LAVAL COMPUTER reference manual

Introduction

LAVAL COMPUTER is the first computer to implement the revolutionary LAVAL CPU architecture. Like every good acronym, LAVAL is recursively defined: Laval Advanced Vectorized Architecture Laboratory.

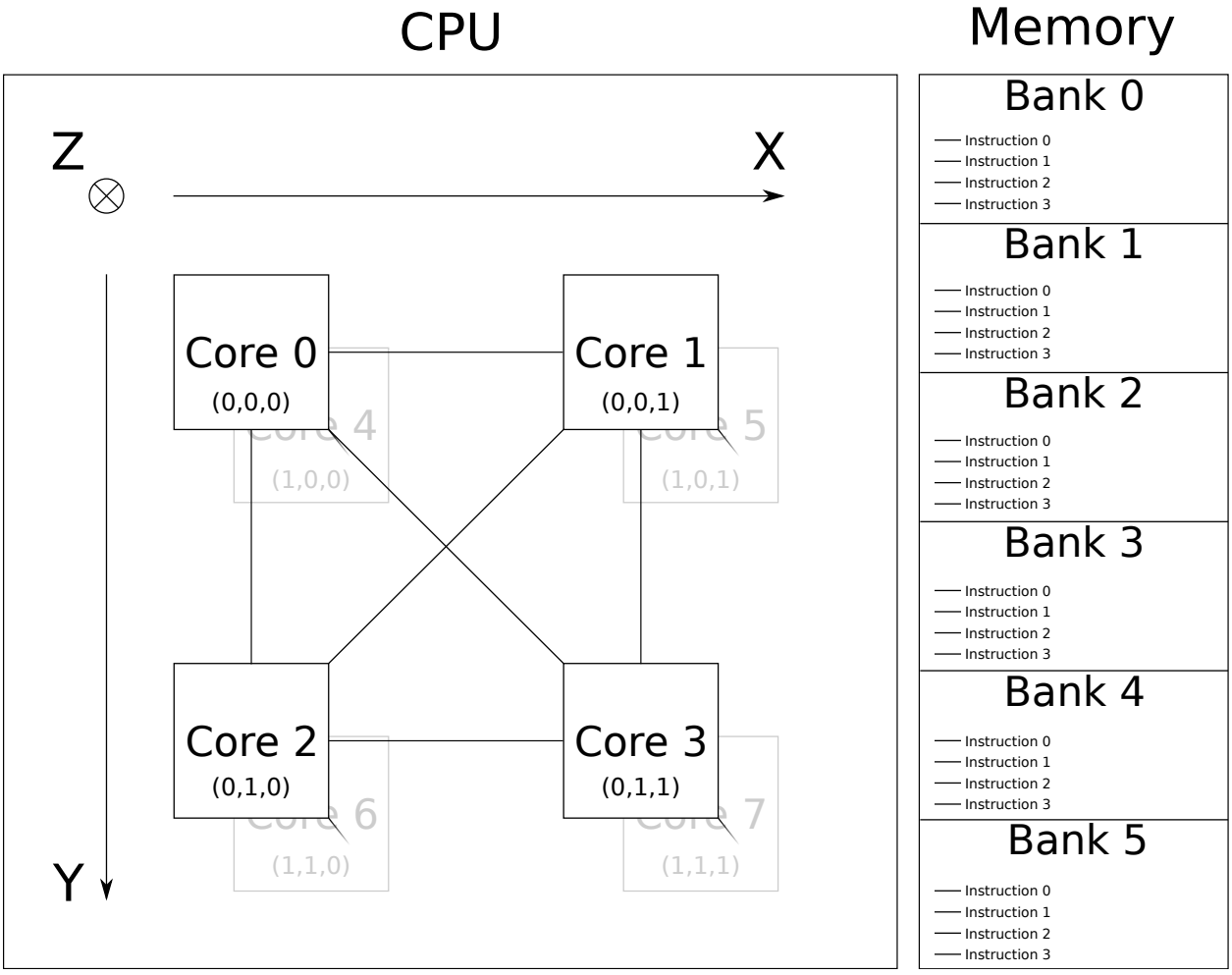
Its a novel architecture, sets to revolutionize the computing world with its massively parallel cores organisation.

Architecture

LAVAL architecture lies somewhere between those of CPU, GPU and FPGA. It consists of a large number of simple cores connected together in a local basis following a cube pattern. It is based on an Harvard architecture, cores execute instructions stored in their linked memory bank, which is read only. Branches are made by switching execution from one memory bank to another one.

Many CPU settings, like the number of cores, are implementation defined. Refer to your specific CPU model for more information.

For example, this figure presents a small 2x2x2 implementation with six memory banks of four bytes each.



Do note that only some of the core-to-core links are drawn to reduce clutter.

Core overview

The main component of the LAVAL architecture is the CORE unit. It is a very low complexity core capable of executing 8 bits instructions *arguments included*. Every instruction executes in a single clock cycle.

A CORE owns a single general purpose register name VAL and cannot address RAM, instead, the collective memory of all the cores is used as the program state. Therefore, as soon as an algorithm needs more than a single 8 bits variable, multiple cores are needed.

A consequence of these characteristics is that no instruction allows to work with constant larger than 4 bits (0xf). To work around that, either use two instructions or use another core to generate the needed values.

Refer to the instruction set section for more information.

Every core owns a multiplexer that allow the core to target one of its 26 neighbors. Inter-core communication is heavily optimized for constant patterns, since an instruction is needed to switch the multiplexer. Algorithms designed so that cores work on a minimal number of incoming values are therefore way more efficient.

Registers

Following is the description of every CPU register. Most operations on these registers are abstracted to the programmers by the ISA but understanding them is useful to better grasp the CPU inner working.

Every register is 8 bits long.

VAL

VAL is the single general purpose register of a CORE unit. Most instruction read or modify VAL.

Negative numbers are encoded using a two complement representation.

VAL is zero initialized.

MUX

MUX is the core multiplexer register.

Its value defines which core a core may access in a given cycle.

Its value is encoded as a base 3 number where the most significant trit addresses the Z dimension and the least significant trit addresses the X dimension.

A trit encode an offset relative to the current core, as followed:

- 0: BEFORE
- 1: CURRENT
- 2: AFTER

For example, the direction `BEFORE, CURRENT, AFTER` is encoded as `(0 * 3^2) + (1 * 3^1) + (2 * 3^0)`.

Refer to the Inputs/Outputs section for more details about the multiplexer usage.

Its default value is `CURRENT, CURRENT, CURRENT`, which is an illegal direction to fetch from.

PC

PC is the program counter.

Its value defines which instruction of the current memory bank will be executed next. PC is incremented after every instruction who does not stall the core's pipeline. The pipeline is stalled when an instruction fetching a value from the multiplexer fails to do so. This happen when the pointed core have not yet issued a synchronization instruction (SYN).

As with any register incrementing, its value may overflow. This means its value automatically reset to zero instead of getting to 0x100 (256).

PC is zero initialized.

MEMBANK

MEMBANK is the memory pointer register.

Its value defines from which memory bank the next instruction will be fetched. Its value is set at core initialization according to the `core_to_mem` setting. Its value may also be changed at runtime using one of the jump instructions.

Statuses

Each core also includes one or more status registers. As of today, they are not exposed to the end user and their exact layouts are implementation defined. A future ISA revision may standardize their layouts.

Inter-core transfers

Each core owns a multiplexer which may be pointed toward one of the its 26 neighbours. A core may then use its multiplexer to **load** a value from the target core.

Warning: You may not connect a core to itself.

Such transfers are synchronized, the *loading* instruction will not proceed and none of the CPU registers will get affected as long as the target core does not emit a *synchronization* instruction. The same is true for synchronization instructions, a core will block on such instruction until at least one core as fetched a value from it.

A core executing a synchronization instruction make its registers available to all currently connected cores. This means many cores may load a value from a common core and a single synchronization is needed.

When a core has been fetched, is synchronization flag is reset and another synchronization instruction will be needed for the next fetch.

Finally, a core may not fetch from an non-existent core, except if the fetching core is configured as a CPU input (more on that later).

Examples in pseudo-code:

```
Core0:
    VAL <- 5
    Synchronize  ; Proceed immediatly since at least one core is fetching.

Core1:
    Connect to core 0
    Load from connected core
    ; Load immediatly since the target core is executing a synchronization instruction.
    ; VAL is now 5
```

```
Core2:
    Connect to core 0
    Load from connected core
    ; Load immediatly since the target core is executing a synchronization instruction.
    ; VAL is now 5

Core0:
    VAL <- 5
    Do nothing
    Synchronize  ; Proceed immediatly since at least one core is fetching.

Core1:
    Connect to core 0
    Load from connected core
    ; Wait for one cycle, since the target has not yet synchronized, then load.
    ; VAL is now 5

Core2:
    Connect to core 0
    Load from connected core
    ; Wait for one cycle, since the target has not yet synchronized, then load.
    ; VAL is now 5

Core0:
    VAL <- 5
    Synchronize  ; Proceed immediatly since at least one core is fetching.

Core1:
    Connect to core 0
    Load from connected core
    ; Load immediatly since the target core is executing a synchronization instruction.
    ; VAL is now 5

Core2:
    Connect to core 0
    Do nothing
    Load from connected core
    ; The target core has reset is synchronzation flag. Wait indefinitely.

Core0:
    VAL <- 5
    Synchronize  ; Wait for one cycle since no one is fetching from this core in this cycle.

Core1:
    Connect to core 0
    Do nothing
    Load from connected core
    ; Load immediatly since the target core is executing a synchronization instruction.
    ; VAL is now 5

Core2:
    Connect to core 0
    Do nothing
    Load from connected core
```

```
; Load immediatly since the target core is executing a synchronization instruction.
; VAL is now 5
```

Inputs/Outputs

As described up to now, a CPU may only works on some predefined constants, which is pretty useless by itself. The get useful, a CPU must communicates with the real world.

Inputs

A core may be connected to a CPU input. Doing so enables that specific core to receive data from the outside.

Refer to the assembler section for more information on how to configure inputs.

Apart from the needed configuration, receiving data from the outside or from another core works in exactly the same way.

To use its input a core must points its multiplexer to a normally invalid direction, but not itself. This means, by design, that only cores on the edge of the CPU may have an input.

It is forbidden to wire two inputs or both inputs and outputs to the same core.

Outputs

A core may be connected to a CPU output. Doing so enables that specific core to send data to the outside.

Refer to the assembler section for more information on how to configure outputs.

Apart from that needed configuration, sending data to the outside or to another core works in exactly the same way.

To use its output a core must simply synchronize (SYN) its current VAL. The CPU output will then automatically fetch the value and unlock the core. The result is the same as if another core loaded from it in the same clock cycle.

By design, outputs are limited to cores on the edge of the CPU.

It is forbidden to wire both inputs and outputs to the same core.

Assembler

This section describe how to use the assembler to write a program targeting the LAVAL ISA.

The assembler does not care about whitespaces as long as the required whitespace are presents (for example between an instruction and its arguments) and the tokens are undivided (no space is allowed in the middle of an instruction mnemonic).

Settings

Assembler programs start with a setting section. Settings are used to list the requirement of a cpu running this program. Settings are written, one per line, using the following format:

`\.(\w+) ([\d, ]*)`

- First capture group is the setting name
- Second capture group contains the setting's argument(s)

Settings most not appear after the first memory bank.

.cores

CPU dimensions

Argument	Size	Description
0	0..65535	Number of cores on Z dimension
1	0..65535	Number of cores on Y dimension
2	0..65535	Number of cores on X dimension

.mem_number

Number of memory banks

Argument	Size	Description
0	0..255	Number of memory banks

.mem_size

Memory banks size

Argument	Size	Description
0	0..255	Size in bytes of each memory bank

.core_to_mem

Assign memory banks to cores at CPU initialization

Argument	Size	Description
0	0..255	Memory bank assigned at boot to core 0
1	0..255	Memory bank assigned at boot to core 1
n	0..255	Memory bank assigned at boot to core n

`n` is the total number of cores.

.in

Inputs assignment

Argument	Size	Description
0	0..255	Core to which input #0 is connected
1	0..255	Core to which input #1 is connected
n	0..255	Core to which input #n is connected

`n` is the number of inputs and must be smaller or equal to 65535.

.out

Ouputs assignment

Argument	Size	Description
0	0..255	Core to which output #0 is connected
1	0..255	Core to which output #1 is connected
n	0..255	Core to which output #n is connected

`n` is the number of outputs and must be smaller or equal to 65535.

Memory bank

Memory banks are programmed by declaring a block of instructions starting with the following indicator:

`(\d+):`

- Capture group indicate the ID of the memory bank that will contains the following instructions.

Do note that memory bank IDs are 0-indexed.

Instructions

Instruction respect the following format:

`(\w{3})( -?\d+(?:, ?\d+)*)?`

- First capture group is the instruction mnemonic
- Second (optional) capture group contains the instruction argument(s)

Please note that the specific number and length of the arguments depend on the instruction. Refer to their documentation for more information.

Instructions *must* be part of a memory bank.

Comments

A line comment may be inserted using the `;` character. Everything following that character on the same line will get ignored by the assembler.

Preprocessor

The assembler also mandates the use of a preprocessor. It is used to provide some constants to the programmer:

Constant	Value
BEFORE	0
CURRENT	1
AFTER	2

Constant are simply textually replaced by their value before the file is passed to the assembler.

Examples

```
; Repeatly move value from a single input to a single output. Speed optimized.

.cores 1, 1, 1
.mem_number 2
.mem_size 2
.core_to_mem 0
.in 0
.out 0

0:
    MUX CURRENT, BEFORE, CURRENT
    JMP 1

1:
    SYN
    JMP 1
```

The above program is optimized for speed. It takes exactly two clock cycles to move a value from its input to its output. In a resource constrained system, it may be appropriate to sacrifice some speed for cost. The above program program may be modified for this objective like so:

```
; Repeatly move value from a single input to a single output. Cost optimized.

.cores 1, 1, 1
.mem_number 1
.mem_size 3
.core_to_mem 0
.in 0
.out 0

0:
    MUX CURRENT, BEFORE, CURRENT
    SYN
    JMP 0
```

Now, the program is smaller overall (1 membank x 3 bytes vs 2 membank x 2 bytes) but takes 3 clock cycles to move a value. Many programs may be optimized one way or another using similar tricks.

Instruction set

Basic instructions

NOP

No operation

Argument: None

Description:

Do nothing for one cycle

SYN

Sync

Argument: None

Description:

Sync VAL with connected mux(es). Multiple cores may receive the same synced value as long as they fetch it on the same cycle. This instruction will block until at least one core has fetched a value.

DBG

Output to debugger

Argument: None

Description:

Output core status—i.e., the values of all the registers, to the connected debugger.

HLT

Halt

Argument: None

Description:

Stop CPU execution and return VAL. Two cores returning at the same time return an undefined VAL.

HCF

Argument: None

Description:

???

Controlling mux

MUX

Set multiplexer to another core

Arguments:

Size	Description
0..2	Offset on dimension 0
0..2	Offset on dimension 1
0..2	Offset on dimension 2

Description:

Point mux to another core by setting the MUX register as indicated by instruction's arguments.

Notes:

A core may not be connected to itself.

Example:

```
MUX CURRENT, BEFORE, AFTER
```

CTC

Connect to carry

Argument: None

Description:

Connect the core multiplexer to its target carry bit.

Notes:

The multiplexer may also be connected to the VAL register using CTV.

CTV

Connect to VAL

Argument: None

Description:

Connect the core multiplexer to its target VAL register. This is the default configuration.

Notes:

The multiplexer may also be connected to the carry bit using CTC.

Reading from mux

MXD

Multiplexer discard

Argument: None

Description:

Fetch and discard a value from the mux. Use this instruction to unlock a core blocked on a SYN instruction.

Notes:

This instruction keeps VAL unaffected.

Example:

```
.cores 1, 1, 2
.mem_number 2
.mem_size 3
.core_to_mem 0, 1

0:
    LCL 1
    SYN     ; Will waits here for one cycle, then executes on the next
    LCL 2   ; Will executes on the fourth cycle

1:
    NOP
    NOP
    MXD     ; VAL is still zero
```

MXL

Multiplexer load

Argument: None

Description:

Fetch and load into VAL the value from mux.

MXA

Multiplexer addition

Argument: None

Description:

Fetch and add the value from the mux to VAL. MXA always performs a signed addition—i.e., VAL is considered negative if the core's sign bit is true. MXA set the sign bit if the operation results in a negative number.

MXS

Multiplexer subtraction

Argument: None

Description:

Fetch and subtract the value from the mux to VAL. MXS always performs a signed subtraction—i.e., VAL is considered negative if the core's sign bit is true. MXS set the sign bit if the operation results in a negative number.

Jumping

JMP

Jump unconditionally

Argument:

Size	Description
0..15	Membank id

Description:

Point current core to a new membank as indicated by the argument. PC is reset to 0.

Example:

```
.cores 1, 1, 1
.mem_number 2
.mem_size 2
.core_to_mem 0, 1

0:
    NOP     ; First cycle.
    JMP 1   ; Second cycle.
```

```
1:
    NOP     ; Third cycle.
```

JLZ

Jump if less than zero

Argument:

Size	Description
0..15	Membank id

Description:

Point current core to a new membank as indicated by the argument if VAL if less than 0. PC is reset to 0.

JEZ

Jump if equal to zero

Argument:

Size	Description
0..15	Membank id

Description:

Point current core to a new membank as indicated by the argument if VAL if equal to 0. PC is reset to 0.

JGZ

Jump if greater than zero

Argument:

Size	Description
0..15	Membank id

Description:

Point current core to a new membank as indicated by the argument if VAL if greater than 0. PC is reset to 0.

Using constants

LCL

Load constant into low part

Argument: None

Size	Description
0..15	Value to load

Description:

Load a 4 bits constant into the 4 lower bits of VAL. The four higher bits are unaffected.

Notes:

To load the higher bits, use LCH.

LCH

Load constant into high part

Argument: None

Size	Description
0..15	Value to load

Description:

Load a 4 bits constant into the 4 higher bits of VAL. The four lower bits are unaffected.

Notes:

To load the lower bits, use LCL.

LSL

Logical shift, left

Argument: None

Size	Description
0..15	Value to shift by

Description:

Logically left shift VAL by number of bits indicated by the argument. Every bit of VAL is moved a given number of bit positions. The vacant bit-positions are filled with zeros.

Note:

The sign bit is left untouched. This means that you may use this instruction to multiply negative number. For example: `-1 << 1 = -2`.

LSR

Logical shift, right

Argument: None

Size	Description
0..15	Value to shift by

Description:

Logically right shift VAL by number of bits indicated by the argument. Every bit of VAL is moved a given number of bit positions. The vacant bit-positions are filled with zeros.

Note:

The sign bit is left untouched. This means you must be extremely careful to use this instruction with negative numbers since its behaviour is probably not what you want. Example: `-65 >> 1 = 95`

CAD

Constant addition

Argument: None

Size	Description
0..15	Value to add

Description:

Add a constant to VAL. CAD always performs a signed addition—i.e., VAL is considered negative if the core's sign bit is true. CAD set the sign bit if the operation results in a negative number.

CSU

Constant subtraction

Argument: None

Size	Description
0..15	Value to subtract

Description:

Subtract a constant to VAL. CSU always performs a signed subtraction—i.e., VAL is considered negative if the core's sign bit is true. CSU set the sign bit if the operation results in a negative number.

CAN

Constant AND

Argument: None

Size	Description
0..15	Mask

Description:

Apply a logical AND to the four lower bits of VAL. The 4 higher bits are cleared.

COR

Constant OR

Argument: None

Size	Description
0..15	Mask

Description:

Apply a logical OR to the four lower bits of VAL. The 4 higher bits are unaffected.

The COR instruction is unable by itself to affect all the bits of VAL. It may be useful to then use the CAN instruction to restrict VAL to affected values.

Simulator

For simulator usage, refer to its included help: `simulator --help`.