# Instruction set of the Mic1 Macro Language

<table>
<thead>
<tr>
<th>Binary</th>
<th>Mnemonic</th>
<th>Instruction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000xxxxxxxxxxxx</td>
<td>LODD</td>
<td>Load direct</td>
<td>$ac := m[x]$</td>
</tr>
<tr>
<td>0001xxxxxxxxxxxx</td>
<td>STOD</td>
<td>Store direct</td>
<td>$m[x] := ac$</td>
</tr>
<tr>
<td>0010xxxxxxxxxxxx</td>
<td>ADDD</td>
<td>Add direct</td>
<td>$ac := ac + m[x]$</td>
</tr>
<tr>
<td>0011xxxxxxxxxxxx</td>
<td>SUBD</td>
<td>Subtract direct</td>
<td>$ac := ac - m[x]$</td>
</tr>
<tr>
<td>0100xxxxxxxxxxxx</td>
<td>JPOS</td>
<td>Jump positive</td>
<td>if $ac \geq 0$ then $pc := x$</td>
</tr>
<tr>
<td>0101xxxxxxxxxxxx</td>
<td>JZER</td>
<td>Jump zero</td>
<td>if $ac = 0$ then $pc := x$</td>
</tr>
<tr>
<td>0110xxxxxxxxxxxx</td>
<td>JUMP</td>
<td>Jump</td>
<td>$pc := x$</td>
</tr>
<tr>
<td>0111xxxxxxxxxxxx</td>
<td>LOCO</td>
<td>Load constant</td>
<td>$ac := x \ (0 \leq x \leq 4095)$</td>
</tr>
<tr>
<td>1000xxxxxxxxxxxx</td>
<td>LODL</td>
<td>Load local</td>
<td>$ac := m[sp + x]$</td>
</tr>
<tr>
<td>1001xxxxxxxxxxxx</td>
<td>STOL</td>
<td>Store local</td>
<td>$m[x + sp] := ac$</td>
</tr>
<tr>
<td>1010xxxxxxxxxxxx</td>
<td>ADDL</td>
<td>Add local</td>
<td>$ac := ac + m[sp + x]$</td>
</tr>
<tr>
<td>1011xxxxxxxxxxxx</td>
<td>SUBL</td>
<td>Subtract local</td>
<td>$ac := ac - m[sp + x]$</td>
</tr>
<tr>
<td>1100xxxxxxxxxxxx</td>
<td>JNEG</td>
<td>Jump negative</td>
<td>if $ac &lt; 0$ then $pc := x$</td>
</tr>
<tr>
<td>1101xxxxxxxxxxxx</td>
<td>JNZE</td>
<td>Jump nonzero</td>
<td>if $ac \neq 0$ then $pc := x$</td>
</tr>
<tr>
<td>1110xxxxxxxxxxxx</td>
<td>CALL</td>
<td>Call procedure</td>
<td>$sp := sp - 1; m[sp] := pc; pc := x$</td>
</tr>
<tr>
<td>1111000000000000</td>
<td>PSHI</td>
<td>Push indirect</td>
<td>$sp := sp - 1; m[sp] := m[ac]$</td>
</tr>
<tr>
<td>1111001000000000</td>
<td>POPI</td>
<td>Pop indirect</td>
<td>$m[ac] := m[sp]; sp := sp + 1$</td>
</tr>
<tr>
<td>1111010000000000</td>
<td>PUSH</td>
<td>Push onto stack</td>
<td>$sp := sp - 1; m[sp] := ac$</td>
</tr>
<tr>
<td>1111011000000000</td>
<td>POP</td>
<td>Pop from stack</td>
<td>$ac := m[sp]; sp := sp + 1$</td>
</tr>
<tr>
<td>1111100000000000</td>
<td>RETN</td>
<td>Return</td>
<td>$pc := m[sp]; sp := sp + 1$</td>
</tr>
<tr>
<td>1111101000000000</td>
<td>SWAP</td>
<td>Swap ac, sp</td>
<td>$tmp := ac; ac := sp; sp := tmp$</td>
</tr>
<tr>
<td>11111110yyyyyyyy</td>
<td>INS</td>
<td>Increment sp</td>
<td>$sp := sp + y \ (0 \leq y \leq 255)$</td>
</tr>
<tr>
<td>11111111yyyyyyyy</td>
<td>DESP</td>
<td>Decrement sp</td>
<td>$sp := sp - y \ (0 \leq y \leq 255)$</td>
</tr>
</tbody>
</table>

`xxxxxxxxxxxxx` is a 12-bit machine address; in column 4 it is called $x$.
`yyyyyyyyyy` is an 8-bit constant; in column 4 it is called $y$. 
The Mic1 example is based on the **AMD 2903** bit slice processor

The various instruction formats include:

- 4 bit opcodes with remaining 12 bits used as either address or immediate value. In both cases the 12 bits are treated as an unsigned magnitude integer with range from 0 to 4095

| 0000 - 1110 Op Codes from LODD to CALL | Used an 12 bit address range 0 to 4095 Or a 12 bit unsigned integer with this range |

- 7 bit opcodes with the eighth bit set to zero and the low 8 bits used only as a positive value with range of 0 to 255 for the INSP and DESP (increment/decrement stack pointer) instructions (always zeros for other 7 bit opcodes)

| 1111000 - 1111111 Op Codes from PSHI to DESP | 0 | Low 8 bits unused except for INSP and DESP where 0 - 255 range |

- Eighth bit zero except with the halt instruction: 11111111

Data use is (for now) based on simple 16 bit 2s complement integers:

| Sign Bit | 15 bits of integer significance, providing values from -32K to +(32K - 1) |
Below is a simple example of a program that includes a function called **adder** that takes two arguments that include the address of an array of 2s complement integers, and the number of elements in that array, such that its signature is:

```
adder array_count array_address
```

The program sets up the stack with the appropriate argument values and then calls **adder**. The **adder** routine finds the array of numbers, adds them together and then returns with the sum in the **AC** (as previously mentioned, the convention is to return function results in the AC). The main program, upon return from the adder call, then stores the AC contents into the memory **rslt**: location and calls halt to enter the debugger.

```
start:  lodd daddr:   ;load AC with data address
        push        ;push AC to stack (2nd arg)
        lodd dcnt:  ;load AC with data count
        push        ;push AC to stack (1st arg)
        call adder: ;push return address on stack
        stod rslt:  ;store AC (has sum) to rslt: location
        halt        ;enter debugger

daddr:  data:  ;location holds data array address
    25          ;first of 5 data values
    50          ;last of 5 data values
    75
    100
    125

dcnt:   5          ;location holds data array element count
rslt:   0          ;location for the sum to be stored
.loc 20    ;forces adder routine to start at location 20
adder:  lodl 1      ;get 1st arg from stack into AC (data count)
        stod mycnt: ;store count at location mycnt:
        lodl 2      ;get 2nd arg from stack into AC (data addr)
        pshi        ;push indirect first datum to stack
        addd myc1:  ;add 1 (value at myc1:) to addr in AC
        stod myptr: ;store new addr to location myptr:
        loop:       lodd mycnt: ;load AC with value at mycnt: (data count)
                     subd myc1: ;subtract 1 (value at myc1:) from AC
                     jzer done: ;if new data count is 0 go to location done:
                     stod mycnt: ;if more data to add, store new data count
        lodd myptr: ;load AC with addr of next datum
        pshi        ;push indirect next datum to stack
        addd myc1:  ;add 1 (value at myc1:) to addr in AC
        stod myptr: ;store new addr to location myptr:
        pop         ;pop top of stack into AC (new datum)
        addl 0      ;add new top of stack location to AC
        insp 1      ;move stack pointer down one place
        push        ;push new sum in AC onto stack
        jump loop:  ;jump to location loop:
        done:       pop         ;come here when all data added, sum in AC
                     retn        ;return to caller
                     halt        ;should never get here (safety halt)
mycnt:  0          ;location for running count
myptr:  0          ;location for running data pointer
myc1:   1          ;location of a constant value of 1
```
The program from the previous page must be assembled, and then run with the Mic1 emulator. The following is a transcript of this activity using the mercury system:

```
bash-2.05$ cd cs305
bash-2.05$ pwd
/usr/cs/fac1/bill/cs305 (you can cd -bill/cs305 to get here)
bash-2.05$ ./masm < adder.asm > adder.obj
bash-2.05$ ./mic1 prom.dat adder.obj 0 1024

Read in 81 micro instructions
Read in 45 machine instructions
Starting PC is: 0000000000000000 base 10: 0
Starting SP is: 0000010000000000 base 10: 1024

Program Counter : 0000000000000011 base 10: 7
Accumulator    : 0000000101110111 base 10: 375
Instruction Reg : 1111111100000000 base 10: 65280
Temp Instr      : 1000000000000000 base 10: 32768
Stack Pointer   : 0000000111111110 base 10: 1022
A Register      : 1111111111111110 base 10: 65534
B Register      : 0000000000000000 base 10: 0
C Register      : 0000000000000000 base 10: 0
D Register      : 0000000000000000 base 10: 0
E Register      : 0000000000000000 base 10: 0
F Register      : 0000000000000000 base 10: 0

Total cycles : 683

Type decimal address to view memory, q to quit or c to continue: 7
the location 7 has value 00000000000001000, or 8 or signed 8
Type <Enter> to continue debugging
Type q to quit
Type f for forward range
Type b for backward range: f
Type the number of forward locations to dump: 10
the location 8 has value 00000000000001010, or 25 or signed 25
the location 9 has value 00000000000001000, or 50 or signed 50
the location 10 has value 00000000000001001, or 75 or signed 75
the location 11 has value 00000000000001100, or 100 or signed 100
the location 12 has value 00000000000001101, or 125 or signed 125
the location 13 has value 00000000000000010, or 5 or signed 5
the location 14 has value 00000000000001010, or 375 or signed 375
the location 15 has value 00000000000001110, or 65535 or signed -1
the location 16 has value 00000000000001111, or 65535 or signed -1
the location 17 has value 00000000000000001, or 65535 or signed -1

Type decimal address to view memory, q to quit or c to continue: 1024
the location 1024 has value 00000000000001000, or 8 or signed 8
Type <Enter> to continue debugging
Type q to quit
Type f for forward range
Type b for backward range: b
Type the number of reverse locations to dump: 6
the location 1023 has value 00000000000001010, or 25 or signed 25
the location 1022 has value 00000000000001000, or 50 or signed 50
the location 1021 has value 00000000000001001, or 75 or signed 75
the location 1020 has value 00000000000001100, or 100 or signed 100
the location 1019 has value 00000000000001101, or 125 or signed 125
the location 1018 has value 00000000000000010, or 5 or signed 5
the location 1017 has value 00000000000000011, or 375 or signed 375
the location 1016 has value 00000000000000001, or 65535 or signed -1

Type decimal address to view memory, q to quit or c to continue: q

MIC-1 emulator finishing, goodbye
bash-2.05$
```