CESC16 Documentation

WHAT IS CESC16?

CESC16 is the 16 bit version of the CESC architecture (Competent but Extremely Simple Computer), a homebrew breadboard computer I designed previously.

The greatest limitation of the old design was the 256 instruction limit caused by the 8 bit architecture. I could have kept the 8 bit ALU and expanded just the MAR, but I wanted to keep things simple and having to mix 8 bit and 16 bit values made instructions very slow.

This iteration was meant to be built on a PCB from the start, so I could afford doing major changes that wouldn't have been possible on a breadboard:

- The entire architecture is 16 bit wide: this makes jumps very fast (since the address can be copied all at once) and avoids having to use slow dword operations for big numbers.
- The register file has been expanded greatly, now having 14 GPRs instead of 4.

You can find the GitHub repository for the original CESC architecture at github.com/p-rivero/CESCA.

FEATURES:

- Simple but powerful RISC architecture without pipelining, inspired by MIPS and RISC-V
    - Most instructions take 3 or 4 clock cycles (see "Cycles" on instruction table below)
    - Native stack capabilities to speed up push/pop and call/return instructions
    - Supports signed and unsigned comparisons for conditional jumping
    - Even though it's a RISC architecture, Direct, Indirect, Reg+Reg and Reg+Imm addressing modes are supported for *all ALU operations*, using a sort of x86 Intel syntax.
    - Supports hardware interrupts from up to 4 GPIO ports (each with a dedicated IRQ line)
    - Like other homebrew CPUs, its microcode is easy to reprogram to make new instructions
- **2 MHz** clock rate + **0.72 Hz to 480 Hz** variable speed 555 timer + manual clock pulse mode
- 16 bit architecture: ALU, Register File and Memory Addresses are all 16 bit wide.
- 256 kB ROM + 128 kB RAM*: 64k instructions (32 bit wide) + 64k* data words (16 bit wide)
    - ROM is split in two 16 bit banks, so each address contains 32 bits of data
    - * 512 bytes (256 words) of RAM are reserved for controlling the MMIO.
- 14 general purpose registers (GPRs): 5 safe, 4 temporary, 4 arguments, 1 base pointer

  2 special purpose registers: Zero constant, Stack Pointer
    - All of them can be used as operands and/or destination for ALU/Memory instructions
    - The destination and operands can all be different registers (like in MIPS and RISC-V)
- CPU created entirely with discrete logic chips (74 series); I/O implemented with 3 Arduino Nano.
    - **Input:** Arduino translates the PS/2 keyboard input to ASCII and causes an interrupt.
    - **Output:** Arduino outputs video signal for a VGA terminal. More info in this video.
        - Displays 25 lines with 40 text characters each, on an 8x8 pixel font
        - Resolution of 320x480 pixels @ 60Hz, 6 bit color (64 colors)
    - **Disk:** Using an Arduino and a CH376 module, the CPU can read/write a USB drive.

Read DOCS/Input.pdf and DOCS/Output.pdf to learn how to get input and send output. **[in progress]**

INSTRUCTION FORMAT:

OOOOm FFF DDDD AAAA	Argument (16 bits)

O: Opcode **F**: Opcode modifier (**F**unction)

m: Opcode modifier (Addressing **m**ode): 0=Register, 1=Immediate

D: rD (Destination register) **A**: rA (1st operand register)

X: Don't care

Possible arguments:

XXXXXXXXXXXX**BBBB**: rB (2nd operand register)

IIIIIIIIIIIIIIII: Immediate 16 bit value/address

IIIIIIIIIIIIbbbb: Either of the previous 2 options (depending on m bit)

INSTRUCTION TABLE:

	Mnemonic	Machine code	Cycles
ALU:	ALU rD, rA, rB/Imm16	0000mFFFDDDDAAAA IIIIIIIIIIIIbbbb	3
	sll rD, rA, Imm4	0001IIIIDDDDAAAA XXXXXXXXXXXXXXXX	Imm+1
	srl rD, rA, Imm4	0010IIIIDDDDAAAA XXXXXXXXXXXXXXXX	Imm+1
	sra rD, rA, Imm4	0011IIIIDDDDAAAA XXXXXXXXXXXXXXXX	Imm+1
	ALU rD, rA, [mode]	010mmFFFDDDDAAAA IIIIIIIIIIIIbbbb	4/5
	ALU [mode], rB	011mmFFFbbbbAAAA IIIIIIIIIIIIbbbb	4/5
Memory:	swap rD, [rA+Imm16]	10000001DDDDAAAA IIIIIIIIIIIIIIII	5
	peek rD, [rA+Imm16], W	1000001WDDDDAAAA IIIIIIIIIIIIIIII	3
	push rB/Imm16	1000010mXXXX0001 IIIIIIIIIIIIbbbb	3
	pushf	10000110XXXX0001 XXXXXXXXXXXXXXXX	3
	pop rD	10000111DDDD0001 XXXXXXXXXXXXXXXX	3
	popf	10001000XXXX0001 XXXXXXXXXXXXXXXX	3
Jumps:	JMP rA/Imm16	110mFFFFXXXXAAAA IIIIIIIIIIIIIIII	2
	call rB/Imm16	1110000mXXXX0001 IIIIIIIIIIIIbbbb	4
	syscall rB/Imm16	1110001mXXXX0001 IIIIIIIIIIIIbbbb	4
	enter rB/Imm16	1110010mXXXX0001 IIIIIIIIIIIIbbbb	4
	ret	11100110XXXX0001 XXXXXXXXXXXXXXXX	3
	sysret	11100111XXXX0001 XXXXXXXXXXXXXXXX	3
	exit	11101000XXXX0001 XXXXXXXXXXXXXXXX	3

Additional comments:

- ALU means any mnemonic from the "ALU Operations" table below.
- JMP means any mnemonic from the "Jump Conditions" table below.
- Read DOCS/Instructions.pdf to learn how to use each instruction, the available addressing modes and macros, and more in-depth information in general.

Flags:

Flag name		Meaning of active flag (`Flag=1`)
`Z`	`Zero`	The result of the last ALU operation was exactly 0x0000.
`C`	`Carry`	The last `add` or `sub` operation caused an unsigned overflow (carry or borrow).
`V`	`Overflow`	The last `add` or `sub` operation caused a signed overflow.
`S`	`Sign bit`	The result of the last ALU operation was negative when interpreted in 2's complement.

Jump Conditions:

FFFF	Mnemonic(s)	Pseudocode	Description
`0000`	`jmp`	`1`	Jump unconditionally (always jumps)
`0001`	`jz` `je`	`Z`	Jump if Zero Jump if Equal
`0010`	`jnz` `jne`	`!Z`	Jump if Not Zero Jump if Not Equal

`0011`	`jc`
	`jb`
	`jnae`
		`C`	Jump if Carry
			Jump if Below (unsigned `<`)
			Jump if Not Above or Equal

`0100`	`jnc`
	`jae`
	`jnb`
		`!C`	Jump if Not Carry
			Jump if Above or Equal (unsigned `>=`)
			Jump if Not Below

`0101`	`jo`	`V`	Jump if Overflow
`0110`	`jno`	`!V`	Jump if Not Overflow
`0111`	`js`	`S`	Jump if Sign (MSB = 1)
`1000`	`jns`	`!S`	Jump if Not Sign (MSB = `0`)
`1001`	`jbe` `jna`	`C || Z`	Jump if Below or Equal (unsigned `<=`) Jump if Not Above
`1010`	`ja` `jnbe`	`!C && !Z`	Jump if Above (unsigned `>`) Jump if Not Below or Equal
`1011`	`jl` `jnge`	`V != S`	Jump if Less (signed `<`) Jump if Not Greater or Equal
`1100`	`jle` `jng`	`(V!=S) || Z`	Jump if Less or Equal (signed `<=`) Jump if Not Greater
`1101`	`jg` `jnle`	`(V==S) && !Z`	Jump if Greater (signed `>`) Jump if Not Less or Equal
`1110`	`jge` `jnl`	`V == S`	Jump if Greater or Equal (signed `>=`) Jump if Not Less
`1111`	-	-	Unused

ALU Operations:

FFF	Mnemonic	Pseudocode	Description / observations
`000`	`mov D, A`	`D = A`	Moves the contents of A into D (without updating the flags!).
`001`	`and D, A, B`	`D = A&B`	Performs a bitwise logic AND between A and B.
`010`	`or D, A, B`	`D = A|B`	Performs a bitwise logic OR between A and B.
`011`	`xor D, A, B`	`D = A^B`	Performs a bitwise logic XOR between A and B.
`100`	`add D, A, B`	`D = A+B`	Adds the contents of A and B.
`101`	`sub D, A, B`	`D = A-B`	Subtracts the contents of A and B.
`110`	`addc D, A, B`	`D = A+B+C`	Add with Carry: Add 1 to result if last operation caused carry.
`111`	`subb D, A, B`	`D = A-B-C`	Subtract with Borrow: Subtract 1 to result if last op. caused borrow.

CALLING CONVENTION:

Registers:

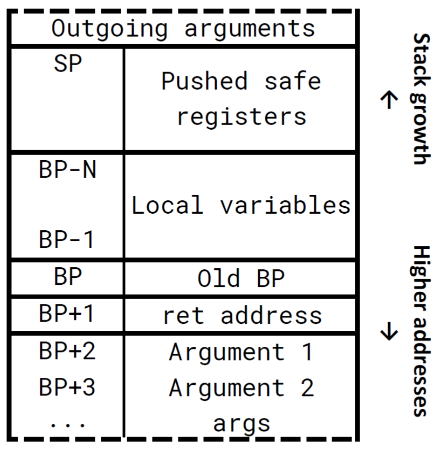
Machine code	Name	Description	Saver
`0000`	`zero`	Hardwired to `0x0000` (read only). Useful for implementing macros and as a quick source for the value 0 (boolean `false`).	-
`0001`	`sp`	Stack pointer. The OS initializes it to `0xFF00`. A subroutine can allocate memory for its local variables by decrementing it.	Callee
`0010`	`bp`	Base Pointer. Fixed reference point for the funtion. All local variables and stack arguments are located at a constant offset (see ABI).	Callee
`0011..0111`	`s0 - s4`	Safe registers that get preserved in a subroutine call. The callee must push and then restore the registers it wants to use (see ABI).	Callee
`1000..1011`	`t0 - t3`	Temporary registers that can be wiped by a called subroutine. Useful for storing intermediate values.	Caller
`1100..1111`	`a0 - a3`	Used for passing arguments to a called subroutine. `a0` is also used for storing the return value (the subroutine can wipe their content).	Caller

Remarks:

- The implemented stack is a *full stack*: `sp` points at the actual data, not the next free space.
- Some instructions (`push`, `pop`, `call`, `ret`, ...) modify the `sp` automatically.
- Flags are not preserved in a subroutine call. The caller can save its flags to the stack by using the `pushf` instruction before performing the call, and restore them afterwards using `popf`.

Structure of the call frame:

See the calling conventions below to see how the call frame is constructed.



Calling convention for variadic functions:

The C standard supports variadic functions (functions with a variable amount of arguments). The calling convention for those functions is almost identical to the `cdecl` x86 calling convention.

- The bp register is used in order to access arguments and local variables. The callee is responsible for saving its value.
- All arguments are pushed to the stack, *from right to left*. The first argument is located at the memory address bp+2 (since bp+0 stores the old bp, and bp+1 stores the return address).
- Once the call frame has been set up, the subroutine may allocate space for local variables by subtracting a number of words from sp. The first variable is stored at bp-1, the second variable is at bp-2, and so on.
- Return values are always stored in a0.

```c
int callee(int one, int two, ...);

void example(int A, int B, int C, ...) {
      int local;  int other_stuff[9];
      local += callee(1, A, B);
}
```

```
example:
  push bp         ; save old call frame
  mov bp, sp      ; initialize new call frame
  sub sp, sp, 10  ; allocate 10 words for locals
  push s0         ; push the used safe registers
  mov t0, [bp+3]  ; fetch B
  push t0
  mov t0, [bp+2]  ; fetch A
  push t0
  push 1
  call callee     ; call subroutine 'callee'
  add sp, sp, 2   ; remove call args from frame
  add [bp-1], a0  ; local += result from 'callee'
  pop s0          ; restore used safe registers
  mov sp, bp      ; restore old stack pointer
  pop bp          ; restore old call frame
  ret             ; return
```

Calling convention for non-variadic functions:

For regular functions (fixed number of arguments), it's possible to optimize the calling convention by using the argument registers (a0, a1, a2, a3).

- The first (leftmost) 4 arguments are passed in registers. The rest of the arguments are pushed to the stack from right to left (and read using bp). Therefore, the 4th argument is located in a3 and the 5th argument is in memory address bp+2.
- Recall that the argument registers are not preserved by a subroutine call. Therefore, an incoming argument is needed after a call, first it must be moved to a local variable (either in the stack or in a safe register).

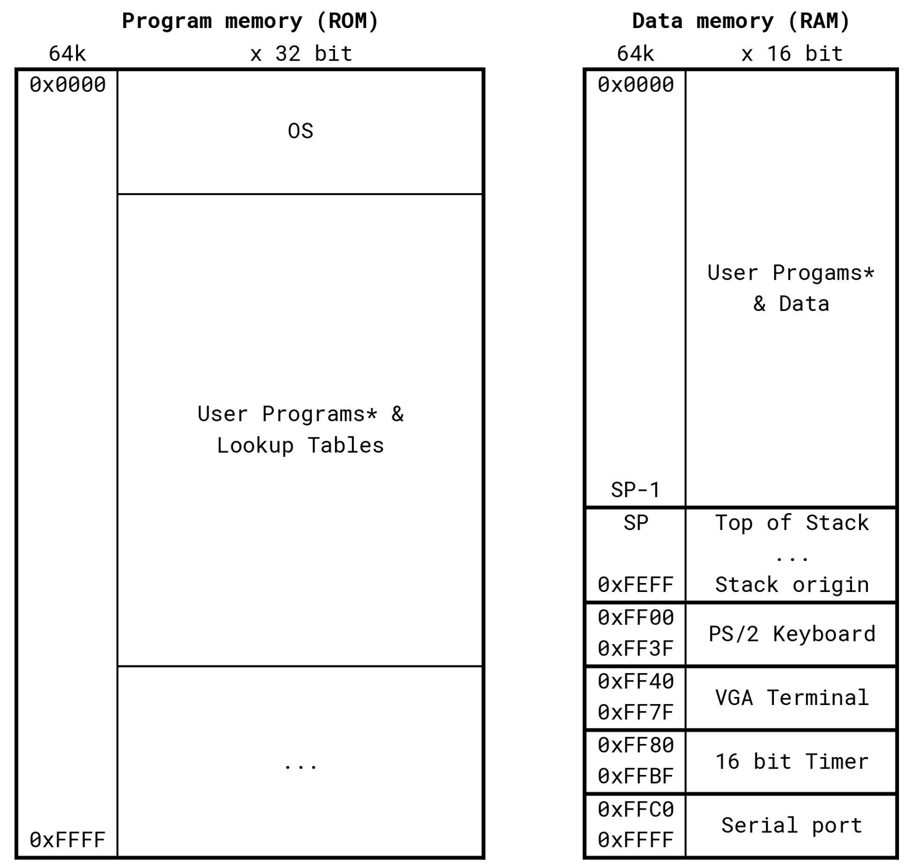
```c
int callee(int one, int two, int three);

void example(int A, int B, int C) {
      int local;  int other_stuff[9];
      local += callee(1, A, B);
}
```

```
example:
  push bp         ; save old call frame
  mov bp, sp      ; initialize new call frame
  sub sp, sp, 10  ; allocate 10 words for locals
  push s0         ; push the used safe registers
  mov s2, s1      ; use B as third argument
  mov s1, s0      ; use A as second argument
  mov s0, 1
  call callee     ; call subroutine 'callee'
  add [bp-1], a0  ; local += result from 'callee'
  pop s0          ; restore safe registers
  mov sp, bp      ; restore old stack pointer
  pop bp          ; restore old call frame
  ret             ; return
```

MEMORY MAP:

There are 2 separate memory spaces, both indexed with a 16-bit address (64k words). Code can be fetched and executed from either of them*.

- **Program Memory (ROM, 32 bit wide):** Cannot be written, and its contents are always available. By default, instructions are fetched from ROM. The `peek` instruction always accesses ROM.
- **Data Memory (RAM, 16 bit wide):** Its contents are undefined after boot (must be constructed at runtime). The rest of the memory instructions (including stack operations) always access RAM.



* Instructions can be stored in one 32-bit word of ROM, or in two 16-bit words of RAM (using big-endian format: the opcode goes first, followed by the argument).

Program flow:

Execution starts at address 0x0000 of ROM. The OS handles startup and then calls user code (also stored in ROM). The user code can do one of 2 things:

- Stay in ROM, using RAM just as storage (Harvard Architecture).
- Construct a program in RAM, either taking user input (assembler) or not (self-modifying code), and then call it using the `enter` instruction.
    - The program in RAM can call subroutines in ROM using the `syscall` instruction. Those ROM subroutines must finish with a `sysret` instruction. If they don't (like the OS routines!), then they must be called using the `CALL_GATE` routine (read the *"Calling system routines"* section below).
    - Once the program in RAM finishes, it can return control to the ROM program that called it by using the `exit` instruction.

Once the main user program (in ROM) finishes, it can return control to the OS using a regular `ret` instruction. The OS can then halt or restart the computer, open a ROM monitor, etc.

OPERATING SYSTEM:

Since CESC16 is an extremely simple architecture, all code runs at kernel level. Therefore, an OS isn't really needed in terms of security, but it's still useful to have a collection of commonly used subroutines and a safe hardware entry point.

The current OS draft includes:

- Startup code for resetting the CPU correctly and calling the user code (and then terminating user code execution safely), as well as handling interrupts.
- Math library with commonly used functions: 16 and 32 bit multiplication and division/modulus (more to be added later: exponentiation, square root, trigonometric functions).
- Input and Output libraries for writing hardware-agnostic code with I/O. They also allow the user to call the OS utilities directly using interrupts.
- Utilities and debug tools: Startup menu that allows the user to launch a specific program (if the ROM contains more than one) and a RAM/ROM monitor. **[work in progress]**

Calling system routines:

Programs that are stored in RAM must use `syscall` to call an OS routine (this way the program jumps to ROM instead of a random position in RAM). However, all OS routines assume they are called from ROM and will end with a regular `ret` instruction (therefore, program execution will stay in ROM)!

In order to ensure that the call returns safely to RAM, instead of calling the routine directly, the program must store the address it wishes to call in `t0`, and then perform: `syscall CALL_GATE`.

The code of CALL_GATE is shown below:

```
CALL_GATE:
    call t0
    sysret
```

Programs that are stored in ROM can call OS routines directly using the regular `call` instruction. However, in order to increase code clarity, `syscall` should be used instead (when used from ROM, it behaves like a regular `call`). Do not use CALL_GATE on programs that are being fetched from ROM.

A full list of the available system calls can be found in DOCS/syscalls.md **[work in progress]**

ROOM FOR IMPROVEMENT:

The goal of CESC16 is to be a powerful architecture while still being as simple as possible. This causes it to be in a weird spot, since the lack of advanced features means it's far from a real consumer CPU, but its chip count and PCB cost is much higher than most homebrew CPUs.

<u>To be a real consumer CPU it would need</u>:

- Better I/O support (more than 4 devices), and being able to mask interrupt sources.
- Actual pipelining (as close to 1 cycle per instruction as possible).
- Hardware multiplication/division, since it's an extremely common operation.

<u>To greatly reduce its chip count it would need</u>:

- 8 bit architecture would be fast enough, as long as it has 16 bit addresses.
- Less registers (use addressing modes to save most variables in SRAM). Having many physical registers (like in MIPS and RISC-V) only makes sense if it allows you to implement pipelining.

Embracing either a minimalistic 8 bit architecture (allowing a faster clock) or a pipelined complex architecture would result in much faster code execution than my current design.

So, why did I choose this approach?

The reason is simple: having fun. My objective is to make writing assembly code for my computer as fun as it can possibly be. On a homebrew CPU speed is not the most important factor.

- Compared with hardware-efficient ISAs like the 6502 or PIC, instructions for MIPS and RISC-V are much more intuitive, powerful and fun to work with. That's why I took inspiration from them, even if this meant I had to use more chips.
- If you need an integer greater than 255, writing several instructions in order to do a dword operation just isn't as fun to program as with native 16 bit instructions.
- Having pipelining and better I/O would be great, but it doesn't affect programming that much.
- Registers in memory may be very efficient, but RAM is a black box and you can't wire LEDs to it. When I build a homebrew CPU I want to be able to see *all* my registers at the same time (however, I did end up implementing some Intel-style x86 addressing modes for completeness).

Why do you use Arduinos in a homebrew CPU?

I understand why some people are against using microcontrollers in homebrew computers. After all, a single Arduino Nano is more powerful than the entire CPU.

However, everyone can decide what they want to allow in their design. I decided that peripherals are not part of the CPU and therefore are allowed to use microcontrollers. After all, even if I managed to implement VGA and PS/2 controllers with 74 series chips, the screen and keyboard themselves have microcontrollers inside, so the design still wouldn't be completely "microcontroller free".

Closing words

I hope that reading this documentation has given you ideas for your own builds, just as reading from others was what inspired me to create my own. Remember that the aim of a homebrew computer isn't to create the perfect and most efficient design, just one that is new and unique, and most importantly one that you enjoy and feel proud of.

If you have questions or any suggestion, you can send me a message on Reddit or an email.

Want more homebrew CPUs? Here are some links (download the PDF to be able to click them):

- MUPS/16 CPU
- James Sharman - Making an 8 Bit pipelined CPU
- Slu4 - Minimalistic Breadboard 8-Bit CPU
- nand2tetris: Part 1, Part 2