

Connex Accelerator Instruction Set Architecture Specification

October 21, 2018

Contents

1	Instruction Formats	2
1.1	Opcode Formats	2
2	Instructions	3
2.1	Scalar Instructions	3
2.2	Vector Instructions	4
3	Similitude Connex ISA, LLVM IR (and Opincaa ASM)	6

1 Instruction Formats

The Connex accelerator utilizes a 32-bit Instruction Set Architecture (ISA). Instructions are divided into Scalar Instructions (SI) and Vector Instructions (VI). There are two main instruction formats, shown in Table 1. Register addresses are 5 bits in size, allowing for a maximum of 32 registers (SIMD or Scalar). The immediate value is 16 bits in size, requiring the removal of the right operand address and the use of a reduced opcode for immediate value instructions. The immediate value, when present, replaces the right operand in both the scalar and vector pipelines.

Instruction Type	Bit Offset					
	31:26	25:23	22:15	14:10	9:5	4:0
Immediate Value	OPCODE	IMMEDIATE VALUE			LEFT	DEST
Non Immediate	OPCODE		RESERVED	RIGHT	LEFT	DEST

Table 1: Instruction Formats

1.1 Opcode Formats

The Connex opcode is 6 or 9 bits in length and is always present on the most-significant bits of the instruction. The opcode consists of a 3-bit fixed section and a 6-bit variable section which is formatted differently depending on the contents of the fixed section. The PIPE bit is always present at offset 8 and specifies whether the instruction is vector (PIPE=1) or scalar (PIPE=0). The IMM bit is always present at offset 7 and specifies whether the instruction is Immediate-Value (IMM=1) or Non-Immediate (IMM=0). The ALU bit is always present at offset 6 and specifies whether the instruction utilizes ALU (ALU=1) or other processing resources (ALU=0). When ALU is set the instruction always writes back results to the register file.

Opcode formats for Vector Instructions are listed in Table 2. The WB bit is present if ALU is not set and specifies if the instruction writes back results to the register file (WB=1) or does not write back (WB=0). The NON-ALU SEL field specifies which processing resource is targeted by the instruction. Table 3 shows the resources selected by the values of NON-ALU SEL. When IMM is set, bit 0 of NON-ALU SEL is set. This enables access of Immediate-Value instructions only to the Local Store and Immediate Value instruction field.

The NON-ALU SEL field specifies which processing resource is targeted by the instruction. Table 3 shows the resources selected by the values of NON-ALU SEL. When IMM is set, bit 0 of NON-ALU SEL is set. This enables access of Immediate-Value instructions only to the Local Store and Immediate Value instruction field.

The OP field is present if ALU is set and specifies which type of operation is selected inside the ALU. Table 4 shows available operation types. When IMM is set, bit 0 of OP is set. This enables access of Immediate Value instructions only to Arithmetic and Logical operations

The SUB-OP field selects the particular operation to be executed within an operation type. Table 5 shows how SUB-OP values correspond to ALU operations.

Bit Offset								
8	7	6	5	4	3	2	1	0
PIPE	IMM	ALU						
1	0	0	WB	NON-ALU-SEL			MODIFIERS	
		1	SUB-OP		OP			
	1	0	WB	NON-ALU-SEL[2:1]			1	
		1	SUB-OP		OP[1]		1	

Table 2: VI Opcode Formats

NON-ALU SEL Value	Accessed Resource
000	Index Read
100	Inter-Cell Shift
001	Local Store Read
101	Local Store Write
010	Multiply Read
110	Extension Register Read
011	Immediate Value Read
111	Cell Enable

Table 3: NON-ALU SEL Values

OP Value	Operation Type
00	Shift/Popcount
01	Arithmetic
10	Comparison
11	Logical

Table 4: OP Values

Op Type	SUB-OP Value	Operation	Op Type	SUB-OP Value	Operation
Shift Popcount	00	Left Shift Logical	Comparison	00	Equal
	01	Right Shift Logical		01	Signed Less
	10	Right Shift Arithmetic		10	Unsigned Less
	11	Popcount		11	Reserved
Arithmetic	00	Sum	Logical	00	Logical Not
	01	Difference		01	Logical Or
	10	Sum with Carry		10	Logical And
	11	Difference with Carry		11	Logical Xor

Table 5: SUB-OP Values

2 Instructions

2.1 Scalar Instructions

Scalar instructions (SI) follow the same formats as vector instructions. The PIPE bit is not set for scalar instructions. Scalar instructions affect two scalar registers:

- LC loop counter, specifies how many times a subsequent jump will execute

- PC program counter, indicates where instructions are fetched from, in the current instruction stream

Table 6 lists the scalar instructions and their behaviour.

Mnemonic	Description	Opcode
nop	No operation	000000000
setlc	LC = Immediate Value	10101
ijmpnzdec	Require: Immediate Value <1023 If (LC != 0): $PC = PC - ImmediateValue$ $LC = LC - 1$ If(LC == 0): $PC = PC + 1$ LC reverts to initial value	10011

Table 6: Scalar Instructions

2.2 Vector Instructions

Table 7 presents all vector instructions. Some instructions execute conditionally upon the value of the Active flag, i.e., if Active is not set, they behave as nop. Certain instructions set the carry, less and equal flags:

- the carry flag is set by:
 - **add**, when $R[left] + R[right]$ overflows,
 - **addc**, when $R[left] + R[right] + \text{carry}$ overflows,
 - **sub**, when $R[left] - R[right]$ underflows,
 - **subc**, when $R[left] - R[right] - \text{carry}$ underflows,
- the less flag is set by **lt** when $R[left]$ is less than $R[right]$,
- the equal flag is set by **eq** when $R[left]$ is equal to $R[right]$,

In the table, referring to flags, a U entry indicates undefined (data dependent) values of flags after the execution of the instruction. Where no value is indicated, the instruction does not modify flags. Otherwise, the instruction may behave, with regard to a particular flag, in an identical way to add, addc, sub, subc, lt, or eq.

Notes:

- Memory instructions **write**, **iwrite** and **read** (except **iread**) require the insertion of a delay slot of one cycle between them and the instruction(s) that generate their operands. Following are all relevant examples we can have with delay slots:

$$R1 = R2 + R3$$

NOP // or other instruction to fill the delay slot

$$R4 = LS[R1] / LS[R1] = R4 / LS[R10] = R1 / LS[5] = R1$$

- It is necessary to insert a delay slot of one cycle between selection instructions (**wherexx**) and the instruction which affects the flag utilized for selection. For example:

```

R1 = (R2 == R3)
NOP // Alex: or other instruction that does not
    // alter Equal flag to fill the delay slot
WHERE_EQUAL

```

- When all or some of the cells are disabled, reduction operations with operands from the local store (code example: `R1 = LS[15]` then immediately `REDUCE(R1)`) return an undefined result. The programmer must ensure that all cells are re-enabled, by issuing an `endwhere` instruction, before any such reduction occurs.
- At power-up, `Active` is zero (i.e., the cell's register file and local store will be disabled) until an **endwhere** instruction is received.
- There is no explicit **MOV** (move instruction), but the programmer can move data from one register to the other in several ways:
 - `ishl R0, R1, 0` (produces undefined flags)
 - `ishr R0, R1, 0` (produces undefined flags)
 - `ishra R0, R1, 0` (produces undefined flags)
 - `or R0, R1, R1` (recommended, produces constant flags: `Carry = 0 Less = 0 Equal = 1`)
 - `and R0, R1, R1` (recommended, produces constant flags: `Carry = 0 Less = 0 Equal = 1`)
- Regarding the functionality:
The shift vector unit seems to contain 2 architecturally non-visible vector registers, which are actually continuously operated by the unit: a 1st register with values to be moved around and a 2nd register with movement directions, which should have only non-negative values. The cell-shift instructions take as input 2 (vector) register operands, which are copied, respectively, in the architecturally non-visible registers of the shift vector unit. These instructions take normally several cycles to finish (i.e., the shift vector unit to converge, to obtain a "steady-state" result). The **ldsh** instruction retrieves the result from the shift vector unit.

The **cellshl** instruction decreases in each cycle by at most 1 unit each value of the 2nd register if not zero, until all the values of the 2nd register become zero.

During each cycle of execution of **cellshl**, each element of the 1st register is copied from the immediate/neighbor right cell, (modulo number of lanes, i.e., it considers the register to be wrapped around) if the corresponding element of the 2nd register is not zero (we look for zero at the current element, not in the neighbor right cell), in which case this latter value is also decremented. Due to the modulo operation, the cell-shift instruction experiences also a rotate effect.

Example of execution of the instruction *cellshl R0, R1*, where `R0 = [3 4 5 6]`, `R1 = [0 1 2 2]` (we assume the number of lanes of Connex is 4):

Before cycle 1:

```

1st reg:  3 4 5 6    // the data is loaded in the 1st register
2nd reg:  0 1 2 2    // the move directions are loaded in the 2nd register

```

```

End of cycle 1:
  1st reg:  3 5 6 3
  2nd reg:  0 0 1 1

```

```

End of cycle 2:
  1st reg:  3 5 3 3
  2nd reg:  0 0 0 0

```

Key takeaways: **cellshl** puts values to the left, while **cellshr** puts values to the right. The number of cycles to execute these operations can be considered equal to the maximum value of the 2nd vector operand.

3 Similitude Connex ISA, LLVM IR (and Opincaa ASM)

The Connex *Instruction Set Architecture (ISA)* is presented in Table 8. The Connex *Instruction Set Architecture (ISA)* contains arithmetic, bitwise logical, logical, memory access and **nop** instructions. It also has sum-reduce (**red**, in Opincaa represented by *REDUCE*), shift vector (**cellshl/r**, which moves the data in the vector and **ldsh**, which reads the shift vector), block predication instructions (**whereeq/lt/cry** and **endwhere**, which have in Opincaa corresponding instructions starting with *EXECUTE*) and loop with counter instructions (**setlc** and **ijmpnzdec**, in Opincaa represented by *REPEAT(imm)* and *END_REPEAT*). A complete description of the ISA is given in [3] and [1]. As we can see, Connex has rather limited control flow instructions: only loops of constant trip counts and a predication mechanism using the Boolean values of the Carry, Less or Equal flags, set previously for each lane. It does not have instructions such as call or conditional branch, available, for example, in NVIDIA GPU’s PTX assembly [5]. While the lack of branches implies there is no control divergence, the Connex predicated blocks can be arbitrarily large and the inherent inefficiency of having threads executing conditional code on a SIMD processor remains.

As already discussed, all instructions take 1 cycle, except the sum reduction, which takes $\log_2(CVL)$ cycles, and the shift vector operations, which take at the maximum CVL cycles under normal conditions.

We note that the *red* Connex instruction performs sum reduction over a vector of 16-bit *unsigned* elements, which is required for the efficient implementation of reduction for 32-bit (or 64, etc) integer element vectors - for example, we can send from the CPU to the standard Connex with 128 (16-bit) lanes a vector of 64 32-bit integers for reduction, which fills a vector line of the accelerator. We should also have a *red* Connex instruction for vectors of 16-bit *signed* elements. Similarly, the multiplication is performed on unsigned 16-bit integer vector operands in order to allow the efficient multiplication of 32-bit (or larger) signed integers. An interesting property is that the lowest 16 bits of the 32-bit result, returned by the **multlo** instruction, is the same for signed and unsigned multiplication, so we should normally add to the Connex ISA only a **multhi** for 16-bit signed integer input operands.

The LLVM IR vector instructions were first described in [2]. They have little predication support: only the vector load and store, gather and scatter are masked.

The LLVM language is in most aspects more higher level than the Connex assembly language, which we target to compile to. LLVM has arithmetic, bitwise logical and logical instructions that work on scalar and also vector operands. LLVM's *select* instruction can be translated to Connex using its **where** and arithmetic operations. The *shufflevector* and *extractelement* LLVM instructions with arbitrary arguments are rather difficult to translate efficiently to Connex. On the other hand, Connex's **vload** instruction can be represented in LLVM IR with an *insertelement* followed by a *shufflevector*. Also, the LLVM IR masked gather and scatter intrinsics have an equivalent in the **read** and **write** Connex instructions.

Opincaa is an everything (decomposition, mapping, communication and synchronization) explicit parallel programming model [6]. Opincaa is similar to OpenCL [4], only that it targets just the Connex accelerator and writes kernels for it in vector assembly language instead of OpenCL C.

References

- [1] C. Bîră, L. Petrică, and R. Hobincu. OPINCAA: A Lightweight and Flexible Programming Environment For Parallel SIMD Accelerators. *Romanian Journal of Information Science and Technology*, 16(4), 2013.
- [2] R. L. Bocchino, Jr. and V. S. Adve. Vector LLVA: A Virtual Vector Instruction Set for Media Processing. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, VEE '06, pages 46–56, New York, NY, USA, 2006. ACM.
- [3] Gheorghe M. Ștefan. The Connex Instruction Set Architecture, 2015.
- [4] A. Munshi, B. Gaster, T. G. Mattson, J. Fung, and D. Ginsburg. *OpenCL Programming Guide*. Addison-Wesley Professional, 1st edition, 2011.
- [5] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2013.
- [6] D. B. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Comput. Surv.*, 30(2):123–169, June 1998.

Mnemonic	Description	Condition	Opcode	Carry Flag	Less Flag	Equal Flag
nop	No operation		000000000			
red	Launch reduction with R[left]		100000000			
iwrite	LS[Immediate Value] = R[left]	Active	110010			
iread	R[dest] = LS[Immediate Value]	Active	110100			
write	LS[R[right]] = R[left]	Active	100010100	Sub	Lt	Eq
read	R[dest] = LS[R[right]]	Active	100100100			
vload	R[dest] = Immediate Value	Active	110101			
ldix	R[dest] = INDEX	Active	100100000			
endwhere	Enable All Cells (set Active everywhere)		100011111			
wherecry	Load Carry Flag into Active		100011100			
whereeq	Load Equal Flag into Active		100011101			
wherelt	Load Less Flag into Active		100011110			
mult	Initiate R[left] * R[right]		100001000	Add	Lt	Eq
multlo	R[dest] = Low half of multiplication result	Active	100101000			
multhi	R[dest] = High half of multiplication result	Active	100111000			
cellshr	Shift Register = R[left] then shift right by R[right] positions		100010001	Sub	Lt	Eq
cellshl	Shift Register = R[left] then shift left by R[right] positions		100010010	Sub	Lt	Eq
ldsh	R[dest] = Shift Register	Active	100110000			
add	R[dest] = R[left] + R[right]	Active	101000100	Add	Lt	Eq
sub	R[dest] = R[left] - R[right]	Active	101010100	Sub	Lt	Eq
addc	R[dest] = R[left] + R[right] + Carry	Active	101100100	Addc	Ult	Eq
subc	R[dest] = R[left] - R[right] - Carry	Active	101110100	Subc	Ult	Eq
eq	R[dest] = (R[left] == R[right]) ? 1:0	Active	101001000	Add	Lt	Eq
ult	R[dest] = (R[left] <R[right]) ? 1:0 (unsigned)	Active	101101000	Addc	Ult	Eq
lt	R[dest] = (R[left] <R[right]) ? 1:0	Active	101011000	Sub	Lt	Eq
shl	R[dest] = R[left] <<R[right]	Active	101000000	Add	Lt	Eq
ishl	R[dest] = R[left] <<right	Active	101000001	U	U	U
shr	R[dest] = R[left] >>R[right]	Active	101010000	Sub	Lt	Eq
ishr	R[dest] = R[left] >>right	Active	101010001	U	U	U
shra	R[dest] = R[left] >>>R[right]	Active	101100000	Addc	Ult	Eq
ishra	R[dest] = R[left] >>>right	Active	101100001	U	U	U
popcount	R[dest] = Sum of bits of R[left]	Active	101110000			
not	R[dest] = ~R[left]	Active	101001100	U	U	U
or	R[dest] = R[left] — R[right]	Active	101011100	Sub	Lt	Eq
and	R[dest] = R[left] & R[right]	Active	101101100	Addc	Ult	Eq
xor	R[dest] = R[left] ^ R[right]	Active	101111100	Subc	Ult	Eq

Table 7: Vector Instructions

Category	Opincaa Connex Instructions
arithmetic	$R(d) = R(s1) + R(s2); //$ add $R(d) = R(s1) - R(s2); //$ sub $R(d) = R(s1) + R(s2) + \text{carry}; //$ addc $R(d) = R(s1) - R(s2) + \text{carry}; //$ subc
	$R(s1) * R(s2); //$ mult , initialize multiplication $R(d_l) = \text{MULT_LOW}()$ and $R(d_h) = \text{MULT_HIGH}()$ <i>/* multlo and multhi, get 16-bit lower and higher part of result of multiplication */</i>
sum-reduce	$\text{REDUCE}(R(s)); //$ red , result has 32 bits
bitwise logical	$R(d) = \sim R(s); //$ not $R(d) = R(s1) R(s2); //$ or $R(d) = R(s1) \& R(s2); //$ and $R(d) = R(s1) \wedge R(s2); //$ xor
	$R(d) = R(s1) \ll R(s2); //$ shl $R(d) = R(s1) \ll \text{imm}; //$ ishl $R(d) = R(s1) \gg R(s2); //$ shr $R(d) = R(s1) \gg \text{imm}; //$ ishr $R(d) = \text{SHRA}(R(s1), R(s2)); //$ shra $R(d) = \text{ISHRA}(R(s), \text{imm}); //$ ishra
	$R(d) = \text{POPCNT}(R(s)); //$ popcount , bits sum
logical	$R(d) = R(s1) == R(s2); //$ eq $R(d) = R(s1) < R(s2); //$ lt $R(d) = \text{ULT}(R(s1), R(s2)); //$ ult
shift vector	<i>/* load in shift register vector R(s1), then shift left/right by R(s2) positions */</i> $\text{CELL_SHL}(R(s1), R(s2)); //$ cellshl $\text{CELL_SHR}(R(s1), R(s2)); //$ cellshr
	<i>/* load in R(d) the current value of the shift register */</i> $R(d) = \text{SHIFT_REG}; //$ ldsh
load/store	$R(d) = \text{LS}[\text{imm}]; //$ iread , imm.-addr. load $R(d) = \text{LS}[R(s)]; //$ read , indirect load
	$\text{LS}[\text{imm}] = R(s); //$ iwrite , imm.-addr. store $\text{LS}[R(s)] = R(s); //$ write , indirect store
	$R(d) = \text{INDEX}; //$ ldix , load index of each lane $R(d) = \text{imm}; //$ vload , load immediate
predication	$\text{EXECUTE_IN_ALL}(...); //$ endwhere $\text{EXECUTE_WHERE_EQ}(...); //$ whereeq $\text{EXECUTE_WHERE_LT}(...); //$ wherelt $\text{EXECUTE_WHERE_CRY}(...); //$ wherecry
loop with counter	$\text{REPEAT}(\text{imm}); //$ setlc , $\text{imm} \in \{0..1023\}$ $\text{END_REPEAT}; //$ ijmpnzdec
	$\text{NOP}; //$ nop

Table 8: The Connex ISA described in Opincaa Connex, with C++ syntax. $R(d)$ is an arbitrary destination register, $R(s1)$ is the first source register for a binary operator ($d, s1, s2 \in \{0..31\}$); imm is the immediate constant operand with $\text{imm} \in \{-32768..32767\}$, unless otherwise specified.