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 in runtime. Due to restrictions on my design, this is the only way to manually look at the current contents in memory.

Input/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.
- **Keyboard controller**: An Arduino translates key presses from a PS/2 keyboard to an "Input register". Then, this fake resgister can be read by the computer and acknowledged when it's done (the "register" gets cleared and the next input gets loaded).

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)
        - 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
```

INSTRUCTION TABLE:

	Mnemonic	Machine code	Cycles
ALU:	ALU* Rd, Ra, Rb	0000DDAA FFFFXXBB	4
	add Rd, Ra, Imm8	0001DDAA IIIIIIII	4
	ALU* NONE, Ra, Rb	001000AA FFFFXXBB	3
	cmp Ra, Imm8	001001AA IIIIIIII	3
	mask Ra, Imm8	001010AA IIIIIIII	3
	mask IN, Imm8	001011XX IIIIIIII	5
Memory:	mov Rd, Imm8	0011DDXX IIIIIIII	3
	mov Rd, IN	0100DDXX XXXXXXXX	4
	mov IN, Ack	010100XX XXXXXXXX	3
	mov [Addr8], Ra	010101AA @@@@@@@@	4
	mov [Rb], Ra	010110AA XXXXXXBB	5
	push Ra	010111AA XXXXXXXX	5
	mov Rd, [Addr8]	0110DDXX @@@@@@@@	4
	mov Rd, [Ra]	0111DDAA XXXXXXXX	4
	pop Rd	1000DDXX XXXXXXXX	4
	swap Rd, Ra	1001DDAA XXXXXXXX	5
Jumps:	jmp Addr8	101000XX @@@@@@@@	3
	jmp Ra	101001AA XXXXXXXX	3
	call Addr8	101010XX @@@@@@@@	6
	ret	101011XX XXXXXXXX	4
	jz Addr8	101100XX @@@@@@@@	3
	jnz Addr8	101101XX @@@@@@@@	3
	jc Addr8	101110XX @@@@@@@@	3
	jnc Addr8	101111XX @@@@@@@@	3
	jv Addr8	110000XX @@@@@@@@	3
	jnv Addr8	110001XX @@@@@@@@	3
	jlz Addr8	110010XX @@@@@@@@	3
	jp Addr8	110011XX @@@@@@@@	3
	jgz Addr8	110100XX @@@@@@@@	3
	jleu Addr8	110101XX @@@@@@@@	3
	jlt Addr8	110110XX @@@@@@@@	3
	jle Addr8	110111XX @@@@@@@@	3
Output & misc.:	mov LCDcmd, Imm8	111000XX IIIIIIII	3
	mov LCD, Imm8	111001XX IIIIIIII	3
	mov LCD, Ra	111010AA XXXXXXXX	3
	mov LCD, [Addr8]	111011XX @@@@@@@@	4
	mov OUT, Ra	111100AA XXXXXXXX	3
	mov OUT, [Addr8]	111101XX @@@@@@@@	4
	hlt	111110XX XXXXXXXX	2
	nop	111111XX XXXXXXXX	8

*ALU Operations:

Funct	Mnemonic	Description / observations
0000	mov Rd, Rb	Move the contents of Rb 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 AND between Ra and Rb.
0110	or Rd, Ra, Rb	Performs a bitwise OR between Ra and Rb.
0111	not Rd, Ra	Performs a bitwise NOT to Ra.
1000	xor Rd, Ra, Rb	Performs a bitwise XOR between Ra and Rb.
1001	nand Rd, Ra, Rb	Performs a bitwise NAND between Ra and Rb.
1010	nor Rd, Ra, Rb	Performs a bitwise NOR between Ra and Rb.
1011	xnor Rd, Ra, Rb	Performs a bitwise 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 state of the oVerflow flag is undefined for all operations except `add`, `sub`, `addc` and `subb`.
- After a shift / rotation, 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, and the oVerflow flag is undefined.

- The fake register `NONE` can be used as the destination register for any of those mnemonics. The flags will be updated without writing the ALU result to any register (useful for comparing and testing registers).
- The `mov`, `add`, `cmp`, and `mask` instructions have register and immediate versions: they can be used with either a register (Rb) or an 8-bit immediate value as their second operand.

- The `not`, `srl`, `sra` and `rol` instructions ignore the value stored in Rb.

INSTRUCTIONS (Detailed):

ALU Operations:

ALU 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:

add 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 a 2s compliment immediate in order to subtract, but then the result on the Carry flag will be <u>inverted</u>! The reason is that since this instruction performs an `add`, the Carry flag will contain the carry instead of the borrow.

Compare Operations:

ALU NONE, Ra, Rb	001000AA 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. This is useful for setting the flags without messing up the stored contents.
`sub` is used to compare 2 registers, and masks can be used with `and`. These operations are extremely common, so it's recommended to use the `cmp` and `mask` macros instead.
See table above for ALU operations, mnemonics and descriptions.

Compare with Immediate:

cmp Ra, Imm8	001001AA IIIIIIII	Ra - Imm8 ⚐

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

Mask with Immediate:

mask Ra, Imm8	001010AA IIIIIIII	Ra & Imm8 ⚐

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

Mask Input register:

mask IN, Imm8	001011XX IIIIIIII	IN_Reg & Imm8 ⚐

Performs a bitwise AND between the Input register and an immediate value. This is useful for quickly testing or polling the input without having to write to a register.
Read **Documentation/Keyboard_interface.pdf** to learn how to interpret the contents of the Input register.

Move Immediate:

mov Rd, Imm8	0011DDXX IIIIIIII	Rd = Imm8

Moves an immediate value into Rd.

Move Input into register:

mov Rd, IN	0100DDXX XXXXXXXX	Rd = IN_Reg

Copies the contents of the Input register to Rd. Read **Documentation/Keyboard_interface.pdf** to learn how to interpret the contents of the Input register.

Acknowledge Input:

mov IN, Ack	010100XX XXXXXXXX	IN_Reg ← Ack

Sends the *Ack* signal to the Input controller, indicating that the computer has finished processing the current input. The contents of the Input register get set to 0x00 and the next input gets processed.

Store to Address (direct addressing):

mov [Addr8], Ra	010101AA @@@@@@@@	RAM[Addr8] = Ra

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

Store to address (indirect addressing):

mov [Rb], Ra	010110AA @@@@@@@@	RAM[Rb] = Ra

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

Push to the stack:

push Ra	010111AA 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 (direct addressing):

mov Rd, [Addr8]	0110DDXX @@@@@@@@	Rd = RAM[Addr8]

Loads into Rd the memory contents from an immediate address.

Load from address (indirect addressing):

mov 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:

jmp Addr8	101000XX @@@@@@@@	PC = Addr8

Jumps unconditionally to an immediate address.

Jump to Register:

jmp Ra	101001AA XXXXXXXX	PC = Ra

Jumps unconditionally to the address stored in a register.

Call subroutine:

call Addr8	101010XX @@@@@@@@	Stack←PC+1; PC=Addr8

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 if 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 if 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 if 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 if 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 if 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 if 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 if Less than Zero:

jlz 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 if 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 if Greater than Zero:

jgz 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 if 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 Ra, Rb` instruction, the jump will be performed only if Ra<=Rb when interpreted as <u>unsigned</u> integers.
Note that a "Jump if 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 if 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 instruction is used after `cmp Ra, Rb`, the jump will be performed only if Ra<Rb when interpreted as <u>signed</u> integers.

Jump if 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 Ra, Rb` instruction, the jump will be performed only if Ra<=Rb when interpreted as <u>signed</u> integers.

Send Command to LCD:

mov LCDcmd, Imm8	111000XX IIIIIIII	LCD[Command] ← Imm8

Sends an immediate command to the LCD module in order to make it work. Read **Documentation/LCD_interface.pdf** for available characters and commands.

Send Character to LCD (immediate):

mov LCD, Imm8	111001XX IIIIIIII	LCD[Data] ← Imm8

Sends an immediate character for the LCD module to display. Read **Documentation/LCD_interface.pdf** for available characters and commands.

Send Character to LCD (register):

mov LCD, Ra	111010AA XXXXXXXX	LCD[Data] ← Ra

Sends a character stored in Ra for the LCD module to display. Read **Documentation/LCD_interface.pdf** for available characters and commands.

Send Character to LCD (memory):

mov LCD, [Addr8]	111011XX @@@@@@@@	LCD[Data]←RAM[Addr8]

Sends a character stored in memory (from an immediate address) for the LCD module to display. Read **Documentation/LCD_interface.pdf** for available characters and commands.

Send Output (register):

mov OUT, Ra	111100XX @@@@@@@@	OUT_Reg ← Ra

Stores the contents of Ra in the output register. The decimal decoder automatically outputs its contents using a 4-digit 7-segment display.

Send Output (memory):

mov OUT, [Addr8]	111101XX @@@@@@@@	OUT_Reg ← RAM[Addr8]

Moves the contents stored in memory (from an immediate address) to the output register. The decimal decoder automatically outputs its contents 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 8 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 its 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 commonly used 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              →              sub NONE, Ra, Rb


Test register:
test Ra                 →              mov NONE, Ra
```

Operation aliases

```
Increment Ra:
inc Ra                  →              add Ra, Ra, 1


Decrement Ra:
dec Ra                  →              add 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
```

Input and output

```
Initialize LCD:
mov LCDcmd, Init        →              mov LCDcmd, 0x38    ; 2 lines. Font size: 8 dots
                                       mov LCDcmd, 0x0E    ; Display ON, static cursor ON
                                       mov LCDcmd, 0x06    ; Increment on input, shift OFF


Clear LCD:
mov LCDcmd, Clr         →              mov LCDcmd, 0x01
```

Conditional jumps (aliases)

```
Jump if equal:
jeq addr                →              jz addr


Jump if not equal:
jne addr                →              jnz addr


Jump if 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 if Ra > Rb unsigned:
jgtu Ra, Rb, addr        →        cmp Rb, Ra          jltu addr


Jump if Ra > Imm8 unsigned:
jgtu Ra, Imm8, addr      →        cmp Ra, Imm8        jleu skip(1)   jmp addr



Jump if Ra >= Rb unsigned:
JGEU Ra, Rb, addr        →        cmp Ra, Rb          jnc addr


Jump if Ra >= Imm8 unsigned:
JGEU Ra, Imm8, addr      →        cmp Ra, Imm8        jnc addr



Jump if Ra > Rb signed:
JGT Ra, Rb, addr         →        cmp Rb, Ra          jlt addr


Jump if Ra > Imm8 signed:
JGT Ra, Imm8, addr       →        cmp Ra, Imm8        jle skip(1)    jmp addr



Jump if Ra >= Rb signed:
JGE Ra, Rb, addr         →        cmp Rb, Ra          jle addr


Jump if Ra >= Imm8 signed:
JGE Ra, Imm8, addr       →        cmp Ra, Imm8        jlt skip(1)    jmp addr
```

Useful subroutines:

- [Library of useful math subroutines](#) (**Assembly / Examples / MATH.asm** on Github).
- [Other interesting subroutines](#) (**Assembly / Examples / interesting_subroutines.asm** on Github).

Limitations:

Managing large workloads:
- 8 bit architecture means that all work with larger (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.
- No support for interrupts. This method would be more efficient for handling input than the current polling system (and the timing requirements would be less strict), but it's way more complicated.
- 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).*