

What is CESCO / CESC?:

CESCA (Carefully Enhanced Simple Computer Architecture) is the ISA/ABI used by the 8 bit computer I made, called CESC (Carefully Enhanced Simple Computer). The origin of those names is long and boring, but it's enough for you to know they're catalan names.

CESCA Instruction Set Architecture (ISA):

FEATURES:

4 general purpose registers: R0, R1, R2, R3

- All of them are connected to the ALU, so it can use any of of them as first or second operand.

Special purpose registers (all 8 bit): **PC** (Program Counter), **SP** (Stack Pointer), **MAR** (Memory Address Register), **IR** (Instruction Register) and an **Output Register**.

- Instructions take 2 bytes, but the IR only stores the opcode. The arguments, after being fetched, are stored directly to where they are needed.
- My computer also uses some temporary registers in the ALU. Those aren't included here as they aren't a part of the ISA specification, but of my particular implementation (other implementations may not use them).
- The Output Register drives the decimal display. The LCD display has its own controller and its memory, so no extra registers are required.

4 flags: Zero, Carry, oVerflow and Sign flags indicate if the last ALU operation resulted in a 0, an unsigned overflow (carry / borrow), a signed overflow or a negative number, respectively.

1 KB (4x256) of addressable memory space: control logic chooses between 4 banks of memory and the MAR holds the 8 bit address (for a total of 256 bytes each):

- 256 bytes: Program memory high (opcodes)
- 256 bytes: Program memory low (arguments)
- 256 bytes: Data memory
- 256 bytes: Stack

Note that this has the following implications:

- A stack overflow won't corrupt program or data memory
- All instructions must take 2 bytes (even if they don't need an argument), since the opcode and the argument have the same address but they aren't mixed in the same bank.
- Fetch cycles are faster, since the PC only has to be fetched once.

8 bit decimal display and LCD panel for outputing 8 bit numbers through the decimal display, or any sequence of characters through the LCD panel.

INSTRUCTION FORMATS:

Register: 0000DDAA FFFFXXBB
Immediate: 0000DDAA IIIIIIII ("I" can also be an address "@")
Reduced: 0000DDAA XXXXXXXX

O: Opcode	D: Rd (Destination) or extended opcode	
A: Ra (1st operand)	B: Rb (2nd operand)	I: Immediate value
@: Immediate address	F: ALU Function	X: Don't care

INSTRUCTIONS:

	Mnemonics	Machine code	
Arithmetic / logic instructions:	[ALU OPERATIONS] Rd, Ra, Rb	0000DDAA	FFFFXXBB
	ADDI Rd, Ra, Imm8	0001DDAA	IIIIIIIII
	ANDI Ra, Imm8	0010DDAA	IIIIIIIII
	[CMP OPERATIONS] Ra, Rb	001100AA	FFFFXXBB
	CMP-SUBI Ra, Imm8	001101AA	IIIIIIIII
	CMP-ANDI Ra, Imm8	001110AA	IIIIIIIII
	CLF	001111XX	01XXXXXX
Data movement:	LI Rd, Imm8	0100DDXX	IIIIIIIII
	ST-Addr Ra, Addr8	01010XAA	@@@@@@@@
	ST-Reg Ra, Rb	01011XAA	XXXXXXBB
	LD-Addr Rd, Addr8	0110DDXX	@@@@@@@@
	LD-Reg Rd, Ra	0111DDAA	XXXXXXXX
	PUSH Ra	1000XXAA	XXXXXXXX
	POP Rd	1001DDXX	XXXXXXXX
Jump instructions:	J Addr8	101000XX	@@@@@@@@
	JR Ra	101001AA	XXXXXXXX
	JAL Addr8	101010XX	@@@@@@@@
	RET	101011XX	XXXXXXXX
	JZ Addr8	101100XX	@@@@@@@@
	JNZ Addr8	101101XX	@@@@@@@@
	JC Addr8	101110XX	@@@@@@@@
	JNC Addr8	101111XX	@@@@@@@@
	JV Addr8	110000XX	@@@@@@@@
	JNV Addr8	110001XX	@@@@@@@@
	JN Addr8	110010XX	@@@@@@@@
	JP Addr8	110011XX	@@@@@@@@
	JSP Addr8	110100XX	@@@@@@@@
	JLEU Addr8	110101XX	@@@@@@@@
	JLT Addr8	110110XX	@@@@@@@@
	JLE Addr8	110111XX	@@@@@@@@
Output and misc:	LCD-Com Imm8	111000XX	IIIIIIIII
	LCD-Imm Imm8	111001XX	IIIIIIIII
	LCD-Reg Ra	111010AA	XXXXXXXX
	LCD-Mem Addr8	111011XX	@@@@@@@@
	DEC-Reg Ra	111100AA	XXXXXXXX
	DEC-Mem Addr8	111101XX	@@@@@@@@
	HLT	111110XX	XXXXXXXX
	NOP	111111XX	XXXXXXXX

ALU/CMP Operations:

Funct	Mnemonic	Description / observations
0000	MOVE Rd, Ra	Move the contents of Ra into Rd (won't trigger carry or overflow flags).
0001	ADD Rd, Ra, Rb	Adds the contents of Ra and Rb.
0010	SUB Rd, Ra, Rb	Subtracts the contents of Ra and Rb (Ra - Rb).
0011	ADDC Rd, Ra, Rb	Add with Carry: Adds Ra and Rb (plus the carry flag).
0100	SUBB Rd, Ra, Rb	Subtract with Borrow: Subtracts Ra and Rb (minus the carry flag).
0101	AND Rd, Ra, Rb	Performs a bitwise logic AND between Ra and Rb.
0110	OR Rd, Ra, Rb	Performs a bitwise logic OR between Ra and Rb.
0111	NOT Rd, Ra	Performs a bitwise logic NOT to Ra.
1000	XOR Rd, Ra, Rb	Performs a bitwise logic XOR between Ra and Rb.
1001	NAND Rd, Ra, Rb	Performs a bitwise logic NAND between Ra and Rb.
1010	NOR Rd, Ra, Rb	Performs a bitwise logic NOR between Ra and Rb.
1011	XNOR Rd, Ra, Rb	Performs a bitwise logic XNOR between Ra and Rb.
1100	SLL Rd, Ra	Shift Left Logical: Ra gets shifted left 1 position (corresponds to A+A).
1101	SRL Rd, Ra	Shift Right Logical: Ra gets shifted right 1 position (and filled with a 0).
1110	SRA Rd, Ra	Shift Right Arithmetic: Ra gets shifted right (and the sign is extended).
1111	ROL Rd, Ra	Rotate Left: Performs a circular shift (SLL and add the carry to the end).

REMARKS:

- After a subtraction, the carry flag indicates the borrow (it's only active on an unsigned overflow).
- The prefix "CMP-" in front of any of those mnemonics indicates it's a CMP instruction.
- The mnemonic "CMP" (without any ALU function) must be interpreted by the assembler as "CMP-SUB", since comparing integers is the most common use case of this instruction.
- The immediate instructions ADDI, ANDI, CMP-SUBI and CMP-ANDI work the same way as their non-immediate counterparts, but instead of Rb an immediate value is used.
- After a shift / rotation, the state of the overflow flag is undefined. The carry flag is set when an unsigned overflow occurs on SLL / ROL, and it's never set in SRL / SRA. Zero and Sign flags continue to work as expected.
- The operations MOVE, NOT, SLL, SRL, SRA and ROL ignore the value stored in Rb.


INSTRUCTION DETAILS:

ALU Operations:

[ALU_OP] Rd, Ra, Rb	0000DDAA FFFFXXBB	Rd = ALU(Ra, Rb) 
---------------------	-------------------	--


Performs the ALU operation indicated by the 4 Funct bits, using the contents of Ra and Rb as operands. The result of the operation is stored in Rd and the flags are updated accordingly. See table above for ALU operations, mnemonics and descriptions.

ADD Immediate:

ADDI Rd, Ra, Imm8	0001DDAA IIIIIIII	Rd = Ra + Imm8 
-------------------	-------------------	--

Adds an immediate value to Ra and stores the result in Rd. The flags are updated accordingly.

AND Immediate:

ANDI Rd, Ra, Rb	0010DDAA IIIIIIII	Rd = Ra & Rb 
-----------------	-------------------	--


Performs a bitwise logic AND between Ra and an immediate value, and stores the result in Rd. The flags are updated accordingly.

Compare Operations:

CMP-[ALU_OP] Ra, Rb	001100AA FFFFXXBB	ALU(Ra, Rb) 
---------------------	-------------------	---

This instruction is identical to an ALU operation (it can perform the same Funct operations), but it doesn't write the results to any register (therefore Rd isn't needed). This is useful for setting the flags without messing up the stored contents. CMP-SUB is used to compare 2 integers and CMP-AND allows using masks. See table above for ALU operations, mnemonics and descriptions.

Compare-SUB Immediate:

CMP-SUBI Ra, Imm8	001101AA IIIIIIII	Ra - Imm8 
-------------------	-------------------	---

Subtracts an immediate value to Ra without storing the result anywhere. The flags are updated accordingly.

Compare-AND Immediate:

CMP-ANDI Ra, Imm8	001110AA IIIIIIII	Ra & Imm8 P
-------------------	-------------------	----------------------

Performs a bitwise logic AND between Ra and an immediate value without storing the result anywhere. The flags are updated accordingly.

Clear Flags:

CLF	001111XX 01XXXXXX	Flags P = 0b0000
-----	-------------------	---------------------------

Clears the flags (all 4 flags are set to 0).

Load Immediate:

LI Rd, Imm8	0100DDXX IIIIIIII	Rd = Imm8
-------------	-------------------	-----------

Puts an immediate value into Rd.

Store to Address:

ST-Addr Ra, Addr8	01010XAA @@@@@@@@	RAM[Addr8] = Ra
-------------------	-------------------	-----------------

Stores the contents of Ra to memory, using an immediate address.

Store to address in Register:

ST-Reg Ra, Rb	01011XAA @@@@@@@@	RAM[Rb] = Ra
---------------	-------------------	--------------

Stores the contents of Ra to memory, using the address stored in Rb.

Load from Address:

LD-Addr Rd, Addr8	0110DDXX @@@@@@@@	Rd = RAM[Addr8]
-------------------	-------------------	-----------------

Loads into Rd the memory contents from an immediate address.

Load from address in Register:

LD-Reg Rd, Ra	0111DDAA XXXXXXXX	Rd = RAM[Ra]
---------------	-------------------	--------------

Loads into Rd the memory contents from the address stored in Ra.

Push to the stack:

PUSH Ra	1000XXAA XXXXXXXX	Stack Ra
---------	-------------------	----------

Pushes the contents of Ra to the stack. The stack pointer starts at 0xFF and grows upwards. The starting position and direction are arbitrary (it can start at any position and grow in any direction) since the stack has its own memory bank, but those choices mimic the stack of real processor architectures, as well as a physical stack.

Pop from the stack:

POP Rd	1001DDXX XXXXXXXX	Rd Stack
--------	-------------------	----------

Pops the top of the stack and stores it in Rd. This should only be done if PUSH has been used before.

Jump:

J Addr8	101000XX @@@@	PC = Addr8
---------	---------------	------------

Jumps unconditionally to an immediate address.

Jump to Register:

JR Ra	101001AA XXXXXXXX	PC = Ra
-------	-------------------	---------

Jumps unconditionally to the address stored in a register.

Jump And Link:

JAL Addr8	101010XX @@@@	Stack PC+1; PC = Ra
-----------	---------------	---------------------

Calls a subroutine: pushes the address of the next instruction to the stack before jumping unconditionally.

Return from subroutine:

RET	101011XX XXXXXXXX	PC Stack
-----	-------------------	-------------

Pops the top of the stack and jumps unconditionally to that address. Make sure you have used POP as many times as PUSH to ensure the return address is at the top of the stack.

Jump on Zero:

JZ Addr8	101100XX @@@@	if(Z) PC = Addr8
----------	---------------	------------------

Jumps to an immediate address if the zero flag is set (the result of the last ALU operation was 0x00).

Jump on Not Zero:

JNZ Addr8	101101XX @@@@	if(!Z) PC = Addr8
-----------	---------------	-------------------

Jumps to an immediate address if the zero flag is not set (the result of the last ALU operation wasn't 0x00).

Jump on Carry:

JC Addr8	101110XX @@@@	if(C) PC = Addr8
----------	---------------	------------------

Jumps to an immediate address if the carry flag is set (the result of the last ALU operation caused an unsigned overflow: carry or borrow).

Jump on Not Carry:

JNC Addr8	101111XX @@@@	if(!C) PC = Addr8
-----------	---------------	-------------------

Jumps to an immediate address if the carry flag is not set (the result of the last ALU operation didn't cause an unsigned overflow).

Jump on overflow:

JV Addr8	110000XX @@@@	if(V) PC = Addr8
----------	---------------	------------------

Jumps to an immediate address if the overflow flag is set (the result of the last ALU operation caused a signed overflow).

Jump on Not overflow:

JNV Addr8	110001XX @@@@	if(!V) PC = Addr8
-----------	---------------	-------------------

Jumps to an immediate address if the carry flag is not set (the result of the last ALU operation didn't cause a signed overflow).

Jump on Negative:

JN Addr8	110010XX @@@@	if(S) PC = Addr8
----------	---------------	------------------

Jumps to an immediate address if the sign flag is set (the result of the last ALU operation is interpreted as negative in 2s complement: bit 7 is 1).

Jump on Positive:

JP Addr8	110011XX @@@@	if(!S) PC = Addr8
----------	---------------	-------------------

Jumps to an immediate address if the sign flag is not set (the result of the last ALU operation is interpreted as positive in 2s complement: bit 7 is 0). Note that 0x00 is considered positive.

Jump on Strictly Positive:

JSP Addr8	110100XX @@@@	if(!S & !Z) PC=Addr8
-----------	---------------	----------------------

Jumps to an immediate address if the sign flag AND the zero flag are not set: the result of the last ALU operation is interpreted as strictly positive (positive and not zero).

Jump on Less or Equal Unsigned:

JLEU Addr8	110101XX @@@@	if(C Z) PC = Addr8
------------	---------------	--------------------

Jumps to an immediate address if the zero flag OR the carry flag are set (corresponds to performing JZ followed by JC). If this is used after a "CMP-SUB Ra, Rb" instruction, the jump will be performed only if $Ra \leq Rb$ when interpreted as unsigned integers.

Note that a "Jump on Less Than Unsigned" instruction isn't needed since it's the same as using JC. Instead, it's implemented as an assembler macro.

Jump on Less Than signed:

JLT Addr8	110110XX @@@@	if(V^S) PC = Addr8
-----------	---------------	--------------------

Jumps to an immediate address if the overflow flag XOR the sign flag are set. If this is used after a "CMP-SUB Ra, Rb" instruction, the jump will be performed only if Ra<Rb when interpreted as signed integers.

Jump on Less or Equal signed:

JLE Addr8	110111XX @@@@	if((V^S) Z) PC=Addr8
-----------	---------------	----------------------

Jumps to an immediate address if either: the overflow flag XOR the sign flag are set, OR the zero flag is set. If this is used after a "CMP-SUB Ra, Rb" instruction, the jump will be performed only if Ra<=Rb when interpreted as signed integers.

LCD Command:

LCD-Com Imm8	111000XX IIIIIIII	LCD[Command] Imm8
--------------	-------------------	-------------------

Sends an immediate command to the LCD module in order to make it work. [See this document for available characters and commands.](#)

LCD Immediate character:

LCD-Imm Imm8	111001XX IIIIIIII	LCD[Data] Imm8
--------------	-------------------	----------------

Sends an immediate character for the LCD module to display. [See this document for available characters and commands.](#)

LCD character from Register:

LCD-Reg Ra	111010AA XXXXXXXX	LCD[Data] Ra
------------	-------------------	--------------

Sends a character stored in Ra for the LCD module to display. [See this document for available characters and commands.](#)

LCD character from Memory:

LCD-Mem Addr8	111011XX @@@@	LCD[Data] RAM[Addr8]
---------------	---------------	----------------------

Sends a character stored in memory (from an immediate address) for the LCD module to display. [See this document for available characters and commands.](#)

Decimal display from Register:

DEC-Reg Ra	111100XX @@@@	DEC Ra
------------	---------------	--------

Stores the contents of Ra in the output register for the decimal decoder to display (using a 4-digit 7-segment display).

Decimal display from Memory:

DEC-Mem Addr8	111101XX @@@@	LCD[Data] RAM[Addr8]
---------------	---------------	----------------------

Moves the contents stored in memory (from an immediate address) to the output register for the decimal decoder to display (using a 4-digit 7-segment display).

Halt:

HLT	111110XX XXXXXXXX	-
-----	-------------------	---

Halts the CPU clock and the program ends.

No Operation:

NOP	111111XX XXXXXXXX	-
-----	-------------------	---

Does nothing for 7 clock cycles. This instruction can be used safely as a placeholder. Since this architecture isn't pipelined, the only real use for NOP is to slow down programs that need to be run at high speeds (so that humans are able to read the results on the display before they disappear). Therefore, NOP wastes as many clock cycles as possible.

CESCA Application Binary Interface (ABI):

All subroutines for the CESC computer must follow these rules:

Passing arguments:

- The subroutine can accept up to 2 arguments, which are provided in **R0** and **R1**.
- If more arguments are required, those are stored in a vector in data memory and a pointer is provided as one of the arguments.

Returning values:

- The subroutine can return 1 value, by leaving it in **R0**.
- If a subroutine needs to return several values, those can be written to global variables or to a pointer provided as an argument.

Volatile and protected registers:

- **R0 and R1 are volatile:** Their contents may get wiped by subroutines.
- **R2 and R3 are protected:** Their contents will always be preserved between subroutine calls. The called subroutine is responsible for pushing their values to the stack and then restoring them if it's going to use those registers.
- **The flags are volatile:** The caller is responsible for taking precautions if it needs to use the state of the flags before the subroutine.

Variables in data memory:

- **Global variables** are stored in hardcoded positions in data memory and can be accessed using absolute addresses.
- If a subroutine needs to store its **local variables** in data memory, it's recommended to treat them as global variables and reserve a hardcoded address for them.
- If the latter is not possible, the subroutine should request a **pointer to safe space** as one of its arguments and store it's contents from there.

Return address:

- The return address is always at the **top of the stack**.
- This is the default behaviour when using JAL to call the subroutine and RET to return.

MACROS:

An assembler should provide at least the following macros in order to perform common actions with a single mnemonic. The mnemonic on the left gets replaced by the instruction(s) on the right of the arrow.

Compare aliases

Compare Ra to Rb:

CMP Ra, Rb → CMP-SUB Ra, Rb

Compare Ra to an immediate value:

CMP Ra, Imm8 → CMP-SUBI Ra, Imm8

Test register:

TEST Ra → CMP-MOVE Ra, Imm8

Increment / decrement

Increment Ra:

INCR Ra → ADDI Ra, Ra, 1

Decrement Ra:

DECR Ra → ADDI Ra, Ra, -1

LCD control

Initialize LCD:

LCD-Init → LCD-Com 0x38 LCD-Com 0x0E LCD-Com 0x06

Clear LCD:

LCD-Clr → LCD-Com 0x01

Conditional jumps (aliases)

Jump on equal:

JEQ *addr* → JZ *addr*

Jump on not equal:

JNE *addr* → JNZ *addr*

Jump on less than unsigned:

JLTU *addr* → JC *addr*

Conditional jumps (compare and jump)

Jump on Ra = Rb:

JEQ Ra, Rb, *addr* → CMP Ra, Rb JEQ *addr*

Jump on Ra = Imm8:

JEQ Ra, Imm8, *addr* → CMPI Ra, Imm8 JEQ *addr*

Jump on Ra != Rb:

JNE Ra, Rb, *addr* → CMP Ra, Rb JNE *addr*

Jump on Ra != Imm8: JNE Ra, Imm8, <i>addr</i> →	CMPI Ra, Imm8	JNE <i>addr</i>	
Jump on Ra < Rb unsigned: JLTU Ra, Rb, <i>addr</i> →	CMP Ra, Rb	JLTU <i>addr</i>	
Jump on Ra < Imm8 unsigned: JLTU Ra, Imm8, <i>addr</i> →	CMPI Ra, Imm8	JLTU <i>addr</i>	
Jump on Ra <= Rb unsigned: JLEU Ra, Rb, <i>addr</i> →	CMP Ra, Rb	JLEU <i>addr</i>	
Jump on Ra <= Imm8 unsigned: JLEU Ra, Imm8, <i>addr</i> →	CMPI Ra, Imm8	JLEU <i>addr</i>	
Jump on Ra > Rb unsigned: JGTU Ra, Rb, <i>addr</i> →	CMP Rb, Ra	JLTU <i>addr</i>	
Jump on Ra > Imm8 unsigned: JGTU Ra, Imm8, <i>addr</i> →	CMPI Ra, Imm8	JLEU (PC+2)	J <i>addr</i>
Jump on Ra >= Rb unsigned: JGEU Ra, Rb, <i>addr</i> →	CMP Ra, Rb	JNC <i>addr</i>	
Jump on Ra >= Imm8 unsigned: JGEU Ra, Imm8, <i>addr</i> →	CMPI Ra, Imm8	JNC <i>addr</i>	
Jump on Ra < Rb signed: JLT Ra, Rb, <i>addr</i> →	CMP Ra, Rb	JLT <i>addr</i>	
Jump on Ra < Imm8 signed: JLT Ra, Imm8, <i>addr</i> →	CMPI Ra, Imm8	JLT <i>addr</i>	
Jump on Ra <= Rb signed: JLE Ra, Rb, <i>addr</i> →	CMP Ra, Rb	JLE <i>addr</i>	
Jump on Ra <= Imm8 signed: JLE Ra, Imm8, <i>addr</i> →	CMPI Ra, Imm8	JLE <i>addr</i>	
Jump on Ra > Rb signed: JGT Ra, Rb, <i>addr</i> →	CMP Rb, Ra	JLT <i>addr</i>	
Jump on Ra > Imm8 signed: JGT Ra, Imm8, <i>addr</i> →	CMPI Ra, Imm8	JLEU (PC+2)	J <i>addr</i>
Jump on Ra >= Rb signed: JGE Ra, Rb, <i>addr</i> →	CMP Rb, Ra	JLE <i>addr</i>	
Jump on Ra >= Imm8 signed: JGE Ra, Imm8, <i>addr</i> →	CMPI Ra, Imm8	JLT (PC+2)	J <i>addr</i>

Jump on Ra positive:

JP Ra, *addr* → TEST Ra JP *addr*

Jump on Ra negative:

JN Ra, *addr* → TEST Ra JN *addr*

USEFUL SUBROUTINES:

Here are some subroutines for common tasks you may find useful.

Fast SRL (≥ 5 positions)

When an SRL has to be performed on a register 5 or more times, this method is more efficient. Note that the number of shifted positions (n) is hardcoded.

Arguments: The number to be shifted (on Ra).

Returns: The shifted result.

FastSRL:

{ ROL Ra, Ra } : Repeat line 8-n times

ANDI Ra, Ra, ($2^n - 1$) ; 2^n is the nth power of 2 (instead of xor)

RET

(More subroutines will be added here)

DEFICIENCIES:

- Can't shift / rotate more than 1 position at once.
- No multiplication and division instructions, must be done on a subroutine.
- No indexed addressing (like *LD Rd, 3(Ra)*, where the address is $3 + Ra$). Can't access SP directly.