routine employs only 30 instructions to classify qty. (16) objects, with each object comprising qty. (16) inputs. The entire process consumes only 2310 clocks to complete. Classification of a single 16-input object consumes only 144 clocks. Every clock cycle there are 18 floating-point multiplies, 37 floating-point additions, 2 floating-point divisions, 5 exponentials and 256 floating-point compares. In addition, there are 32 convertFromDecimalCharacter conversions and two convertToDecimalCharacter conversions—every clock cycle.

```
0000010C 307CDC0000000010       classify:  _   _1:clrDVNCZ = _1:#DoneBit ;clear Done bit
0000010D 327FF72000013880                  _   _4:AR0 = _4:#_1.0         ;get pointer to vector where the "1.0" goes
0000010E 317FEF100000000F                  _   _2:REPEAT = _2:#15
0000010F 2380403000200000                  _   _8:*AR0++[8] = _8:@one    ;generate a vector of qty (16) "1.0"  as constant
00000110 0000000000000000                  _
00000111 327FFD2000010080                  _   _4:AR6 = _4:#outBuffer    ;get pointer to output buffer location
00000112 017FED17FFB00000                  _   _2:LPCNT0 = _2:AR4        ;AR4 already contains # of objects to classify,
                                                                         ;as Host CPU pushed it in from test bench
00000113 327FF72000012880                  _   _4:AR0 = _4:#object0      ;point to first object vector X
                                ;--------------------------------------
                                ; input layer (layer0) computation using TanH activation
                                ;--------------------------------------
00000114 327FF82000013080        loop:     _   _4:AR1 = _4:#obj0Lay0Wt   ;point to first weight vector W
00000115 327FF92000004000                  _   _4:AR2 = _4:#layer00      ;point to first layer results (layer0)
00000116 307FCD0000000007                  _   _1:actMode = _1:#TanH     ;set activation mode
00000117 317FEF100000000F                  _   _2:REPEAT = _2:#15
00000118 8B800AF8000F8401                  act  s8:*AR2++[1] = (s128:*AR0++[0], s128:*AR1++[128])  ;run all the weight W
00000119 0000000000000000                  _                             ;vectors against the current object X vector
0000011A 0200002840000000                  _   _4:0 = _4:*AR0++[128]     ;bump AR1 by 128 to point to next weight
                                ;--------------------------------------
                                ; SoftMax process
                                ;--------------------------------------
                                    ;--------------------------------------
                                    ; second layer (layer1) is intermediate output layer where exponentials are stored
                                    ;--------------------------------------
0000011B 327FF82000004000                  _   _4:AR1 = _4:#layer00      ;point to first layer results (layer0)
0000011C 327FFA2000013880                  _   _4:AR3 = _4:#_1.0         ;point to vector of 1.0s for use as exp weights
0000011D 307FCD0000000006                  _   _1:actMode = _1:#Exp      ;set activation mode to exponential
0000011E 317FEF100000000F                  _   _2:REPEAT = _2:#15        ;AR2 is already pointing to layer1
0000011F 8B800AB8009B8003                  act  s8:*AR2++[1] = (s8:*AR1++[1], s8:*AR3++[0]) ;exponentiate all the results
00000120 0000000000000000                  _                             ; in layer0 and store in layer1
                                    ;--------------------------------------
                                    ; summation of all the exponentials
                                    ;--------------------------------------
                                                                         ;AR1 is already pointing to layer1
                                                                         ;AR3 is already pointing to 1.0 vector
```



```
                                                    ;AR2 is already pointing to layer2
00000121 0B8082F8081F8003                 _   s8:*AR2++[16] = (s128:*AR1++[16], s128:*AR3++[0]) ;store single result in
00000122 0000000000000000                 _                                 ;position 0 of layer 2 (the 3rd layer)
                                      ;--------------------------------------
                                      ; divide each exponential by the sum
                                      ;--------------------------------------
                                                    ;AR1 is already pointing to layer2
00000123 327FFA2000004010                 _   _4:AR3 = _4:#layer01      ;point to the first exponential result in layer 1
                                                    ;AR2 is already pointing to layer3
00000124 307FCD000000000B                 _   _1:actMode = _1:#SoftMax  ;division e^xi/sum(e^xi)--actually just a
00000125 317FEF100000000F                 _   _2:REPEAT = _2:#15        ;division operation here
00000126 8B800AB800BB8001                 act  s8:*AR2++[1] = (s8:*AR3++[1], s8:*AR1++[0]) ;divide each exponential
00000127 0000000000000000                 _                                 ;of layer 1 by the sum of all of the exponentials
                                  ;--------------------------------------
                                  ; HardMax process
                                  ;--------------------------------------
00000128 327FF82000004030                 _   _4:AR1 = _4:#layer03      ;point to start of SoftMax result vector
00000129 327FFA2000013880                 _   _4:AR3 = _4:#_1.0         ;use weight of 1.0 for each node
                                                    ;AR2 is already pointing to layer4
0000012A 307FCD000000000C                 _   _1:actMode = _1:#HardMax  ;use HardMax activation mode
0000012B 8F8002F8001F8003                 act s128:*AR2++[0] = (s128:*AR1++[0], s128:*AR3++[0])
0000012C 0000000000000000                 _
0000012D 327FFC2000004000                 _   _4:AR5 = _4:#layer00      ;spill the 5 layers of each pass into the
0000012E 317FEF1000000004                 _   _2:REPEAT = _2:#4         ;output buffer for retrieval by the host
0000012F 0784067808500000                 _   _128:*AR6++[128] = _128:*AR5++[16] ;when process is done
00000130 0000000000000000                 _
00000131 127FF417FED43FE3                 _  _4:PCS = (_2:LPCNT0, 16, loop)   ;conditional load of PC with relative
00000132 0000000000000000                 _                                   ; address if specified bit is set
00000133 397FF52000000108                 _  s2:PC = _4:#done           ;go back to done--unconditional load of PC
                                                                        ;with absolute address
```

