LNCPU

Sommario

[1 Introduction 2](#_Toc60333579)

[2 Registers 3](#_Toc60333580)

[2.1 General purpose registers 3](#_Toc60333581)

[2.1.1 RA 3](#_Toc60333582)

[2.1.2 RB, RC, RD 3](#_Toc60333583)

[2.2 Special registers 3](#_Toc60333584)

[2.2.1 RAM control registers 3](#_Toc60333585)

[2.2.1.1 MAR 3](#_Toc60333586)

[2.2.1.2 MDS 3](#_Toc60333587)

[2.2.1.3 SS 4](#_Toc60333588)

[2.2.1.4 SP 4](#_Toc60333589)

[2.2.1.5 4](#_Toc60333590)

[2.2.2 ROM control registers 4](#_Toc60333591)

[2.2.2.1 CS:PC 4](#_Toc60333592)

[2.2.2.2 SDS 4](#_Toc60333593)

[2.2.3 FLAGS 4](#_Toc60333594)

[2.2.4 IR 4](#_Toc60333595)

[3 ALU 5](#_Toc60333596)

[3.1 ALU Operand Selector 5](#_Toc60333597)

[4 RAM 6](#_Toc60333598)

[4.1 Memory segmentation 6](#_Toc60333599)

[4.2 Control signals 6](#_Toc60333600)

[5 ROM 7](#_Toc60333601)

[Appendix A – Instruction List 9](#_Toc60333602)

[Appendix B – Control signals 11](#_Toc60333603)

# Introduction

LNCPU is an 8-bit data bus 12-bit address bus CPU design equipped with a 8kB volatile RAM and a 8 kB read-only externally programmable ROM for code and data storage.

It has 4 general purpose registers (one of which is a two-directional shift register) along with an ALU that offer basic arithmetic and logic operations (addition, subtraction, AND, OR, XOR, NOT) and sets a 3-bit FLAGS register for conditional jumps.

# Registers

## General purpose registers

General purpose registers are free for the programmer to use. They operate:

a. Synchronously on the data bus.

b. Asynchronously on the ALU internal operand register through the ALU operand selector (see ALU section).

### RA

RA (register A) is an 8-bit left and right shift buffered register.

Control signals assigned to RA:

|  |  |
| --- | --- |
|  | Register input enable (active low) |
|  | Register synchronous output enable (active low) |
|  | Register left shift enable (falling edge) |
|  | Register right shift enable (falling edge) |

Control signals assigned to RA

### RB, RC, RD

RB, RC and RD (register B, C and D) are 8-bit buffered registers.

Control signals assigned to each register (generic RX):

|  |  |
| --- | --- |
|  | Register input enable (active low) |
|  | Register synchronous output enable (active low) |

Control signals assigned to RB, RC, RD each

## Special registers

### RAM control registers

#### MAR

**MAR** (memory address register) is an 8-bit register containing the 8 least significant bits of the RAM address to be accessed at the next RAM read operation. The only control signal assigned to MAR is : MAR outputs asynchronously into the RAM address selector (see RAM section).

#### MDS

**MDS** (memory data segment) is an 8-bit register containing the 5 most significant bits of the RAM address to be accessed at the next RAM read operation, except stack push/pop operations. The 3 most significant bits of this register are hence unused. The only control signal assigned to MDS is : MDS outputs asynchronously into the RAM segment selector (see RAM section).

#### SS

**SS** (stack segment) is an 8-bit register containing the 5 most significant bits of the RAM address to be accessed at the next RAM stack (push/pop). The 3 most significant bits of this register are hence unused. The only control signal assigned to SS is : SS outputs asynchronously into the RAM segment selector (see RAM section).

#### SP

**SP** (stack pointer) is an 8-bit counter register containing the 8 least significant bits of the RAM address to be accessed at the next RAM stack operation. Control signals assigned to SP:

* : increment/decrement on the next clock cycle
* : count direction, 1 for up, 0 for down
* : SP input enable (active low)

MSP outputs asynchronously into the RAM address selector (see RAM section).

#### 

### ROM control registers

#### CS:PC

CS:PC is a 13-bit register containing the address in ROM of the next instruction to be executed. In case an instruction has encoded arguments, CS:PC is incremented in order to retrieve the arguments following the opcode.

#### SDS

**SDS** (storage data segment) is an 8-bit register containing the 5 most significant bits of the ROM address to be accessed at the next ROM data read operation. The 3 most significant bits of this register are hence unused. The only control signal assigned to SDS is : SDS outputs asynchronously into the ROM segment selector (see ROM section).

### FLAGS

### IR

# ALU

The ALU (arithmetic and logic unit) of the LNCPU can perform:

* Arithmetic binary operations: addition, subtraction.
* Logical bitwise operations: AND, OR, XOR.
* Logical bitwise operations: NOT.

Whenever the ALU is active (i.e. when the operator selector signals form a valid operation encoding) the FLAGS register is set, replacing the previous value.

## ALU Operand Selector

The ALU Operand Selector selects the correct operands (X and Y) among the 4 general purpose registers based on 2 control signals for each operand (4 total).

|  |  |  |
| --- | --- | --- |
| / | / | **Operand** |
| 0 | 0 | RA |
| 0 | 1 | RB |
| 1 | 0 | RC |
| 1 | 1 | RD |

## ALU Operator selector

The ALU Operator Selector selects the correct operation by de-multiplexing 3 control signal sinto 8 control bits that control each operation sub-circuit’s buffer. At most one of these control bits will be enabled at any time; resulting in the correct operation being stored in the ALU internal output register.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | **Operation** |
| 0 | 0 | 0 | No operation |
| 0 | 0 | 1 | Addition |
| 0 | 1 | 0 | Subtraction |
| 0 | 1 | 1 | Logical OR |
| 1 | 0 | 0 | Logical AND |
| 1 | 0 | 1 | Logical XOR |
| 1 | 1 | 0 | Logical NOT |
| 1 | 1 | 1 | No operation |

# RAM

## Memory segmentation

The LNCPU has a fully addressable 8kB volatile RAM.

In order to be fully accessible, the RAM is segmented in 32 segments, each 256-byte long; only one segment is accessible at any time.

The LNCPU’s RAM’s full address bus is 13-bit-long (0 through 12) and composed of:

* A 5-bit segment index for the 5 most significant bits;
* An 8-bit address for the 8 least significant bytes.

Registers SS (stack segment) and MDS (memory data segment) specify the segment index.

Registers SP (stack pointer) and MAR (memory address register) specify the 8-bit address.

It can operate in two modes:

* *Stack*: operations PUSH and POP respectively increment and decrement SP thus creating a variable stack; The stack location can be moved by changing the value of SS.
* *Data*: operations such as MOV read and write from/to RAM at arbitrary addresses that can’t be tracked internally; the handling of such addresses is thus left to the user by manipulating (directly or indirectly, see below) registers MDS and MAR.

The only possible combinations for full addresses are SS:SP and MDS:MAR; the correct combination is chosen based on the RAM\_SS (RAM segment selection) control signal:

|  |  |
| --- | --- |
| **RAM\_SS** | **RAM address** |
| 0 | MDS:MAR |
| 1 | SS:SP |

POP and PUSH are the only instructions that use SS:SP, all other RAM read/write operations operate via MDS:MAR.

SS and MAR can be written to directly by the programmer (see the instruction list for details) but cannot be read from.

SP can be manipulated both indirectly via stack operations (PUSH, POP) and directly with MOV, but cannot be read from.

MAR is inaccessible to the user; it is set before every RAM I/O instruction to contain the address during data reads.

## Control signals

* : RAM input enable (active low);
* : RAM output enable (active low);
* : RAM address selection, 0 for address bus, 1 for SS:SP;

# ROM

## Control signals

* : ROM address selection, 0 for address bus, 1 for CS:PC;

# Appendix A – Instruction List

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Name** | **Syntax** | **Variants[[1]](#footnote-1)\*** | **Total opcodes** | **Description** | **FLAGS**  **set** | **Memory segment** |
| No operation | nop | nop | 1 | No operation. | None. | None. |
| Halt | hlt | hlt | 1 | Halts the CPU irreversibly. | None. | None. |
| Break | brk | Brk | 1 | Causes a non-maskable software interrupt. |  |  |
| Copy | mov <*src> <dest>* | *mov constant,RX*  *mov constant, SS*  *mov constant, SP*  mov *RD, SS*  mov *RD, SP*  mov RX1, RX2  mov *RX, [address]1,*  mov *[address]1, RX*  mov *constant*, *[address]1*  mov *RX*, [RD]2  mov [RD]2, RX  mov *constant*, [RD]2  mov *RX*, [*page:*RD]3  mov [*page:*RD]3, RX  mov *constant*, [*page:RD*]3  mov RX, [RC:RD]4  mov [RC:RD]4, RX  mov *constant*, [RC:RD]4  mov *RX, [long address]5,*  mov *[long address]5, RX*  mov *constant*, *[long address]5* | 65 | Copies the content of <*src> into <dest>.*  Addresses in memory are specified with square brackets. When copying into special registers and when using indirect addressing modes, RD is the only source allowed. | None. | None. |
| Push | push <*src>* | push RX  push [*address*]1  push [RD]2  push [page:RD]3  push [rc:rd]4  push [long address]5  push *constant* | 10 | Copies the specified value into the address SS:MSP, then increments MSP. | None. | SS |
| Pop | pop <dest> | pop RX  pop [*address*]1  pop [RD]2  pop [page:RD]3  pop [rc:rd]4  pop [long address]5 | 9 | Decrements MSP and then copies the value at address SS:MSP into the specified destination. | None. | SS |
| Add | add <O1> <O2> | add RX1, RX2  add RX­, *constant* | 20 | Performs an arithmetic addition on the given operands and stores the results in the first operand’s source. | NZC | MDS |
| Subtract | sub <O1> <O2> | sub RX1, RX2  sub RX, *constant* | 20 | Subtracts the second operand from the first and stores the result back into the first operand’s source; | NZC | MDS |
| Compare | cmp <O1> <O2> | cmp RX1, RX2  cmp RX, *constant* | 20 | Subtracts the second operand from the first without storing the result. | NZC | MDS |
| Logical  AND | and <O1> <O2> | and RX1, RX2  and RX, *constant* | 20 | ANDs the two operands together and stores the result in the first operand’s source. | NZ | None. |
| Logical  OR | or <O1> <O2> | or RX1, RX2  or RX, *constant* | 20 | ORs the two operands together and stores the result in the first operand’s source. | NZ | None. |
| Logical  XOR | xor <O1> <O2> | xor RX1, RX2  xor RX, *constant* | 20 | XORs the two operands together and stores the result in the first operand’s source. | NZ | None. |
| Logical NOT | not <O> | not RX | 4 | Inverts the given operand and stores the result in the operand’s source. | NZ | None. |
| Increment | inc RX | inc RX | 4 | Increments the given register. | NZC |  |
| Decrement | dec RX | dec RX | 4 | Decrements the given register. |  |  |
| Shift left (RA only) | shl | shl RA | 1 | Performs a left bitwise shift on the RA register. | None. | None. |
| Shift right (RA only) | shr | shr RA | 1 | Perform a right bitwise shift on the RA register. | None. | None. |
| Short jump if carry | jc *<address>* | jc *<address>* | 1 | If the carry (C) flag is set, loads the given address into the least significant 8 bits of CS:PC. | None. | None. |
| Short jump if zero | jz *<address>* | jc *<address>* | 1 | If the zero (Z) flag is set, loads the given address into the least significant 8 bits of CS:PC. | None. | None. |
| Short jump if negative | jn *<address>* | jn <*address>* | 1 | If the negative (N) flag is set, loads the given address into the least significant 8 bits of CS:PC. | None. | None. |
| Unconditional jump | goto *<address>* | goto <*address>* | 1 | Loads the given address into the least significant 8 bits of CS:PC. | None. | None. |
| Long jump if negative | jn *<address>* | jn <cs>:<*address>* | 1 | If the negative (N) flag is set, loads the given code segment into the most significant 5 bits of CS:PC and the given address into the least significant 8 bits of CS:PC. | None. | None. |
| Long jump if carry | jc *<address>* | jc *<cs>:<address>* | 1 | If the carry (C) flag is set, loads the given code segment into the most significant 5 bits of CS:PC and the given address into the least significant 8 bits of CS:PC. | None. | None. |
| Long jump if zero | jz *<address>* | jz *<cs>:<address>* | 1 | If the zero (Z) flag is set, loads the given code segment into the most significant 5 bits of CS:PC and the given address into the least significant 8 bits of CS:PC. | None. | None. |
| Unconditional long jump | goto *<address>* | goto <cs:*address>* | 1 | Loads the given code segment into the most significant 5 bits of CS:PC and the given address into the least significant 8 bits of CS:PC. | None. | None. |
| Call | call <cs:address> | call <cs:address> | 1 | Pushes CS and PC (in this order) onto the stack; then performs an unconditional jump to the given address and code segment. | None. | None. |
| Return from call | ret | ret | 1 | Pops PC and CS (in this order) from the stack | None. | None. |
| Set interrupt disable flag | sid | sid | 1 | Sets the I (interrupt disable) bit in the FLAGS register. | I | I |
| Clear interrupt disable flag | cid | cid | 1 | Clears the I (interrupt disable) bit in the FLAGS register. | I | I |
| Return from interrupt | iret | iret | 1 | Returns from the interrupt, restoring CS:PC and FLAGS from the stack. | ANZC | None. |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  | 234 |  |  |  |

# Appendix B - Opcodes

|  |  |  |
| --- | --- | --- |
| **Opcode** | **Instruction** | **Clock cycles** |
|  | nop | 3 |
|  | hlt | 4 |
|  | mov\_ra\_rb | 4 |
|  | mov\_ra\_rc | 4 |
|  | mov\_ra\_rd | 4 |
|  | mov\_rb\_ra | 4 |
|  | mov\_rb\_rc | 4 |
|  | mov\_rb\_rd | 4 |
|  | mov\_rc\_ra | 4 |
|  | mov\_rc\_rb | 4 |
|  | mov\_rc\_rd | 4 |
|  | mov\_rd\_ra | 4 |
|  | mov\_rd\_rb | 4 |
|  | mov\_rd\_rc | 4 |
|  | mov\_rd\_ss | 4 |
|  | mov\_rd\_mds | 4 |
|  | mov\_rd\_sds | 4 |
|  | mov\_rd\_sp | 4 |
|  | mov\_ra\_ram | 5 |
|  | mov\_rb\_ram | 5 |
|  | mov\_rc\_ram | 5 |
|  | mov\_rd\_ram | 5 |
|  | mov\_rom\_ra | 5 |
|  | mov\_rom\_rb | 5 |
|  | mov\_rom\_rc | 5 |
|  | mov\_rom\_rd | 5 |
|  | mov\_ram\_ra | 5 |
|  | mov\_ram\_rb | 5 |
|  | mov\_ram\_rc | 5 |
|  | mov\_ram\_rd | 5 |
|  | mov\_cst\_ra | 5 |
|  | mov\_cst\_rb | 5 |
|  | mov\_cst\_rc | 5 |
|  | mov\_cst\_rd | 5 |
|  | mov\_cst\_ss | 5 |
|  | mov\_cst\_mds | 5 |
|  | mov\_cst\_sds | 5 |
|  | mov\_cst\_sp | 5 |
|  | mov\_cst\_ram | 5 |
|  | mov\_rom\_ram | 6 |
|  | push\_ra | 5 |
|  | push\_rb | 5 |
|  | push\_rc | 5 |
|  | push\_rd | 5 |
|  | push\_cst | 5 |
|  | push\_rom | 6 |
|  | pop\_ra | 5 |
|  | pop\_rb | 5 |
|  | pop\_rc | 5 |
|  | pop\_rd | 5 |
|  | add\_ra\_ra | 5 |
|  | add\_ra\_rb | 5 |
|  | add\_ra\_rc | 5 |
|  | add\_ra\_rd | 5 |
|  | add\_rb\_ra | 5 |
|  | add\_rb\_rb | 5 |
|  | add\_rb\_rc | 5 |
|  | add\_rb\_rd | 5 |
|  | add\_rc\_ra | 5 |
|  | add\_rc\_rb | 5 |
|  | add\_rc\_rc | 5 |
|  | add\_rc\_rd | 5 |
|  | add\_rd\_ra | 5 |
|  | add\_rd\_rb | 5 |
|  | add\_rd\_rc | 5 |
|  | add\_rd\_rd | 5 |
|  | sub\_ra\_ra | 5 |
|  | sub\_ra\_rb | 5 |
|  | sub\_ra\_rc | 5 |
|  | sub\_ra\_rd | 5 |
|  | sub\_rb\_ra | 5 |
|  | sub\_rb\_rb | 5 |
|  | sub\_rb\_rc | 5 |
|  | sub\_rb\_rd | 5 |
|  | sub\_rc\_ra | 5 |
|  | sub\_rc\_rb | 5 |
|  | sub\_rc\_rc | 5 |
|  | sub\_rc\_rd | 5 |
|  | sub\_rd\_ra | 5 |
|  | sub\_rd\_rb | 5 |
|  | sub\_rd\_rc | 5 |
|  | sub\_rd\_rd | 5 |
|  | or\_ra\_ra | 5 |
|  | or\_ra\_rb | 5 |
|  | or\_ra\_rc | 5 |
|  | or\_ra\_rd | 5 |
|  | or\_rb\_ra | 5 |
|  | or\_rb\_rb | 5 |
|  | or\_rb\_rc | 5 |
|  | or\_rb\_rd | 5 |
|  | or\_rc\_ra | 5 |
|  | or\_rc\_rb | 5 |
|  | or\_rc\_rc | 5 |
|  | or\_rc\_rd | 5 |
|  | or\_rd\_ra | 5 |
|  | or\_rd\_rb | 5 |
|  | or\_rd\_rc | 5 |
|  | or\_rd\_rd | 5 |
|  | and\_ra\_ra | 5 |
|  | and\_ra\_rb | 5 |
|  | and\_ra\_rc | 5 |
|  | and\_ra\_rd | 5 |
|  | and\_rb\_ra | 5 |
|  | and\_rb\_rb | 5 |
|  | and\_rb\_rc | 5 |
|  | and\_rb\_rd | 5 |
|  | and\_rc\_ra | 5 |
|  | and\_rc\_rb | 5 |
|  | and\_rc\_rc | 5 |
|  | and\_rc\_rd | 5 |
|  | and\_rd\_ra | 5 |
|  | and\_rd\_rb | 5 |
|  | and\_rd\_rc | 5 |
|  | and\_rd\_rd | 5 |
|  | xor\_ra\_ra | 5 |
|  | xor\_ra\_rb | 5 |
|  | xor\_ra\_rc | 5 |
|  | xor\_ra\_rd | 5 |
|  | xor\_rb\_ra | 5 |
|  | xor\_rb\_rb | 5 |
|  | xor\_rb\_rc | 5 |
|  | xor\_rb\_rd | 5 |
|  | xor\_rc\_ra | 5 |
|  | xor\_rc\_rb | 5 |
|  | xor\_rc\_rc | 5 |
|  | xor\_rc\_rd | 5 |
|  | xor\_rd\_ra | 5 |
|  | xor\_rd\_rb | 5 |
|  | xor\_rd\_rc | 5 |
|  | xor\_rd\_rd | 5 |
|  | not\_ra | 5 |
|  | not\_rb | 5 |
|  | not\_rc | 5 |
|  | not\_rd | 5 |
|  | shl\_ra | 4 |
|  | shr\_ra | 4 |
|  | jc | 5 |
|  | jz | 5 |
|  | jn | 5 |
|  | ja | 5 |
|  | goto | 5 |
|  | ljc | 7 |
|  | ljz | 7 |
|  | ljn | 7 |
|  | lja | 7 |
|  | lgoto | 7 |
|  | call | 11 |
|  | ret | 7 |
|  | poll\_ra | 4 |
|  | poll\_rb | 4 |
|  | poll\_rc | 4 |
|  | poll\_rd | 4 |
|  | tsm\_ra | 4 |
|  | tsm\_rb | 4 |
|  | tsm\_rc | 4 |
|  | tsm\_rd | 4 |
|  | tsm\_ram | 5 |
|  | tsm\_rom | 5 |
|  | tsmc | 5 |
|  | rfl | 4 |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

# Appendix C – Control signals

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Location** | **Description** | **Default state[[2]](#footnote-2)** |
|  | EEPROM 0 - 0 | RA (register A) input enable (active low) | 1 |
|  | EEPROM 0 - 1 | RA (register A) output enable (active low) | 1 |
|  | EEPROM 0 - 2 | RA (register A) shift left (active low) | 1 |
|  | EEPROM 0 - 3 | RA (register A) shift right (active low) | 1 |
|  | EEPROM 0 - 4 | RB (register B) input enable (active low) | 1 |
|  | EEPROM 0 - 5 | RB (register B) output enable (active low) | 1 |
|  | EEPROM 0 - 6 | RC (register C) input enable (active low) | 1 |
|  | EEPROM 0 - 7 | RC (register C) output enable (active low) | 1 |
|  | EEPROM 1 - 0 | RD (register D) input enable (active low) | 1 |
|  | EEPROM 1 - 1 | RD (register D) output enable (active low) | 1 |
|  | EEPROM 1 - 2 | MDS (memory data segment) input enable (active low) | 1 |
|  | EEPROM 1 - 3 | MAR (memory address register) input enable (active low) | 1 |
|  | EEPROM 1 - 4 | SS (stack segment) input enable (active low) | 1 |
|  | EEPROM 1 - 5 | SP (stack pointer) input enable (active low) | 1 |
|  | EEPROM 1 - 6 | SP (stack pointer) count enable (active low) | 1 |
|  | EEPROM 1 - 7 | SP (stack pointer) count direction | \* |
|  | EEPROM 2 - 0 | SDS (storage data segment) input enable (active low) | 1 |
|  | EEPROM 2 - 1 | SAR (storage address register) input enable (active low) | 1 |
|  | EEPROM 2 - 2 | ALU X input selector 0 | \* |
|  | EEPROM 2 - 3 | ALU X input selector 1 | \* |
|  | EEPROM 2 - 4 | ALU Y input selector 0 | \* |
|  | EEPROM 2 - 5 | ALU Y input selector 1 | \* |
|  | EEPROM 2 - 6 | ALU operator selector 0 | \* |
|  | EEPROM 2 - 7 | ALU operator selector 1 | \* |
|  | EEPROM 3 - 0 | ALU operator selector 2 | \* |
|  | EEPROM 3 - 1 | ALU enable (active low) | 1 |
|  | EEPROM 3 - 2 | ALU output enable (active low) | 1 |
|  | EEPROM 3 - 3 | RAM input enable (active low) | 1 |
|  | EEPROM 3 - 4 | RAM output enable (active low) | 1 |
|  | EEPROM 3 - 5 | RAM address selector | \* |
|  | EEPROM 3 - 6 | ROM output enable (active low) | 1 |
|  | EEPROM 3 - 7 | ROM address selector | \* |
|  | EEPROM 4 - 0 | CS (code segment) input enable (active low) | 1 |
|  | EEPROM 4 - 1 | PC (program counter) input enable (active low) | 1 |
|  | EEPROM 4 - 2 | PC (program counter) increment enable (active low) | 1 |
|  | EEPROM 4 - 3 | CS (code segment) output enable (active low) | 1 |
|  | EEPROM 4 - 4 | PC (program counter) output enable (active low) | 1 |
|  | EEPROM 4 - 5 | IR (instruction register) input enable (active low) | 1 |
|  | EEPROM 4 - 6 | Interface poll | 0 |
|  | EEPROM 4 - 7 | Interface transmit | 0 |
|  | EEPROM 5 - 0 | LCD enable | 0 |
|  | EEPROM 5 - 1 | LCD register select | \* |
|  | EEPROM 5 - 1 | LCD read/write | \* |
|  | EEPROM 5 - 3 | FLAGS input enable (active low) | 1 |
|  | EEPROM 5 - 4 | Reset flags (active low) | 1 |
|  | EEPROM 5 - 5 | Unconditional jump[[3]](#footnote-3) | 0 |
|  | EEPROM 5 - 6 | Jump if carry3 | 0 |
|  | EEPROM 5 - 7 | Jump if zero3 | 0 |
|  | EEPROM 6 - 0 | Jump if negative3 | 0 |
|  | EEPROM 6 - 1 | Jump if data available3 | 0 |
|  | EEPROM 6 - 2 | Terminate instruction, advance to the next | 1 |
|  | EEPROM 6 - 3 | Halt clock (active low) | 1 |
| - | EEPROM 6 – 4 | Unused | - |
| - | EEPROM 6 – 5 | Unused | - |
| - | EEPROM 6 - 6 | Unused | - |
| - | EEPROM 6 - 7 | Unused | - |

1. \* RX denotes any one of the general-purpose registers (RA through RD).

   1 Page 0 addressing mode.

   2 Indirect page 0 addressing mode.

   3 Indirect paged addressing mode.

   4 Full indirect addressing mode.

   5 Absolute addressing mode. [↑](#footnote-ref-1)
2. For enable signals only; for other types of signals that affect the behavior of a component without activating it, an asterisk (\*) will be shown. [↑](#footnote-ref-2)
3. Use or to specify where to load the address [↑](#footnote-ref-3)