CPUNAME Instruction Set Architecture (ISA) Specification

luccie-cmd

March 22, 2025

Contents

1 General overview

2 Registers

Register	Purpose
R1-R32	General purpose registers
SP	Pointer to where the stack head currently is
IP	Pointer to where the CPU will execute instructions
PTP	Pointer to where the MMU will translate addresses
FLAGS	Status flags used for conditional jumps and moves
EADDR	A temporary effective address that the CPU uses to calculate some addresses for instructions
TEMP	A temporary register that the CPU uses for instructions that output nothing (Like COMP and TEST)
CLR	The CPUNAME's internal control register used for defining certain exectecution behaviors

Table 1: Register overview

See table above for register descriptions.
Both SP and IP are virtual addresses meaning that they should be mapped in the PTP to be used.
EADDR and TEMP registers are inaccessible for both the privileged and unprivileged.
Flags, CLR and the PTP registers can only be set directly via the privileged program, otherwise they can only be affected by the ALU.
The CLR register looks something like this in its bits:
Bits 0-3: Paging Level.
Bits 4-127: Reserved.
The FLAGS looks like this in its bits:
Bits 0-127: Reserved.

2.1 NOTES

The flags register affects conditional execution such as jumps and moves and can only be effected by the ALU

3 Instruction encoding

3.1 General Instruction Format

Unlike RISC, the CPUNAME will have variable length instructions, meaning that every instruction only takes up the binary space it needs.
A typical instruction will generaly look like so:

Field	**Size (In bytes)**	**Description**
Prefix	1	The instruction prefix, will change what addressing mode(s) to use. Described in Instruction Set subsection Prefixes
Opcode	1	The main instruction identifier. Contains number of sources and if the destination or source is a memory or register or immediate
Operands(s) (R8/16/32/64/128)	1	A register to use in the instruction
Operands(s) (M8/16/32/64/128)	1, 2, 4, 8 or 16	A memory location to use in the instruction. Number of bytes pulled from memory will depend on the instruction's prefix otherwise default 128 bit addressing is assumed
Operands(s) (I8/16/32/64/128)	1, 2, 4, 8 or 16	An immediate value to use in the instruction. Number of bytes will depend on the instruction's prefix otherwise default 128 bit addressing is assumed

Table 2: Instruction overview

Prefixes and operands are optional, some instructions (Described in the instruction set table where the operands are None) don't use operands. An instruction that has a prefix byte but doesn't take any operands will trigger an Invalid Prefix Opcode interrupt (#IPO). The number of operands for a instruction if specified in the table in Instruction Set, please refer to that for specific instruction encoding and arguments.

4 Instruction set

4.1 Instruction set as mnemonics

4.1.1 Moves

Bytes	**Mnemonic**	**Operand 1**	**Operand 2**	**Description**
00	MOVE	R8/16/32/64/128	R8/16/32/64/128	Move value at source register (Operand 2) to destination register (Operand 1)
01	MOVE	R8/16/32/64/128	M8/16/32/64/128	Move value located at the memory address in (Operand 2) to destination register (Operand 1)
02	MOVE	R8/16/32/64/128	R8/16/32/64/128+ I8/16/32/64/128	Move value at the address in the register of Operand 2, plus the immediate value of Operand 2 into the destination register (Operand 1)
03	MOVE	R8/16/32/64/128	R8/16/32/64/128+ R8/16/32/64/128	Move value at the address in the register of Operand 2, plus the value of the register of Operand 2 into the destination register (Operand 1)
04	MOVE	M8/16/32/64/128	R8/16/32/64/128	Move the value of register Operand 2 into the memory location specified in Operand 1

4.1.2 Stack Manipulation

Bytes	**Mnemonic**	**Operand 1**	**Operand 2**	**Description**
05	PUSH	R8/16/32/64/128	None	Push value in operand 1 on the stack and decrement stack pointer by N bytes (specified using prefix)
06	POP	R8/16/32/64/128	None	Pop value at current stack pointer into Operand 1 and increment stack pointer by N bytes (specified using prefix)

4.1.3 Control flow

Bytes	Mnemonic	Operand 1	Operand 2	Description
07	JIE	I128	None	Perform a conditional jump to the immediate value of Operand 1 if zero flag is set
08	JIL	I128	None	Perform a jump to the immediate value of Operand 1 if SF != OF
09	JMP	I128	None	Perform an unconditional jump to the immediate value of Operand 1
0A	JSR	I128	None	Perform a call (Jump to subroutine) to the immediate value of Operand 1, push the value of the return address (Current IP+17) first
0B	RFS	None	None	Perform a return (Return from subroutine) from a procedure by popping the return value from the stack
0C	U2K	None	None	Performs a call to the elevated program to handle privileged instructions, changes the privilege level

4.1.4 Arithmetic

Bytes	Mnemonic	Operand 1	Operand 2	Description
0D	TEST	R8/16/32/64/128	R8/16/32/64/128	Perform a bitwise AND but don't store the result only update the FLAGS
0E	XOR	R8/16/32/64/128	R8/16/32/64/128	Perform a bitwise XOR On the operands and store the result in the first operand
0F	SHL	R8/16/32/64/128	I8/16/32/64/128	Perform a shift to the left by the number of times specified in Operand 2 on Operand 1

4.1.5 Miscellaneous

Bytes	Mnemonic	Operand 1	Operand 2	Description
10	COMP	R8/16/32/64/128	I8/16/32/64/128	Compare the value in register Operand 1 against the immediate value and set flags

4.1.6 Prefixes

Certain prefixes can be placed before an instruction to modify its behavior. These prefixes can affect the selection of registers, immediate values, memory addressing modes, or other instruction properties. By default, without any prefixes, all operands are 128 bits. The table below defines the available prefixes and their effects.

Bytes	**Effect**
11	Changes instruction to operate using 8 bit values for all operands
12	Changes instruction to operate using 16 bit values for all operands
13	Changes instruction to operate using 32 bit values for all operands
14	Changes instruction to operate using 64 bit values for all operands

4.2 Microinstructions

4.2.1 Loading

Mnemonic	**Arguments**	**Description**
LDR	Register index	Load contents of the register specified in the register index onto the bus
LDI	Value	Load the provided value onto the bus

4.2.2 Writing

Mnemonic	**Arguments**	**Description**
STR	Register index	Store the contents of the bus into the register specified in register index

4.2.3 ALU

Mnemonic	**Arguments**	**Description**
PAA	None	Prepare ALU for an addition
LALU	Operand	Load the Nth ALU operand (with N being specified in the argument) from the bus
AE	None	Allow the ALU to output its result onto the bus

4.2.4 MMU

Mnemonic	**Arguments**	**Description**
MR	None	Load value at the address specified in the EADDR register to the TEMP register
MW	None	Write the value of the current bus to the address specified in the EADDR register

5 Privilige levels

The CPUNAME and versions thereof have 2 privilege levels named unprivileged and privileged. These privilege levels can be compared to X86's ring 0 and ring 3 respectively. A program running in privileged mode can execute any and all instructions and is allowed to change things like the PTP register, SP and IP via direct assignment. However an unprivileged program running is not allowed to access some instructions and registers that a privileged program can access. For example the PTP register inaccessible for a unprivileged program and will throw an the Unpriviliged Execute interrupt (#UPE). A full description of instructions and registers the unprivileged level isn't allowed to access will be provided in a list below.

5.1 Instructions

Currently there are no instructions that can only be run on a privileged level.

5.2 Registers

- PTP
- SP
- IP
- FLAGS
- EADDR
- TEMP

6 Memory

The CPUNAME can support up to 13 levels of paging, each taking up 9 bits of the virtual address with the exception of the last level which points to the physical address, this takes up 12 bits giving a total of 117 bits used for virtual addressing. The remaining 11 bits should all be the same, if for any reason these bits aren't the same when it is needed (Wether for reading or writing) it will trigger a Non Canonical interrupt (#NCC). The 12 bits are the default value and are extended by 9 bits for every paging level activated. The maximum

paging level useable is 13 by default but can be changed by setting a specific value in the control register (CLR) when the maximum paging level of 0 is applied, the MMU treats this as not having any paging enabled. For every new level of paging addded, 9 bits of virtual address space get added which would equate to a function like: $2^{N\times9}$ where N is the number of paging levels enabled. The stack and instruction pointer (SP and IP respectively) are to be mapped in the PTP (If paging is enabled).
PTP describes a physical address stored somewhere in the RAM, the PTP is used for translating virtual addresses to physical ones via a table described below.

Bits	**Name**	**Description**
1	RED	Describes if a page can be read from, trying to read from a non readable page will trigger an Un Readable Page interrupt (#URP).
1	PRS	Describes if a page is present, trying to use a non present page will trigger a Non Present Page interrupt (#NPP).
1	WRT	Describes if a page can be written to, trying to write to a non writeable page will trigger an Un Writeable Page interrupt (#UWP)
1	PVP	Describes if a page is occupied by the privileged program, trying to use a privileged page as an unprivileged program will trigger an Un Privilige Page interrupt (#UPP)
1	NEX	Describes if a page can *NOT* be executed, tying to execute from a non executable page results in a Non Executable Page interrupt (#NEP)
40	NLT	This describes the address of the next level of paging table, if this is the last level it contains the physical page number. Must be non null otherwise it will throw a Null Next Level interrupt (#NNL)
19	RSV	These bits are reserved for future purposes and should be zero at all times. Attempting to set these bits to any other value will result in a Non Zero Reserved interrupt (#NZR)

7 MMU

The CPUNAME's MMU works by using the PTP to translate addresses given to it via the EADDR register, and via the CPU's control register you can change the number of page tables it can use.
If the page tables specified in the control register is 0 it will directly operate on it without doing any passes nor checks. Any value higher or equal to one(1) will translate trough that many page tables
A maximum of 13 page tables can be used (as addressed previously in the memory section) and every page table takes up 9 bits of the virtual address with exception of the last one which takes up 12 bits

Any unused bits should be sign extended (AKA canonical), if they are not the CPU throws a Non Canonical interrupt (#NCC). A full 13 table page layout will look something like this:

Bits	**Description**
0-11	Used for the physical page number indexing which page of RAM to use
12-20	Used for the L1 page table index in the L2 page table
21-29	Used for the L2 page table index in the L3 page table
30-38	Used for the L3 page table index in the L4 page table
39-47	Used for the L4 page table index in the L5 page table
48-56	Used for the L5 page table index in the L6 page table
57-65	Used for the L6 page table index in the L7 page table
66-74	Used for the L7 page table index in the L8 page table
75-83	Used for the L8 page table index in the L9 page table
84-92	Used for the L9 page table index in the L10 page table
93-101	Used for the L10 page table index in the L11 page table
102-110	Used for the L11 page table index in the L12 page table
111-119	Used for the L12 page table index in the L13 (PTP) page table
120-127	Reserved Should be canonically sign extended

8 ALU

The CPUNAME's ALU will work by using SISD and not SIMD, this is because 128 bits is already big enough that most SIMD instructions would be useless. The ALU works by having 2 internal buffers where 2(two) 128 bit values get stored. Whenever the ALU is prepared for anything it will wait for the AE microinstruction and then output its contents to the bus where another microinstruction can then use the value.

9 I/O

The CPUNAME only support MMIO as the PIO isn't used anymore. The MMIO works in the CPU by having some virtual address mapped to the physical

address of the MMIO region and marked as Priviliged, Readable, Writeable, Present and Non Executable. After it has done this the privileged program is free to read and write from it. It should also be noted that the hardware devices read and write from this region using the MMU.
Essentially a MMIO region is just a normal region in the ram.

10 Interrupts

TODO, currently not implemented.

11 Pipeline

TODO, currently not implemented.