What is CESCA / CESC?:

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

Features:

REGISTERS:

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: PC (Program Counter), **SP** (Stack Pointer), **MAR** (Memory Address Register), **IR** (Instruction Register) and an **Output Register** (all are 8 bit).
- 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 2 8-bit temporary registers in the ALU that hold the 2 operands.
- 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 (state register):
- Zero (Z): The result of the last ALU operation is 0.
- Carry (C): The last ALU operation caused an unsigned overflow (ADD = Carry, SUB = Borrow).
- Overflow (V): The last ALU operation caused a signed overflow.
- Sign (S): The result of the last ALU operation is negative (bit 7 = 1).

MEMORY:

1 KB (4x256) of addressable memory space: control logic chooses between 4 memory banks 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 using banks has the following implications:
- A program cannot modify its own code or create new programs.
- 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.
- A stack overflow won't corrupt program or data memory.

Bank configurations: The 4 banks can be split between volatile SRAM and an EEPROM:
- Both program memory banks are stored on an EEPROM: this makes entering the program very easy with an EEPROM programmer.
- Data and Stack banks are stored in RAM, since those are the only ones that the program can modify directly.

However, I decided <u>not</u> to do that and instead have all banks in SRAM for the following reasons:

- Even though the programmer needs to be more complex and be integrated inside the memory module, now it can also write to Data memory directly. This allows to enter programs with initialized global variables (some programs wouldn't fit in 256 instructions if they had to initialize their variables in RAM).
- The programmer can be built with an Arduino, so the RAM contents can also be <u>read</u> and sent to a computer, even on runtime. Due to restrictions on my design, this is the only way to manually look at the current contents in memory.

OUTPUT:

8 bit decimal display: Automatically decodes and outputs an 8 bit number. Very similar to Ben Eater's version but it also includes a hex mode in addition to signed and unsigned modes.
LCD panel: Displays any sequence of characters. It uses the same microcontroller as the module used in Ben's 6502 series.

REFERENCES:

- My design is based on [Ben Eater's version of the SAP-1](#), so it's quite modular and easy to expand.
- The ISA is a mix of MIPS, Ben's instruction set, and some freestyle (questionable at best) choices.
- I also took some inspiration from [James Sharman](#), [Circuit Breaker](#) and [James Bates](#).
- I recomend checking out Digital Computer Electronics, by A.P. Malvino and J.A. Brown (Part 2 contains the original SAP architectures and a PDF can be found online easily).
- Some [r/beneater](#) posts I found extremely useful:
  - Common issues and troubleshooting:
    - [What I Have Learned (a master list of what to do and what not to do)](#)
    - [The challenges that I resolved](#)
    - [What I learned on building the computer](#)
    - [Glitches on EEPROM datalines when their address changes](#)
    - [Frustration due to 28c64 EEPROMs](#)
    - [Post to help out (lowpass filter)](#) / [Bigger image](#)
  - Other interesting posts:
    - [Noise issue in monostable mode of Clock module](#)
    - [How to Make your Build Clean](#)
    - [My EEPROM ALU](#)
    - [Printed circuit boards](#)

The links aren't clickable on the GitHub PDF viewer. You will need to download the file first.

Instruction Set Architecture (ISA):

INSTRUCTION FORMATS:

```
Register:       OOOODDAA FFFFXXBB
Immediate:      OOOODDAA IIIIIIII ("I" can also be an address "@")
Reduced:        OOOODDAA 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  IIIIIIII`
	`ANDI Ra, Imm8`	`0010DDAA  IIIIIIII`
	`[CMP OPERATIONS] Ra, Rb`	`001100AA  FFFFXXBB`
	`CMP-SUBI Ra, Imm8`	`001101AA  IIIIIIII`
	`CMP-ANDI Ra, Imm8`	`001110AA  IIIIIIII`
	`CLF`	`001111XX  01XXXXXX`
Data movement:	`LI Rd, Imm8`	`0100DDXX  IIIIIIII`
	`ST-Addr Ra, Addr8`	`010100AA  @@@@@@@@`
	`ST-Reg Ra, Rb`	`010101AA  XXXXXXBB`
	`PUSH Ra`	`01011XAA  XXXXXXXX`
	`LD-Addr Rd, Addr8`	`0110DDXX  @@@@@@@@`
	`LD-Reg Rd, Ra`	`0111DDAA  XXXXXXXX`
	`POP Rd`	`1000DDXX  XXXXXXXX`
	`SWAP Rd, Ra`	`1001DDAA  XXXXXXXX`
Jump instructions:	`J Addr8`	`101000XX  @@@@@@@@`
	`JR Ra`	`101001AA  XXXXXXXX`
	`CALL 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  IIIIIIII`
	`LCD-Imm Imm8`	`111001XX  IIIIIIII`
	`LCD-Reg Ra`	`111010AA  XXXXXXXX`
	`LCD-Addr Addr8`	`111011XX  @@@@@@@@`
	`DEC-Reg Ra`	`111100AA  XXXXXXXX`
	`DEC-Addr 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 shfited left 1 position (corresponds to A+A).
1101	SRL Rd, Ra	Shift Right Logical: Ra gets shfited right 1 position (and filled with a 0).
1110	SRA Rd, Ra	Shift Right Arithmetic: Ra gets shfited 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. See the Macros in page 11.
- 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, 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. WARNING: You can use ADDI with 2s compliment immediates in order to subtract, but then the result on the Carry flag will be inverted! The reason is that since this instruction performs an ADD, the Carry flag will contain the carry instead of the borrow.

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 insn'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 ⚐

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 ⚐ = 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	010100AA @@@@@@@@	RAM[Addr8] = Ra

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

Store to address in Register:

ST-Reg Ra, Rb	010101AA @@@@@@@@	RAM[Rb] = Ra

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

Push to the stack:

PUSH Ra	01011XAA 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.

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.

Pop from the stack:

POP Rd	1000DDXX 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.

Swap top of the stack:

SWAP Rd, Ra	1001DDAA XXXXXXXX	Rd ⟵ Stack ⟵ Ra

Performs a PUSH from Ra and a POP to Rd at the same time (note that the SP is unchanged).
If Ra and Rd are the same register, this register gets swapped with the top of the stack (see macros in page 11). This is useful for having access to a fifth "virtual register" stored at the top of the stack, that gets swapped with a real register when it's needed and then swapped back.
<u>WARNING</u>: This does NOT swap the contents of Ra and Rd.

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.

Call subroutine:

CALL Addr8	101010XX @@@@@@@@	Stack PC+1; PC = Ra

Pushes the address of the <u>next</u> 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 <u>not</u> 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(!Z) PC = Addr8

Jumps to an immediate address if the carry flag is <u>not</u> 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 <u>not</u> 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 compliment: bit 7 is 0).

Jump on Postitve:

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

Jumps to an immediate address if the sign flag is <u>not</u> set (the result of the last ALU operation is interpreted as positive in 2s compliment: bit 7 is 1). Note that 0x00 is considered positive.

Jump on Strictly Postitve:

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

Jumps to an immediate address if the sign flag AND the zero flag are <u>not</u> 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<=Rb when interpreted as <u>unsigned</u> 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 (see macros in page 11).

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 <u>signed</u> 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 <u>signed</u> 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. <u>See this document for available characters and commands</u> (**Documentation / CESCA LCD interface.pdf** on Github).

LCD Immediate character:

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

Sends an immediate character for the LCD module to display. <u>See this document for available characters and commands</u> (**Documentation / CESCA LCD interface.pdf** on Github).

LCD character from Register:

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

Sends a character stored in Ra for the LCD module to display. <u>See this document for available characters and commands</u> (**Documentation / CESCA LCD interface.pdf** on Github).

LCD character from memory Address:

LCD-Addr 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 (**Documentation / CESCA LCD interface.pdf** on Github).

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 Address:

DEC-Addr 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.

Application Binary Interface (ABI):

My architecture is extremely simple and it's obviously not capable of multitasking or handling an OS, so there is no real need for an ABI. However, since it <u>is</u> capable of an arbitrary depth of subroutine calls, I believe it's useful to have a set of rules all subroutines should follow.

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 saved:** Their contents will always be preserved between subroutine calls. The <u>called</u> 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 <u>caller</u> 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 CALL to call the subroutine and RET to return.

* In some cases, if it's strictly necessary for optimizing the speed of a common subroutine, **R2** and/or **R3** may be used for passing arguments. If that's the case, their content doesn't need to be preserved.

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 side gets replaced by the instruction(s) on the right side 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
```

Operation aliases

```
Increment Ra:
INCR Ra                 →          ADDI Ra, Ra, 1

Decrement Ra:
DECR Ra                 →          ADDI Ra, Ra, -1
```
WARNING: The Carry flag will be inverted! See instruction details for **ADD Immediate** in page 3.

```
Shift Left Logical with Carry:
SLLC Rd, Ra             →          ADDC Rd, Ra, Ra

Swap Ra with top of stack:
SWAP Ra                 →          SWAP Ra, Ra
```

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)
Those macros aren't needed in the assembler, but some people may find them useful:

```
Jump on Ra = Rb:
JEQ Ra, Rb, addr          →          CMP Ra, Rb          JEQ addr

Jump on Ra = Imm8:
JEQ Ra, Imm8, addr        →          CMP Ra, Imm8        JEQ addr


Jump on Ra != Rb:
JNE Ra, Rb, addr          →          CMP Ra, Rb          JNE addr

Jump on Ra != Imm8:
JNE Ra, Imm8, addr        →          CMP Ra, Imm8        JNE addr


Jump on Ra < Rb unsigned:
JLTU Ra, Rb, addr         →          CMP Ra, Rb          JLTU addr

Jump on Ra < Imm8 unsigned:
JLTU Ra, Imm8, addr       →          CMP Ra, Imm8        JLTU addr


Jump on Ra <= Rb unsigned:
JLEU Ra, Rb, addr         →          CMP Ra, Rb          JLEU addr

Jump on Ra <= Imm8 unsigned:
JLEU Ra, Imm8, addr       →          CMP Ra, Imm8        JLEU addr


Jump on Ra > Rb unsigned:
JGTU Ra, Rb, addr         →          CMP Rb, Ra          JLTU addr

Jump on Ra > Imm8 unsigned:
JGTU Ra, Imm8, addr       →          CMP Ra, Imm8        JLEU (PC+2)     J addr


Jump on Ra >= Rb unsigned:
JGEU Ra, Rb, addr         →          CMP Ra, Rb          JNC addr

Jump on Ra >= Imm8 unsigned:
JGEU Ra, Imm8, addr       →          CMP Ra, Imm8        JNC addr


Jump on Ra < Rb signed:
JLT Ra, Rb, addr          →          CMP Ra, Rb          JLT addr

Jump on Ra < Imm8 signed:
JLT Ra, Imm8, addr        →          CMP Ra, Imm8        JLT addr
```

```
Jump on Ra <= Rb signed:
JLE Ra, Rb, addr              →         CMP Ra, Rb            JLE addr


Jump on Ra <= Imm8 signed:
JLE Ra, Imm8, addr            →         CMP Ra, Imm8          JLE addr



Jump on Ra > Rb signed:
JGT Ra, Rb, addr              →         CMP Rb, Ra            JLT addr

Jump on Ra > Imm8 signed:
JGT Ra, Imm8, addr            →         CMP Ra, Imm8          JLEU (PC+2)     J addr



Jump on Ra >= Rb signed:
JGE Ra, Rb, addr              →         CMP Rb, Ra            JLE addr

Jump on Ra >= Imm8 signed:
JGE Ra, Imm8, addr            →         CMP Ra, Imm8          JLT (PC+2)      J addr
```

Useful subroutines:

See this file for a collection of subroutines (**Assembly / Examples / subroutine_library.asm** on Github).

Limitations:

Managing large workloads:
- 8 bit architecture means that all work with large (16 bit) numbers needs to be done on slow dword operations. However, there is no real workaround for this on a breadboard computer.
- Only 1kB of memory: maximum capacity of just 256 instructions and 256 bytes of data. This is a big improvement over the original 16 bytes of the SAP-1, but a bigger space would allow for more complex programs. 256 bytes of stack should be more than enough.

Instruction set flexibility:
- No multiplication and division instructions, those must be done on a subroutine.
- Can't shift / rotate more than 1 position at once.
- No indexed addressing (like *LD Rd, 3(Ra),* where the address is 3 + Ra). This, combined with direct access to the stack pointer, would allow storing local variables in the stack using *ST Ra, offset(Sp).*