CdM-8 instruction set:

Level 3 and Level 3½ mnemonics, descriptions, machine code, example usage

Instructions for copying bit-strings from one place to another

1. Loading a bit-string from a memory cell into a register

`ldi rn,const` *const* → r*n* Flags unchanged

Load the **immediate** single-byte data item *const* into r*n*.
The bit-string representing *const* is copied into r*n*.

2-byte machine code instruction
Opcode: 110100
Operand1: 2-bit register number (00, 01, 10 or 11)
Operand2: immediate value (second byte of instruction)

Example usage
Instruction: ldi r1, 0x6E
In binary: 11010001 01101110
Before: N/A
After: r1 contains 01101110

The assembler creates a 2-byte instruction containing the bit-string representing `ldi` r*n* **immediately followed** by the bit-string representing *const*. The data item *const* is actually fetched from memory at run-time.[1]

`ld rn,rm` *r*r*n* → r*m* Flags unchanged

Load a byte into r*m* from the memory cell addressed by r*n*.
(*r*n* is the memory cell *pointed to* by r*n*. The bit-string read from this memory cell is copied into r*m*.)

1-byte machine code instruction
Opcode: 1011
Operand1: 2-bit register number (00, 01, 10 or 11)
Operand2: 2-bit register number (00, 01, 10 or 11)

Example usage
Instruction: ld r0, r3
In binary: 10110011
Before: r0 contains 01111001
After: r0 and mem[01111001] unchanged, r3 contains a copy of mem[01111001]

2. Storing a bit-string to a memory cell from a register

`st rn,rm` r*m* → *r*n* Flags unchanged

Store the byte in r*m* to the memory cell addressed by r*n*.
(*r*n* is the memory cell *pointed to* by r*n*. This cell is over-written by the bit-string copied from r*m*.)

1-byte machine code instruction
Opcode: 1010
Operand1: 2-bit register number (00, 01, 10 or 11)
Operand2: 2-bit register number (00, 01, 10 or 11)

Example usage
Instruction: st r1, r0

[1] Care must be taken not to over-write instructions stored in main memory whilst a program is running. This is a sure-fire way to introduce bugs that are hard to detect, harder to diagnose, and even harder to correct.

In binary: `10100100`
Before: r1 contains 00000110
After: r1 and r0 unchanged, mem[00000110] contains a copy of r0

3. Copying bit-strings to and from the stack

`push rn` `((SP-1)→SP) then (rn → *SP)` Flags unchanged

Push the byte in `rn` onto the stack.[2]
SP is the *stack pointer* register. This is decremented, then used to point at a memory cell which is over-written by the bit-string copied from `rn`. (*SP is the memory cell pointed to by SP)

1-byte machine code instruction
Opcode: `110000`
Operand1: 2-bit register number (`00, 01, 10 or 11`)
Operand2: None

Example usage
Instruction: `push r2`
In binary: `11000010`
Before: SP contains 00000000
After: r2 unchanged, SP contains 11111111, mem[11111111] contains a copy of r2

`pop rn` `(*SP → rn) then ((SP+1)→SP)` Flags unchanged

Pop a byte off the stack into `rn`.[3]
SP is the *stack pointer* register. This is used to point at a memory cell which is copied into `rn`, then incremented. (*SP is the memory cell pointed to by SP)

1-byte machine code instruction
Opcode: `110001`
Operand1: 2-bit register number (`00, 01, 10 or 11`)
Operand2: None

Example usage
Instruction: `pop r3`
In binary: `11000111`
Before: SP contains 11111111
After: mem[11111111] unchanged, SP contains 00000000, r3 contains a copy of mem[11111111]

4. Copying bit-strings between registers

`move rn,rm` `rm → rn` `Z,N` reflect result `C,V` become 0

Move `rn` **to** `rm`
Copies the content of `rn` to `rm`. `C` and `V` are cleared. `N` and `Z` are based on the modified `rn`.

[2] The stack is a data structure of variable size made up of memory cells. The first byte of the stack is held at memory location `0xFF`, and the stack grows *down* memory from there. It is managed using a register called the Stack Pointer, which contains the address of the most recent byte stored on the stack. It is the responsibility of the programmer to manage the stack properly. Each ***push*** instruction makes the stack **grow** in size by 1 byte, causing it to get closer and closer to those locations where program instructions and initial data are stored, so too many `push`es without a `pop` can cause a program to be corrupted by being over-written by the stack.

[3] Remember: it is the programmer's responsibility to manage the stack properly. Each ***pop*** instruction makes the stack **shrink** in size by 1 byte. A program that uses more `pop`s than `push`es will treat the bytes at location `0x00` and above as part of the stack. These cells hold program instructions and initial data, so subsequent `push`es will corrupt the program.

2-byte machine code instruction
Opcode: `0000`
Operand1: 2-bit register number (`00, 01, 10 or 11`)
Operand2: 2-bit register number (`00, 01, 10 or 11`)

`ldsp rn`	`SP` → `rn`	`Z,N` reflect result	`C,V` become 0

Load Stack Pointer into `rn`
Copies the content of SP to `rn`. `C` and `V` are cleared. `N` and `Z` are based on the modified `rn`.

1-byte machine code instruction
Opcode: `110011`
Operand1: 2-bit register number (`00, 01, 10 or 11`)
Operand2: None

`stsp rn`	`rn` → `SP`	`Z,N` reflect result	`C,V` become 0

Store `rn` to Stack Pointer[4]
Copies the content of `rn` to SP. `C` and `V` are cleared. `N` and `Z` are based on `rn`.

1-byte machine code instruction
Opcode: `110010`
Operand1: 2-bit register number (`00, 01, 10 or 11`)
Operand2: None

[4] This instruction must be used with great care, as it *changes* the content of the stack pointer.

Instructions for manipulating bit-strings within registers

1. Arithmetic operations

Like all other CdM-8 Platform 3 operations these may be used on any bit-strings. However, they are *named* for the results they give when those bit-strings represent numbers.
The flags in the Processor Status (PS) register are affected by each of these operations. C and V are modified in the course of *calculating* the result, whereas Z and N depend solely on the result bit-string: Z is 1 when the result is an all-zeros bit-pattern, and 0 otherwise, N is equal to bit 7 (the *sign bit*) of the result.
Conventionally, C is taken to be the value that is *carried out* from Column 7 of the bit-string, and V tells us whether there has been a *two's complement overflow* (e.g. when the result of adding together two bit-strings representing positive numbers in two's complement form is a bit-string that represents a negative number in two's complement form, such as 01000000 + 01100000 = 10100000).
It is important to remember, however, that the true 'meaning' of each of the status flags depends upon what the bit-strings being manipulated actually represent. For example, it is perfectly possible to apply an *add* operation to a pair of registers containing bit-strings that represent ASCII characters. Neither the resulting bit-string nor the flags would be terribly meaningful under such circumstances, and to interpret V=1 as a two's complement overflow (for example) would be pretty daft.

`add r`n`,r`m `(r`n `+ r`m`) → r`m `C,V,Z,N` reflect result

Add together the bit-strings in `r`n and `r`m, assuming they represent binary numbers.
The result is placed in `r`m.
C is the carry-out from column 7.
V is 1 when $\texttt{r}n_7 = \texttt{r}m_7$ before the operation and $\texttt{r}n_7 \neq \texttt{r}m_7$ afterwards. Otherwise V is 0.

1-byte machine code instruction
Opcode: `0001`
Operand1: 2-bit register number (`00, 01, 10 or 11`)
Operand2: 2-bit register number (`00, 01, 10 or 11`)

`addc r`n`,r`m `(r`n `+ r`m `+ C) → r`m `C,V,Z,N` reflect result

Add together the **C** flag (0 or 1) and the bit-strings in `r`n and `r`m, assuming they represent binary numbers.
The result is placed in `r`m.
Add-with-carry-in is used when performing *byte-sliced addition* on numbers that are represented by bit-strings made up of two or more bytes.
Beforehand the `C` flag holds a *carry-in* value (the carry-out from bit 7 of a lower-order byte), and afterwards its content is the *carry-out* from bit 7 of the addition.
V is 1 when $\texttt{r}n_7 = \texttt{r}m_7$ before the operation and $\texttt{r}n_7 \neq \texttt{r}m_7$ afterwards. Otherwise V is 0.

1-byte machine code instruction
Opcode: `0010`
Operand1: 2-bit register number (`00, 01, 10 or 11`)
Operand2: 2-bit register number (`00, 01, 10 or 11`)

`sub r`n`,r`m `(r`n `− r`m`) → r`m `C,V,Z,N` reflect result

Subtract the byte in `r`m from the byte in `r`n, assuming they represent binary numbers.
The result is placed in `r`m.

1-byte machine code instruction
Opcode: `0011`
Operand1: 2-bit register number (`00, 01, 10 or 11`)
Operand2: 2-bit register number (`00, 01, 10 or 11`)

`cmp r`n`,r`m

`Calculates (r`n − `r`m`)` `C,V,Z,N` reflect result

Compare `r`m with `r`n.
Assume the bytes in `r`n and `r`m represent binary numbers and perform the subtraction (`r`n−`r`m).
Used to modify flags without affecting registers or memory.
The registers `r`n and `r`m remain unchanged by this operation. Any of the four flags may change.

1-byte machine code instruction
Opcode: `0111`
Operand1: 2-bit register number (`00, 01, 10 or 11`)
Operand2: 2-bit register number (`00, 01, 10 or 11`)

`tst r`n `Modifies Z & N flags` `Z,N` reflect result

Test `r`n.
Assume the byte in `r`n represents a binary number and test whether it is zero or negative.
Used to modify flags without changing registers or memory.
The register `r`n remains unchanged by this operation, as do C and V.

This is a Platform 3½ **macro**. The macro-assembler inserts `move r`n`, r`n wherever `tst r`n is requested by the programmer, as it has the same effect.

`neg r`n $(-\texttt{r}n) \rightarrow \texttt{r}n$ `C,V,Z,N` reflect result

Negate `r`n
Replace the contents of `r`n by its 8-bit two's complement.
If `r`n holds the 8-bit two's complement representation of the numerical value x before the operation it will contain the 8-bit two's complement representation of $-x$ afterwards.[5]

1-byte machine code instruction
Opcode: `100001`
Operand1: 2-bit register number (`00, 01, 10 or 11`)
Operand2: None

`inc r`n $(\texttt{r}n + 1) \rightarrow \texttt{r}n$ `C,V,Z,N` reflect result

Increment `r`n
Treats `r`n as a binary number, and adds 1 to it.
The addition 'wraps around', so when `r`n contains 11111111 beforehand it will contain 00000000 afterwards (and the C, V and Z flags will all be set to 1). The V flag will also be set to 1 by `inc` when 01111111 is incremented to 10000000, but otherwise it will be 0.

1-byte machine code instruction
Opcode: `100011`
Operand1: 2-bit register number (`00, 01, 10 or 11`)
Operand2: None

`dec r`n $(\texttt{r}n - 1) \rightarrow \texttt{r}n$ `C,V,Z,N` reflect result

Decrement `r`n
Treats `r`n as a binary number, and subtracts 1 from it.
The subtraction 'wraps around', so when `r`n contains 00000000 beforehand it will contain 11111111 afterwards (and the C, V and Z flags will all be set to 1). The only other time a flag will be set by `inc` is when 01111111 is incremented to 10000000 (in which case the V flag will be set to 1).

1-byte machine code instruction
Opcode: `100010`
Operand1: 2-bit register number (`00, 01, 10 or 11`)
Operand2: None

2. Bit-wise Logic operations

`and r`n`,r`m $(\texttt{r}n \texttt{ and } \texttt{r}m) \rightarrow \texttt{r}m$ `Z,N` reflect result `C,V` become 0

And `r`n with `r`m.
Computes the bitwise conjunction of `r`n and `r`m placing the result in `r`m:
$(\texttt{r}n_0 \wedge \texttt{r}m_0) \rightarrow \texttt{r}m_0$ $(\texttt{r}n_1 \wedge \texttt{r}m_1) \rightarrow \texttt{r}m_1$ $(\texttt{r}n_2 \wedge \texttt{r}m_2) \rightarrow \texttt{r}m_2$ $(\texttt{r}n_3 \wedge \texttt{r}m_3) \rightarrow \texttt{r}m_3$
$(\texttt{r}n_4 \wedge \texttt{r}m_4) \rightarrow \texttt{r}m_4$ $(\texttt{r}n_5 \wedge \texttt{r}m_5) \rightarrow \texttt{r}m_5$ $(\texttt{r}n_6 \wedge \texttt{r}m_6) \rightarrow \texttt{r}m_6$ $(\texttt{r}n_7 \wedge \texttt{r}m_7) \rightarrow \texttt{r}m_7$

2-byte machine code instruction
Opcode: `0100`
Operand1: 2-bit register number (`00, 01, 10 or 11`)

[5] The exception to this is the number -128, represented by 10000000, which has 10000000 as its 8-bit two's complement. So negating -128 gives -128.

Operand2: 2-bit register number (`00, 01, 10 or 11`)

`or rn,rm` `(rn or rm) → rm` `Z,N` reflect result `C,V` become 0

Or rn with rm.
Computes the bitwise disjunction of rn and rm placing the result in rm:
$(\texttt{r}n_0 \vee \texttt{r}m_0) \to \texttt{r}m_0$ $(\texttt{r}n_1 \vee \texttt{r}m_1) \to \texttt{r}m_1$ $(\texttt{r}n_2 \vee \texttt{r}m_2) \to \texttt{r}m_2$ $(\texttt{r}n_3 \vee \texttt{r}m_3) \to \texttt{r}m_3$
$(\texttt{r}n_4 \vee \texttt{r}m_4) \to \texttt{r}m_4$ $(\texttt{r}n_5 \vee \texttt{r}m_5) \to \texttt{r}m_5$ $(\texttt{r}n_6 \vee \texttt{r}m_6) \to \texttt{r}m_6$ $(\texttt{r}n_7 \vee \texttt{r}m_7) \to \texttt{r}m_7$

2-byte machine code instruction
Opcode: `0101`
Operand1: 2-bit register number (`00, 01, 10 or 11`)
Operand2: 2-bit register number (`00, 01, 10 or 11`)

`xor rn,rm` `(rn xor rm) → rm` `Z,N` reflect result `C,V` become 0

Exclusive Or rn with rm.
Computes the bitwise exclusive-or of rn and rm placing the result in rm:
$(\texttt{r}n_0 \oplus \texttt{r}m_0) \to \texttt{r}m_0$ $(\texttt{r}n_1 \oplus \texttt{r}m_1) \to \texttt{r}m_1$ $(\texttt{r}n_2 \oplus \texttt{r}m_2) \to \texttt{r}m_2$ $(\texttt{r}n_3 \oplus \texttt{r}m_3) \to \texttt{r}m_3$
$(\texttt{r}n_4 \oplus \texttt{r}m_4) \to \texttt{r}m_4$ $(\texttt{r}n_5 \oplus \texttt{r}m_5) \to \texttt{r}m_5$ $(\texttt{r}n_6 \oplus \texttt{r}m_6) \to \texttt{r}m_6$ $(\texttt{r}n_7 \oplus \texttt{r}m_7) \to \texttt{r}m_7$

2-byte machine code instruction
Opcode: `0110`
Operand1: 2-bit register number (`00, 01, 10 or 11`)
Operand2: 2-bit register number (`00, 01, 10 or 11`)

`not rn` `(not rn) → rn` `Z,N` reflect result `C,V` become 0

Not rn.
Flips all bits in rn:
$(\neg\texttt{r}n_0) \to \texttt{r}n_0$ $(\neg\texttt{r}n_1) \to \texttt{r}n_1$ $(\neg\texttt{r}n_2) \to \texttt{r}n_2$ $(\neg\texttt{r}n_3) \to \texttt{r}n_3$
$(\neg\texttt{r}n_4) \to \texttt{r}n_4$ $(\neg\texttt{r}n_5) \to \texttt{r}n_5$ $(\neg\texttt{r}n_6) \to \texttt{r}n_6$ $(\neg\texttt{r}n_7) \to \texttt{r}n_7$

1-byte machine code instruction
Opcode: `100000`
Operand1: 2-bit register number (`00, 01, 10 or 11`)
Operand2: None

3. Shifts and Rotates

`shra rn`	`(rn div 2)` → `rn`	`C,Z,N` reflect result	`V` become 0

Arithmetic shift right `rn`
Shift every bit in the bit-string in `rn` one place to the right, whilst leaving the sign bit (bit 7) unchanged. Bit 0 is shifted into `C`; `V` is 0; `N` & `Z` are based on the modified `rn`.
The effect on `rn` is the same as dividing a two's complement number by 2, with the result being that `rn` contains the *quotient* and `C` contains the *remainder* of the division.[6]

1-byte machine code instruction
Opcode: `100110`
Operand1: 2-bit register number (`00, 01, 10 or 11`)
Operand2: None

`shla rn`	`(rn × 2)` → `rn`	`C,V,Z,N` reflect result	`-` become 0

Arithmetic shift left `rn`
Shift every bit in the bit-string in `rn` one place to the left, filling the least significant bit (bit 0) with 0. Bit 7 is shifted into `C`; `V` is 1 if bit 7 changes and 0 if it does not; `N` & `Z` are based on the modified `rn`.
The effect on `rn` is the same as multiplying a two's complement number by 2.[7]

This is a Platform 3½ macro. The macro-assembler inserts `rol rn` wherever `shla rn` is requested by the programmer, as it has the same effect.

`shr rn`	`(rn »)` → `rn`	`C,Z,N` reflect result	`V` become 0

Sliced shift right `rn`
Shifts the bit-string in `rn` one place to the right without maintaining the sign. The old value of `C` is shifted into the sign bit (bit 7), and bit 0 is shifted into `C`. `V` becomes 0. `N` and `Z` are based on the modified `rn`.

1-byte machine code instruction
Opcode: `100100`
Operand1: 2-bit register number (`00, 01, 10 or 11`)
Operand2: None

`shl rn`	`(rn « 1)` → `rn`	`C,V,Z,N` reflect result	`-` become 0

Sliced shift left `rn`
Shifts the bit-string in `rn` one place to the left without ensuring that the result is a multiple of two. The old value of `C` is shifted into bit 0, and the sign bit (bit 7) is shifted into `C`; `V` is 1 if bit 7 changes and 0 if it does not; `N` and `Z` are based on the modified `rn`.

1-byte machine code instruction
Opcode: `100101`
Operand1: 2-bit register number (`00, 01, 10 or 11`)
Operand2: None

`rol rn`	`(rotate-left rn)` → `rn`	`C,V,Z,N` reflect result	`-` become 0

Rotate left `rn`
Treats the bit-string in `rn` as if the opposite ends are directly connected, and shifts it left one place. The sign bit (b7) is shifted into b0, and also into `C`. `V` is cleared. `N` and `Z` are based on the modified `rn`.

[6] This operation may be applied to any 8-bit string, but it can *only* be used to perform division by two two's complement nymbers. `rn` will contain an incorrect value for the quotient under these circumstances.

[7] This operation may be applied to any 8-bit string, and can be used to perform multiplication by two on a bit-string that represent an *unsigned* whole number as well as a two's complement number.

1-byte machine code instruction
Opcode: `100111`
Operand1: 2-bit register number (`00, 01, 10 or 11`)
Operand2: None

Instructions for controlling the flow of execution

1. Branch instructions (none of these change the flags)

`br` *const*	`const` → `PC`	Flags unchanged

Branch unconditionally to a constant address
Copies the bit-string *const* into the Program Counter (PC). This bit-string will be interpreted as an address. The processor will fetch its next instruction from the address *const*, no matter what the state of the flags.

2-byte machine code instruction
Opcode: `11101111`
Operand1: immediate value (second byte of instruction)
Operand2: None

`bz` *const*	`const` → `PC when CVZN matches` − − 1 0	Flags unchanged

Branch on zero to a constant address
Copies the bit-string *const* into the Program Counter (PC), but only if the Z flag contains 1 (in which case the N flag is guaranteed to be 0).
The processor will fetch its next instruction from the address *const*, when the result of the most recent flag-modifying operation was an all-zeros bit-string.

2-byte machine code instruction
Opcode: `11100000`
Operand1: immediate value (second byte of instruction)
Operand2: None

`beq` *const*	`const` → `PC when CVZN matches` − − 1 0	Flags unchanged

Branch on equal to a constant address
Copies the bit-string *const* into the Program Counter (PC), but only if the Z flag contains 1 (in which case the N flag is guaranteed to be 0).
The processor will fetch its next instruction from the address *const*, when the result of the most recent flag-modifying operation was an all-zeros bit-string.

This is a Platform 3½ macro. The macro-assembler inserts `bz` *const* wherever `beq` *const* is requested by the programmer, as it has the same effect.

`bnz` *const*	`const` → `PC when CVZN matches` − − 0 −	Flags unchanged

Branch on non-zero to a constant address
Copies the bit-string *const* into the Program Counter (PC), but only if the Z flag contains 0. This bit-string will be interpreted as an address.
The processor will fetch its next instruction from the address *const*, when the result of the most recent flag-modifying operation was anything other than an all-zeros bit-string.

2-byte machine code instruction
Opcode: `11100001`
Operand1: immediate value (second byte of instruction)

Operand2: None

`bne const` `const → PC when CVZN matches – – 0 –` Flags unchanged

Branch on *not-equal* to a constant address
Copies the bit-string *const* into the Program Counter (PC), but only if the Z flag contains 0. This bit-string will be interpreted as an address.
The processor will fetch its next instruction from the address *const*, when the result of the most recent flag-modifying operation was any bit-string other than all-zeros.

This is a Platform 3½ macro. The macro-assembler inserts `bnz const` wherever `bne const` is requested by the programmer, as it has the same effect.

`blt const` `const → PC when CVZN matches – – 0 1` **Flags unchanged**

Branch on *less-than* to a constant address
Copies the bit-string *const* into the Program Counter (PC), but only if the N flag contains 1 (in which case the Z flag is guaranteed to be 0). This bit-string will be interpreted as an address.
The processor will fetch its next instruction from the address *const*, when the result of the most recent flag-modifying operation was a bit-string that could be interpreted as a negative two's complement number.

2-byte machine code instruction
Opcode: `11100001`
Operand1: **immediate value (second byte of instruction)**
Operand2: **None**

`ble const` `const → PC when CVZN matches – – 0 1 or – – 1 0` **Flags unchanged**

Branch on *less-or-equal* to a constant address
Copies the bit-string *const* into the Program Counter (PC), but only if the Z and N flags are different from one another.
The processor will fetch its next instruction from the address *const*, when the result of the most recent flag-modifying operation was either an all-zeros bit-string or a bit-string that could be interpreted as a negative two's complement number.

2-byte machine code instruction
Opcode: `11100001`
Operand1: **immediate value (second byte of instruction)**
Operand2: **None**

`bgt const` `const → PC when CVZN matches – – 0 0` **Flags unchanged**

Branch on *greater-than* to a constant address
Copies the bit-string *const* into the Program Counter (PC), but only if the Z and N flags are both 0.
The processor will fetch its next instruction from the address *const*, when the result of the most recent flag-modifying operation was a bit-string that could be interpreted as a positive two's complement number (zero is *not* a positive number, any more than it is a negative number).

2-byte machine code instruction
Opcode: 11100001
Operand1: **immediate value (second byte of instruction)**
Operand2: **None**

`bge` *const* `const → PC when CVZN matches − − −0` **Flags unchanged**

Branch on *greater-or-equal* to a constant address
Copies the bit-string *const* into the Program Counter (PC), but only if the N flag is 0.
The processor will fetch its next instruction from the address *const*, when the result of the most recent flag-modifying operation was either a bit-string that could be interpreted as a positive two's complement number, or an all-zeros bit-string.

2-byte machine code instruction
Opcode: 11100001
Operand1: **immediate value (second byte of instruction)**
Operand2: **None**

2. Subroutine call / return

3. Miscellaneous control instructions

`halt` `Stop the clock` **Flags unchanged**

Halt the instruction machine.
Switches off the platform clock.
The PC is not updated, so if the clock is re-started the halt will be executed again. It makes no difference *how* the clock is re-started. Single-stepping the platform has the same effect as re-starting the clock and performing a single *tick*.

1-byte machine code instruction
Opcode: 11010100
Operand1: **None**
Operand2: **None**

`wait` `Suspend the clock` **Flags unchanged**

Wait until an *interrupt* occurs.
Suspends the platform clock in anticipation of an interrupt.
The PC is not updated. If the clock is re-started the wait will be executed again *unless* the re-start is initiated by a hardware interrupt, in which case the PC is loaded with the start address of an interrupt service routine, and *then* the clock is re-started. Single-stepping the platform has the same effect as re-starting the clock *without* an interrupt, and performing a single *tick*.

1-byte machine code instruction
Opcode: 11010101
Operand1: **None**
Operand2: **None**

`jsr` *const* `PC → *SP, then SP+1→SP, then const → PC`

Flags unchanged

Branch on *greater-or-equal* to a constant address
Copies the bit-string *const* into the Program Counter (PC), but only if the N flag is 0.
The processor will fetch its next instruction from the address *const*, when the result of the most recent flag-modifying operation was either a bit-string that could be interpreted as a positive two's complement number, or an all-zeros bit-string.

Wait until an *interrupt* occurs.
Suspends the platform clock in anticipation of an interrupt.
The PC is not updated. If the clock is re-started the wait will be executed again *unless* the re-start is initiated by a hardware interrupt, in which case the PC is loaded with the start address of an interrupt service routine, and *then* the clock is re-started. Single-stepping the platform has the same effect as re-starting the clock *without* an interrupt, and performing a single *tick*.

2-byte machine code instruction
Opcode: `11010110`
Operand1: **immediate value (second byte of instruction)**
Operand2: **None**

11010110 jsr 11010111 rts 11011000 osi 11011001 rti 11011010 crc 11011011 osix