CESC16 Detailed Instructions

This document contains an in-depth description of all the machine instructions supported by the CESC16 computer, as well as the macros provided by the assembler.

`ASSEMBLER MNEMONIC`	`Machine code (binary)`	`Pseudocode (C-style)`

C-style Pseudocode notation:

- `ALU(A, B)` ALU operation between A (first operand) and B (second operand)
- ⚐ Flags get updated according to ALU result
- `FLAGS` Direct access to the flags (status) register
- `RAM[Addr]` Access RAM at a given address (Addr)
- `ROM[Addr][W]` Access a word W (1=Upper, 0=Lower) of ROM at a given address (Addr)
- `memSpace` Controls from which memory space (ROM or RAM) instructions will be fetched
- `push(A)` Push a given register A to the stack. It's the same as `RAM[--sp]=A`
- `A=pop()` Pop a given register A from the stack. It's the same as `A=RAM[sp++]`

Macros notation:

- `[mode]` Address in any of the 4 addressing modes:
    - Direct: `[Addr]`
    - Indirect: `[rB]`
    - Register + immediate index: `[rA+Imm16]`
    - Register + register index: `[rA+rB]`
- `OPERAND` Either a register (`rB`), immediate (`Imm`), or memory address (`[mode]`)
- The mnemonic on the left side of the arrow gets replaced by the instruction(s) on the right side of the arrow: `MACRO → Translated Instructions`

```
ALU Operations:
```

Register operand:

`ALU rD, rA, rB`	`00000FFFDDDDAAAA` `XXXXXXXXXXXXBBBB`	`rD = ALU(rA, rB)` ⚐

Immediate operand:

`ALU rD, rA, Imm16`	`00001FFFDDDDAAAA` `IIIIIIIIIIIIIIII`	`rD = ALU(rA, Imm16)` ⚐

Direct addressing:

`ALU rD, rA, [Addr16]`	`01000FFFDDDDAAAA` `@@@@@@@@@@@@@@@@`	`rD = ALU(rA, RAM[Addr16])` ⚐

Indirect addressing:

`ALU rD, rA, [rB]`	`01001FFFDDDDAAAA` `XXXXXXXXXXXXBBBB`	`rD = ALU(rA, RAM[rB])` ⚐

Reg+Imm indexed addressing:

`ALU rD, [rA+Imm16]`	`01010FFFDDDDAAAA` `IIIIIIIIIIIIIIII`	`rD = ALU(rD, RAM[rA+Imm16])` ⚐

Reg+Reg indexed addressing:

`ALU rD, [rA+rB]`	`01011FFFDDDDAAAA` `XXXXXXXXXXXXBBBB`	`rD = ALU(rD, RAM[rA+rB])` ⚐

Operations on each clock cycle (Register and Immediate):

Fetch instruction + 1st operand	Fetch argument (2nd operand)	Perform ALU operation and store result in register file

Operations on each clock cycle (Direct and Indirect):

Fetch instruction + 1st operand	Fetch argument and compute address	Fetch 2nd operand from memory	Perform ALU op. and store result in regfile

Operations on each clock cycle (Indexed):

Fetch instruction	Fetch argument, compute address	Fetch 1st operand (rD) from regfile	Fetch 2nd operand from memory	Perform ALU op, store result in reg.

Description:

Performs an ALU operation as indicated by the 3 Funct bits, using the contents of rA as first operand (except in the indexed modes) and either the contents of rB, a 16 bit immediate or the contents of a memory address as second operand. The result of the operation is stored in rD and the flags are updated accordingly. See table in main documentation for ALU operations, mnemonics and descriptions.

Remarks about ALU operations:

- Carry and oVerflow flags are undefined after all operations except `add`, `sub`, `addc` and `subb`.
- The `mov` instruction doesn't require the first operand (rA), doesn't update the flags (see `movf` macro for this purpose) and takes only 2 clock cycles (Register and Immediate) or 3 clock cycles (Direct, Indirect and Indexed).
- When an indexed addressing mode is used, rA is needed for computing the address. Therefore, rD is used both as the first argument and the destination register.

Examples:

`mov t0, t1`	The value stored in `t1` gets copied into `t0`. The value at `t1` and the flags are unchanged.
`mov t0, 0x1234`	The value stored in `t0` becomes 0x1234. Flags are preserved.
`and t0, t1, t2`	Perform a logic AND between the contents of `t1` and `t2`. Store result into `t0`. The values at `t1` and `t2` remain unchanged.
`sub t0, t1, [123]`	Fetch the data stored at address 123 (0x7B) and subtract it from the data stored at `t1`. Store the result in `t0` (operands remain unchanged).
`addc t0, t1, [t2]`	Fetch the data pointed by register `t2` and add them to the contents of `t1`. Add 1 to result if Carry bit is set. Store result in `t0` (operands are unchanged).
`mov a0, [t0+s4]`	Add the contents of `t0` and `s4` to get a memory address, then load the data stored at that address into `a0`.
`add a0, [t0+10]`	Add 10 to the contents of `t0` to get a memory address, then add the data stored at that address to the contents of `a0`. Store the result in `a0`.

Macros:

Negate register (bitwise NOT):	`not rD, rA`	→	`xor rD, rA, 0xFFFF`
Shift Left with Carry (1 bit):	`sllc rD, rA`	→	`addc rD, rA, rA`
Move and update Flags*:	`movf rD, OPERAND`	→	`add rD, zero, OPERAND`
Compare register to operand*:	`cmp rA, OPERAND`	→	`sub zero, rA, OPERAND`
Test masked register*:	`mask rA, OPERAND`	→	`and zero, rA, OPERAND`
Test register (or memory):	`test OPERAND`	→	`movf zero, OPERAND`
Clear flags:	`clrf`	→	`movf zero, 0x0001`
No operation:	`nop`	→	`mov zero, zero`

- There are many alternative expansions for `nop`. This one is encoded as all zeros (0x0000).

* It's not possible to implement an indexed version of `movf`, `cmp` or `mask`, since rD also acts as the first operand. However, it is possible to implement a version of `test`.

ALU Operations (destination in memory):

Direct addressing

`ALU [Addr16], rA`	`01100FFFXXXXAAAA` `@@@@@@@@@@@@@@@@`	`RAM[Addr16] =` `ALU(RAM[Addr16], rA)` ⚐

Indirect addressing:

`ALU [rA], rB`	`01101FFFXXXXAAAA` `XXXXXXXXXXXXBBBB`	`RAM[rA] =` `ALU(RAM[rA], rB)` ⚐

Reg+Imm indexed addressing:

`ALU [rA+Imm16], rB`	`01110FFFBBBBAAAA` `IIIIIIIIIIIIIIII`	`RAM[rA+Imm16] =` `ALU(RAM[rA+Imm16], rB)` ⚐

Reg+Reg indexed addressing:

`ALU [rA+rC], rB`	`01111FFFBBBBAAAA` `XXXXXXXXXXXXCCCC`	`RAM[rA+rC] =` `ALU(RAM[rA+rC], rB)` ⚐

Operations on each clock cycle (Direct and Indirect):

Fetch instruction	Fetch argument and compute address	Fetch 1st operand from memory	Perform ALU op. and store result in memory

Operations on each clock cycle (Indexed):

Fetch instruction	Fetch argument, compute address	Fetch 2nd operand (rB) from regfile	Fetch 1st operand from memory	Perform ALU op, store in memory

Description:

Performs an ALU operation as indicated by the 3 Funct bits, using the contents of a memory address (direct, indirect or indexed addressing) as first operand and a register as second operand.

The result of the operation is stored in the same address as the first operand and the flags are updated accordingly. See table in main documentation for ALU operations, mnemonics and descriptions.

Remarks about memory ALU operations:

- Carry and oVerflow flags are undefined after all operations except `add`, `sub`, `addc` and `subb`.
- The decoded memory address is used for both first operand and destination. The second operand must be a register (no Memory-Memory or Memory-Immediate operations). If those restrictions can't be met, considering loading the needed value to a temporary register first.
- The `mov` instruction doesn't update the flags (see `movf` macro for this purpose) and takes only 3 clock cycles (Direct and Indirect) or 4 clock cycles (Indexed modes).

Examples:

`mov [0x1234], zero`	The memory contents at address 0x1234 become 0x0000 (the contents of `zero` get stored in memory). Flags are preserved.
`mov [s0+12], a1`	Store the contents of `a1` into memory. The memory address consists of the contents of `s0`, plus an offset of 12. Flags are preserved.
`xor [6], s1`	Perform a logic XOR between the data at memory address 6 and the contents stored in `s1`. Store the result into address 6 (`s1` remains unchanged).
`add [sp], a1`	Increment the top of the stack by the amount stored in `a1`. The contents of `a1` and `sp` remain unchanged.
`subb [t2+1], t1`	Fetch the data pointed by register `t2` (plus an offset of 1) and subtract the contents of `t1` from it. Subtract 1 to result if Borrow bit is set. Store the result in memory (at the address pointed by `t2` plus 1).
`or [a0+a1], a2`	Add the contents of `a0` and `a1` to get a memory address. Then perform a logic OR between `a2` and the data stored at the memory address. Store the result in memory (at the same memory address).

Shifts:

Shift Left Logical:

`sll rD, rA, Imm4`	`0001IIIIDDDDAAAA` `XXXXXXXXXXXXXXXX`	`rD = rA<<Imm4` ⚐

Shift Right Logical:

`srl rD, rA, Imm4`	`0010IIIIDDDDAAAA` `XXXXXXXXXXXXXXXX`	`rD = rA>>Imm4` ⚐ `(unsigned)`

Shift Right Arithmetic:

`sra rD, rA, Imm4`	`0011IIIIDDDDAAAA` `XXXXXXXXXXXXXXXX`	`rD = rA>>Imm4` ⚐ `(signed)`

Operations on each clock cycle:

Fetch instruction + operand (rA)	Shift 1 position	Shift 1 position	…	Shift 1 position and store result

Description:

The contents of rA get shifted (left or right) as many bits as indicated by Imm4.

- `sll`: bits get shifted to the left and filled with zeros. oVerflow flag is undefined.
- `srl`: bits get shifted to the right and filled with zeros. Carry and oVerflow flags are undefined.
- `sra`: bits get shifted to the right and the sign is extended. Carry and oVerflow flags are undefined.

Flags are updated and the result is stored in rD.

Remarks about shifts:

- Memory contents can't be shifted directly and must be copied to/from a temporary register.
- Bit shifts are the only instructions with variable clock durations. Each shifted bit takes 1 clock cycle, plus 1 extra clock for fetching.
- The ISA allows shifting 0 bits but, since it has no practical use, it can be considered an illegal instruction. The computer will interpret a shift of 0 bits as a NOP.

```
Swap register with memory:
```

`swap rD, [rA+Imm16]`	`10000001DDDDAAAA`
	`IIIIIIIIIIIIIIII`
		`temp = rD;`
		`rD = RAM[rA+Imm16];`
		`RAM[rA+Imm16] = temp`

Operations on each clock cycle:

Fetch instruction	Fetch offset and compute address	Fetch rB (same as rD)	Read data from memory	Write data to memory

Description:

Swaps the contents of rD and the contents stored at an address (with offset) of data memory (RAM). An indexed `mov` is performed simultaneously to and from rD.

Macros:

Direct addressing:	`swap rD, [Addr16]`	→	`swap rD, [zero+Addr16]`
Indirect addressing:	`swap rD, [rA]`	→	`swap rD, [rA+0]`

`Peek program memory:`

`peek rD, [rA+Imm16], W`	`1000001WDDDDAAAA` `IIIIIIIIIIIIIIII`	`rD = ROM[rA+Imm16][W]`

Operations on each clock cycle:

Fetch instruction	Fetch offset and compute address	Read program memory

Description:

Loads into rD the contents from program memory (ROM) at the address contained by rA (plus an offset).

Since the program memory is 32 bits wide, `W` indicates which 16-bit word will be fetched:

- W=1: Most significant bits get fetched (instruction opcode).
- W=0: Least significant bits get fetched (instruction argument).

The assembler uses big endian encoding. Therefore, when `peek` is used to load 16-bit constants, the most significant bits (W=1) correspond to the first word (lower address) and the least significant bits (W=0) correspond to the second word (higher address).

Macros:

Peek upper bits:	`peek rD, [rA+Imm16], Up`	→	`peek rD, [rA+Imm16], 1`
Peek lower bits:	`peek rD, [rA+Imm16], Low`	→	`peek rD, [rA+Imm16], 0`
Peek opcode:	`peek rD, [rA+Imm16], Op`	→	`peek rD, [rA+Imm16], 1`
Peek argument:	`peek rD, [rA+Imm16], Arg`	→	`peek rD, [rA+Imm16], 0`

Stack Push and Pop:

Push register to Stack:

`push rB`	`10000100XXXX0001` `XXXXXXXXXXXXBBBB`	`RAM[--sp] = rB` `(push(rB))`

Push immediate to Stack:

`push Imm16`	`10000101XXXX0001` `IIIIIIIIIIIIIIII`	`push(Imm16)`

Push flags to Stack:

`pushf`	`10000110XXXX0001` `XXXXXXXXXXXXXXXX`	`push(FLAGS)`

Pop register from Stack:

`pop rD`	`10000111DDDD0001` `XXXXXXXXXXXXXXXX`	`rD = RAM[sp++]` `(rD = pop())`

Pop flags from Stack:

`popf`	`10001000XXXX0001` `XXXXXXXXXXXXXXXX`	`FLAGS = pop()`

Operations on each clock cycle:

Fetch instruction	Fetch and update Stack Pointer Only in `push`: Fetch argument	Read/Write data memory

Description:

- `push` pushes the contents of a register (or an immediate value) into the stack: `sp` is decremented by 1 and the data is stored at the new address pointed by `sp`.
- `pop` pops the top of the stack into rD: loads the contents pointed by `sp` into rD and increments `sp`.
- `pushf` and `popf` work the same way, but they store and load the flags (status register). This isn't usually needed for regular subroutines, but an interrupt handler must use them to preserve the status of the main program that got interrupted.

Warning: Addressing modes can also be used to access local variables in the stack (by using `sp` as the base address), but you shouldn't use both methods at once (otherwise, the offsets will keep changing).

The safe way of storing variables in the stack is:

1. Store required safe registers (using `push`)
2. Allocate space in the stack (using `sub sp, sp, N`)
3. Subroutine body (`sp` never changes, so the offsets stay constant)
4. Deallocate space in the stack (using `add sp, sp, N`)
5. Restore context of caller (using `pop`) before returning

SP SP+N-1	Local variables
SP+N	Context of caller
	ret address

Remarks about interrupt handlers: An interrupt handler *must* push the flags and all registers it's going to use (not just the safe registers). However, all of this is already done by the OS before handing over control to the user's interrupt handler, which can treat the registers and flags as if it was a regular subroutine (that is, it only needs to push and pop *safe* registers).

Conditional Jumps:

Jump to register:

JMP rA	1100FFFFXXXXAAAA XXXXXXXXXXXXXXXX	if(condition) PC = rA

Jump to immediate address:

JMP Addr16	1101FFFFXXXXXXXX @@@@@@@@@@@@@@@@	if(condition) PC = Addr16

Operations on each clock cycle:

Fetch instruction	Check flags. If condition is true, load new address into PC

Description:

Checks the condition indicated by the 3 Funct bits, then jumps to an immediate address (or the address stored in rA) only if the condition is true.

Therefore, the next executed instruction is pointed by:

- `Addr16` or `rA`, if the jump condition is met. The `jmp` instruction is always performed.
- `PC+1`, if the jump condition is not met (or `PC+2` if running from RAM).

The jump condition is checked using the flags, which depend on the last ALU operation.

Conditional jumps (and macros) can be separated in 2 groups:

- Check result of last operation: `jz`, `jnz`, `jc`, `jnc`, `jo`, `jno`, `js`, `jns`
- Compare 2 integers (must be executed right after a `cmp` instruction):
  `je`, `ja`, `jae`, `jb`, `jbe`, `jl`, `jle`, `jg`, `jge`

  And their negations:
  `jne`, `jna`, `jnae`, `jnb`, `jnbe`, `jnl`, `jnle`, `jng`, `jnge`

Most of these mnemonics share opcodes since they perform the same action. See *Jump Conditions* table in DOCS/CESC16.pdf for all the alternative names for each real instruction.

Macros:

Skip N instructions:	JMP skip(N)	→	JMP *pc* + N + 1
Skip N instructions (in RAM):	JMP skip(N)	→	JMP *pc* + 2*(N + 1)

Call subroutine:

Call subroutine in same memory space (address in register):

call rA	11100000XXXXAAAA XXXXXXXXXXXXXXXX	push(PC+N); PC = rA

Call subroutine in same memory space (immediate address):

call Addr16	11100001XXXX0001 @@@@@@@@@@@@@@@@	push(PC+N); PC = Addr16

Call subroutine in ROM (address in register):

syscall rB	11100010XXXX0001 XXXXXXXXXXXXBBBB	push(PC+N); PC = rA; memSpace = ROM

Call subroutine in ROM (immediate address):

syscall Addr16	11100011XXXX0001 @@@@@@@@@@@@@@@@	push(PC+N); PC = Addr16; memSpace = ROM

Call subroutine in RAM (address in register):

enter rB	11100100XXXX0001 XXXXXXXXXXXXBBBB	push(PC+N); PC = rA; memSpace = RAM

Call subroutine in RAM (immediate address):

enter Addr16	11100101XXXX0001 @@@@@@@@@@@@@@@@	push(PC+N); PC = Addr16; memSpace = RAM

Operations on each clock cycle:

Fetch instruction	Fetch and update SP Fetch new address	Store PC in stack	Load address into PC, update memory space

Description:

Those instructions push the address of the next instruction to the stack (PC+1 if running from ROM, PC+2 if running from RAM) before jumping unconditionally to an address. Arbitrary depths of subroutine calls are allowed (as well as recursion).

Instructions can be fetched from ROM or RAM. This family of instructions allows jumping between them:

- `call`: stays in the current memory space (used for regular subroutines)
- `syscall`: calls a subroutine stored in ROM (used for system calls)
- `enter`: calls a subroutine stored in RAM (used for entering user programs)

<table>
<tr><th>Memory space BEFORE</th><th>Instruction</th><th>Memory space AFTER</th><th>Gets pushed to stack</th></tr>
<tr><td rowspan="3">ROM</td><td>call</td><td>ROM</td><td rowspan="3">PC+1</td></tr>
<tr><td>syscall</td><td>ROM</td></tr>
<tr><td>enter</td><td>RAM</td></tr>
<tr><td rowspan="3">RAM</td><td>call</td><td>RAM</td><td rowspan="3">PC+2</td></tr>
<tr><td>syscall</td><td>ROM</td></tr>
<tr><td>enter</td><td>RAM</td></tr>
</table>

`Return from subroutine:`

Return from `call`:

`ret`	`11100110XXXX0001` `XXXXXXXXXXXXXXXX`	`PC = pop()`

Return from `syscall`:

`sysret`	`11100111XXXX0001` `XXXXXXXXXXXXXXXX`	`PC = pop()` `memSpace = RAM`

Return from `enter`:

`exit`	`11101000XXXX0001` `XXXXXXXXXXXXXXXX`	`PC = pop()` `memSpace = ROM`

Operations on each clock cycle:

Fetch instruction	Fetch and update Stack Pointer	Pop new address from stack to PC, update memory space

Description:

Those instructions pop the top of the stack and jump unconditionally to that address: control is returned to the routine that performed the call instruction.

Warning: Make sure the subroutine has freed all the memory it had allocated in the stack before using the *return* instruction (otherwise `sp` won't be pointing at the correct return address).

The *return* instructions are companions of a type of *call* instruction:

- `ret` returns from routines that were called using `call`: stays in the current memory space.
- `sysret` returns from routines that were called from RAM (using `syscall`): always jumps to RAM.
- `exit` exits user programs that were called from ROM (using `enter`): always jumps to ROM.

For more information, read the "Memory Map" section in DOCS/CESC16.pdf

Memory space BEFORE	Instruction	Memory space AFTER	Intended use: returning from
`ROM`	`ret`	**`ROM`**	**`call, syscall*`**
	`sysret`	**`RAM`**	**`syscall**`**
	`exit`	**`ROM`**	`[use ret instead]`
`RAM`	`ret`	**`RAM`**	**`call`**
	`sysret`	**`RAM`**	`[use ret instead]`
	`exit`	**`ROM`**	**`enter`**

* OS routines can be called from ROM using `call`, but it's recommended to use `syscall` instead.

** OS routines use `ret` instead of `sysret` (even though they are called using `syscall`), because they don't know if they have been called from ROM or RAM, so they will assume it's been ROM. When calling OS routines from RAM, the `CALL_GATE` routine must be used. Read the "Operating System" section in DOCS/CESC16.pdf for more information.