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
- `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)
- `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:

- `OPERAND` Either a register `rB`, immediate `Imm`, or memory address: `[Addr]` or `[rB]`
- 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`

DETAILED INSTRUCTIONS:

`ALU Operations:`

Register operand:

ASSEMBLER MNEMONIC	Machine code (binary)	Pseudocode (C-style)
`ALU rD, rA, rB`	`X0000FFFDDDDAAAA` `XXXXXXXXXXXXBBBB`	`rD = ALU(rA, rB)` ⚐

Immediate operand:

ASSEMBLER MNEMONIC	Machine code (binary)	Pseudocode (C-style)
`ALU rD, rA, Imm16`	`X0001FFFDDDDAAAA` `IIIIIIIIIIIIIIII`	`rD = ALU(rA, Imm16)` ⚐

Direct addressing:

ASSEMBLER MNEMONIC	Machine code (binary)	Pseudocode (C-style)
`ALU rD, rA, [Addr16]`	`X0010FFFDDDDAAAA` `@@@@@@@@@@@@@@@@`	`rD = ALU(rA, RAM[Addr16])` ⚐

Indirect addressing:

ASSEMBLER MNEMONIC	Machine code (binary)	Pseudocode (C-style)
`ALU rD, rA, [rB]`	`X0011FFFDDDDAAAA` `XXXXXXXXXXXXBBBB`	`rD = ALU(rA, RAM[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 reg. file

Description:
Performs an ALU operation as indicated by the 3 Funct bits, using the contents of rA as first operand 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 and Indirect).
- The `mov` instruction with Direct and Indirect addessing can be used as a substitute for the `lw` instruction. Using the dedicated `lw/lb` instructions is recommended, since they use the superior indexed addressing and `lb` allows loading single bytes. However, if indexed addressing and byte addressing aren't needed, using `mov` may improve code readability.

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 contents stored at address 123 (0x7B) and subtract them from the contents stored at `t1`. Store the result in `t0` (operands remain unchanged).
`addc t0, t1, [t2]`	Fetch the contents 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).

Macros:

Move and update Flags:	`movf rD, OPERAND`	→	`add rD, zero, OPERAND`
Negate register (bitwise NOT):	`not rD, rA`	→	`xor rD, rA, 0xFFFF`
Bitwise NAND, NOR and XNOR: `nand/nor/xnor rD, rA, OPERAND`		→	`and/or/xor rD, rA, OPERAND` `not rD, rD`
Shift Left with Carry (1 bit):	`sllc rD, rA`	→	`addc rD, rA, rA`
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).

ALU Operations (destination in memory):

Direct addressing

`ALU [Addr16], rA`	`X0100FFFXXXXAAAA` `@@@@@@@@@@@@@@@@`	`RAM[Addr16] = ALU(RAM[Addr16], rA)` ⚐

Indirect addressing:

`ALU [rA], rB`	`X0101FFFXXXXAAAA` `XXXXXXXXXXXXBBBB`	`RAM[rA] = ALU(RAM[rA], rB)` ⚐

Operations on each clock cycle:

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

Description:
Performs an ALU operation as indicated by the 3 Funct bits, using the contents of a memory address (direct or indirect 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.
- The `mov` instruction can be used as a substitute for the `sw` instruction. Using the dedicated `sw` instruction is recommended, since it uses the superior indexed addressing. However, if indexed addressing isn't needed, using `mov` is 1 cycle faster and may improve code readability.

Examples:

`mov [123], a1`	Store the contents of `a1` into memory location 123 (0x7B). The value at `a1` is unchanged. Flags are preserved.
`mov [t0], zero`	The memory contents that are being pointed by `t0` become 0x0000 (the contents of `zero` get copied). Flags are preserved.
`xor [6], s1`	Perform a logic XOR between the contents at memory address 6 and the contents stored in `s1`. Store 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], t1`	Fetch the contents 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).

Shifts:

Shift Left Logical:

`sll rD, rA, Imm4`	X011IIIIDDDDAAAA XXXXXXXXXXXXXXXX	rD = rA<<Imm4 ⚐

Shift Right Logical:

`srl rD, rA, Imm4`	X100IIIIDDDDAAAA XXXXXXXXXXXXXXXX	rD = rA>>Imm4 ⚐ (unsigned)

Shift Right Arithmetic:

`sra rD, rA, Imm4`	X101IIIIDDDDAAAA 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.
- `srl`: bits get shifted to the right and filled with zeros.
- `sra`: bits get shifted to the right and the sign is extended.

Flags are updated (Carry and oVerflow flags are undefined) 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.

Load/Store data from/to memory:

Load Word:

`lw rD, Imm16(rA)`	`X1100000DDDDAAAA` `IIIIIIIIIIIIIIII`	`rD = RAM[rA+Imm16]`

Load Byte:

`lb rD, Imm16(rA)`	`X1100001DDDDAAAA` `IIIIIIIIIIIIIIII`	`rD = byte(RAM[rA+Imm16])`

Store Word:

`sw rB, Imm16(rA)`	`X1100010BBBBAAAA` `IIIIIIIIIIIIIIII`	`RAM[rA+Imm16] = rB`

Operations on each clock cycle:

Fetch instruction	Fetch offset and compute address	Only in `sw`: Fetch `rB`	Read or Write data memory

Description:
The contents of rA are added to an immediate offset to get a memory address. Then, the contents of 1 register is transfered to or from data memory (RAM):

- `lw`: the 16-bit word stored in memory at the given address gets copied into `rD` (memory contents are unchanged).
- `lb`: the same as a `lw`, but only the lower 8 bits are fetched and the sign is extended.
- `sw`: the 16-bit word stored in `rB` gets copied into memory at the given address (`rB` is unchanged).

Remarks about load/store:

- The constant `zero` register can be used as rA in order to access an absolute address.
- A `lbu` (Load Byte Unsigned) macro can be implemented with a mask (set upper 8 bits to 0).
- Memory access is not Byte-Oriented and Data Memory is 16 bit wide:
  - There is no way to access only the upper 8 bits of the word at a memory address (other than using `lw` and shifting 8 positions with `srl`).
  - It doesn't matter if we want to store a byte or a 16 bit word, both take the same amount of space in memory (1 word).
  - A `sb` (Store Byte) instruction isn't needed, the only thing that matters is how the data is interpreted when the Load is performed (choosing between `lw`, `lb` and `lbu`). However, `sb` can be used as an alias to `sw` to improve code clarity.

Macros:

Load Byte Unsigned:	`lbu rD, Imm16(rA)`	→	`lb rD, Imm16(rA)` `and rD, rD, 0x00FF`
Store Byte:	`sb rB, Imm16(rA)`	→	`sw rB, Imm16(rA)`
Alternative `mov` notation:	`mov rD, [rA+Imm16]`	→	`lw rD, Imm16(rA)`
	`mov [rA+Imm16], rB`	→	`sw rB, Imm16(rA)`

Swap register with memory:

`swap rD, Imm16(rA)`	`X1100011DDDDAAAA`
	`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 memory	Write data memory

Description:
Swaps the contents of rD and the contents stored at an address (with offset) of data memory (RAM). A `lw` and a `sw` are performed simultaneously to and from rD.

Macros:
Alternative notation: `swap rD, [rA+Imm16]` → `swap rD, Imm16(rA)`

Peek program memory:

`peek rD, Imm16(rA), W`	`X110010WDDDDAAAA` `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. Since the program memory is 32 bits wide, Imm1 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).

Stack Push and Pop:

Push register to Stack:

`push rB`	`X1100110XXXX0001` `XXXXXXXXXXXXBBBB`	`RAM[--sp] = rB` `(push(rB))`

Pop register from Stack:

`pop rD`	`X1100111DDDD0001` `XXXXXXXXXXXXXXXX`	`rD = RAM[sp++]` `(rD = pop())`

Push flags to Stack:

`pushf`	`X1111010XXXX0001` `XXXXXXXXXXXXXXXX`	`push(readFlags())`

Pop flags from Stack:

`popf`	`X1111011XXXX0001` `XXXXXXXXXXXXXXXX`	`writeFlags(pop())`

Operations on each clock cycle:

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

Description:

- `push` pushes the contents of rA into the stack: `sp` is decremented by 1 and rA 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 then `sp` is incremented by 1.
- `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 it's indispensable to use this instructions in an interrupt handler.

Warning: `lw/sw` instructions can also be used to access the stack without the limitations of `push/pop` (by using `sp` as address), but you shouldn't use both methods at once: since `push/pop` affect `sp`, the offset you have to use in `lw/sw` to access each variable will change. Unless you know what you are doing, that will most likely lead to bugs.

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 immediate address:

`JMP Addr16`	`X1101FFFXXXXXXXX` `@@@@@@@@@@@@@@@@`	`if(condition) PC = Addr16`

Jump to register:

`JMP rA`	`X1110FFFXXXXAAAA` `XXXXXXXXXXXXXXXX`	`if(condition) PC = rA`

Operations on each clock cycle:

Fetch instruction	Check flags. Only 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 addres 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.

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, jb, jnb`
- Compare 2 integers (*must* be executed right after a `cmp` instruction):
  `jeq, jne, jlt, jle, jltu, jleu`

Macros:

Jump if Borrow:	`jb` *`addr`*	→	`jnc` *`addr`*
Jump if Not Borrow:	`jnb` *`addr`*	→	`jc` *`addr`*
Jump if Equal:	`jeq` *`addr`*	→	`jz` *`addr`*
Jump if Not Equal:	`jne` *`addr`*	→	`jnz` *`addr`*
Jump if Less Than (Unsigned):	`jltu` *`addr`*	→	`jnc` *`addr`*
Skip N instructions:	`JMP skip(N)`	→	`JMP` *`pc`* `+ N + 1`

`Call subroutine:`

`call Addr16`	`X1111000XXXX0001` `@@@@@@@@@@@@@@@@`	`push(PC+1); PC = Addr16`

Operations on each clock cycle:

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

Description:
The `call` instruction pushes the address of the next instruction to the stack before jumping unconditionally to an address.
Since the PC is pushed to the stack in data memory, arbitrary depths of subroutine calls are allowed (as well as recursion).

Macros:
System Call*: `syscall Addr16` → `call Addr16`

* See “Operating system” section in main documentation

`Return from subroutine:`

`ret`	`X1111001XXXX0001` `XXXXXXXXXXXXXXXX`	`PC = pop()`

Operations on each clock cycle:

Fetch instruction	Fetch and update Stack Pointer	Pop new address from stack and load it into PC

Description:
The `ret` instruction pops the top of the stack and jumps unconditionally to that address.

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