B1601

Instruction Set & Assembler Programming Manual

B1601 assembler programming

Instruction codes

LOAD
Syntax:
```
load val, dst;
```
Operation:
Stores the supplied value (val) to the destination (dst).

COPY
Syntax:
```
copy src, dst;
```
Operation:
Load a value from the source (src) and stores it to the destination (dst).

ADD
Syntax:
```
add src, dst;
```
Operation:
Adds the values of the source (src) and the destination (dst) and stores the result into the destination (dst). On overflow, the overflow flag in the flags register is set, otherwise it is cleared.

SUB
Syntax:
```
sub src, dst;
```
Operation:
Subtracts the value of the source (src) from the value of the destination (dst) and stores the result into the destination (dst). On underflow, the overflow flag in the flags register is set, otherwise it is cleared.

MUL
Syntax:
```
mul src, dst;
```
Operation:
Multiplies the values of the source (src) and the destination (dst) and stores the result into the destination (dst). On overflow, the overflow flag in the flags register is set, otherwise it is cleared.

DIV
Syntax:
```
div src, dst;
```
Operation:
Divides the value of the destination (dst) by the value of source (src) and stores the result into the destination (dst).

COMP
Syntax:
```
comp src, dst;
```
Operation:
Compares the value of the source (src) with the value of the destination (dst). If the values are not equal, the difference flag in the flags register. If the value of the source (src) is less than the value of the destination (dst), sets the less flag in the flags register.

AND
Syntax:
```
and src, dst;
```
Operation:
Performs a bit-wise logical AND on the values of the source (src) and the destination (dst) and stores the result into the destination (dst).

OR
Syntax:
```
or src, dst;
```
Operation:
Performs a bit-wise logical OR on the values of the source (src) and the destination (dst) and stores the result into the destination (dst).

XOR
Syntax:
```
xor src, dst;
```
Operation:
Performs a bit-wise logical XOR on the values of the source (src) and the destination (dst) and stores the result into the destination (dst).

NOT

Syntax:

```
not src, dst;
```

Operation:

Stores the bit-wise logical NOT (inverted) value of the source (src) into the destination (dst).

LFLG

Syntax:

```
lflg src, dst;
```

Operation:

Copies the value of the flags register into the lower 8 bits of the destination (dst).

JUMP

Syntax:

```
jump addr;
jump label;
```

Operation:

Resumes the execution of the program at the specified program memory address (addr). In assembler source codes, the name of a jump label can be used; when the source code is compiled, the assembler compiler automatically translates label names into addresses.

JUMPEQ

Syntax:

```
jumpeq addr;
jumpeq label;
```

Operation:

Resumes the execution of the program at the specified program memory address (addr) if the difference flag in the flags register is cleared (0). In assembler source codes, the name of a jump label can be used; when the source code is compiled, the assembler compiler automatically translates label names into addresses.

CALL

Syntax:

```
call addr;
call label;
```

Operation:

Stores the program memory address of the next instruction on the call stack and resumes the execution of the program at the specified program memory address (addr). In assembler source codes, the name of a jump label can be used; when the source code is compiled, the assembler compiler automatically translates label names into addresses.

RET

Syntax:

```
ret;
```

Operation:

Loads a program memory address from the call stack and resumes the execution of the program at that address.

INC

Syntax:

```
inc src, dst;
```

Operation:

Increases the value of the source (src) by 1 and stores the result to the destination (dst). On overflow, the overflow flag in the flags register is set, otherwise it is cleared.

DEC

Syntax:

```
dec src, dst;
```

Operation:

Decreases the value of the source (src) by 1 and stores the result to the destination (dst). On overflow, the overflow flag in the flags register is set, otherwise it is cleared.

STOP

Syntax:

```
stop;
```

Operation:

Ends program execution.

Source and destination specifications:

The source (src) operand can specify a register or the random access memory (RAM). The destination (dst) operand can specify a register, the random access memory (RAM) or the IO channel (terminal output).

source / destination type	operand name	numerical id	description
register	acc	0	general purpose register
register	addr	1	address register for RAM
register	r0	2	general purpose register
register	r1	3	general purpose register
register	r2	4	general purpose register
register	r3	5	general purpose register
register	r4	6	general purpose register
register	sp	7	stack pointer
memory (RAM)	mem	14	memory location selected by the *addr* register
IO channels	io	15	IO channel (terminal output)

The *addr* register defines the memory location that is used when the memory is specified as the source or destination of an instruction.

CAUTION! Instructions can not load a value from the memory and store a value to the memory in the same clock cycle. Therefore, instructions can not have both their source and their destination specification set to mem (memory). This limitation does not apply to registers.
For this reason, an instruction like, for instance

```
inc r0, r0;
```

is valid while an instruction like, for instance

```
inc mem, mem;
```

is not valid and can not be expected to yield a correct result.

The flags register

The flags register holds the value of 3 flags: *overflow*, *difference* and *less*. Each flag is represented as a 1 bit value in the flags register.

flag name	bit number in the flags register	numerical value
overflow (OF)	1	1
difference (DF)	2	2
less (LF)	3	4

Native code (binary machine code)

All instructions codes are represented as 16 bit values. The LOAD, JUMP and CALL instructions are followed by another 16 bit long argument value.

The general instruction format is:

1 bit	7 bits	4 bits	4 bits
enable flag	instruction code	source selector (if required)	destination selector (if required) condition flags (*jumpeq* instruction only)

If the enable flag is not set, the processor will stop at that instruction instead of executing it. This mechanism can be used to insert a breakpoint into a program's code without having to remember the original instruction code.

Instruction codes:

Values in this table are hexadecimal numbers

See the table key below for the meaning of non-numerical symbols in this table

Instruction name	7 bit instr. code	16 bit instr. code (enabled)	16 bit instr. code (disabled)	full instr. length & execution time
load	00	80 0D	00 0D data	32 bits, 2 cycles
ret	01	81 00	01 00	16 bits, 1 cycle
call	02	82 00 address	02 00 address	32 bits, 2 cycles
jump	03	83 00 address	03 00 address	32 bits, 2 cycles
jumpeq	03	83 01 address	03 01 address	32 bits, 2 cycles
copy	04	84 SD	04 SD	16 bits, 1 cycle
comp	05	85 SD	05 SD	16 bits, 1 cycle
add	06	86 SD	06 SD	16 bits, 1 cycle
sub	07	87 SD	07 SD	16 bits, 1 cycle
mul	08	88 SD	08 SD	16 bits, 1 cycle
div	09	89 SD	09 SD	16 bits, 1 cycle
and	0a	8a SD	0a SD	16 bits, 1 cycle
or	0b	8b SD	0b SD	16 bits, 1 cycle
xor	0c	8c SD	0c SD	16 bits, 1 cycle
not	0d	8d SD	0d SD	16 bits, 1 cycle
lflg	0e	8e 0D	0e 0D	16 bits, 1 cycle
inc	0f	8f SD	0f SD	16 bits, 1 cycle
dec	10	90 SD	10 SD	16 bits, 1 cycle

Table key:

S	4 bit source selector
D	4 bit destination selector
data	16 bit argument value
address	16 bit argument value

Note: There is no actual instruction code for the assembler compiler's *stop* instruction. The assembler compiler generates an all-zero instruction code for this instruction, which is identical to a disabled "load data, acc;" instruction. Since the instruction is not executed, the data argument can be omitted in this special case.

Source and destination selector values:

Values in this table are hexadecimal numbers

source/destination name	4 bit value
acc	0
addr	1
r0	2
r1	3
r2	4
r3	5
r4	6
sp	7
mem	e
io	f

Assembler to native code translation examples

Native code (binary code) values are represented as hexadecimal numbers in this table

Operation	Assembler coder	Native code & explanation
Load the value 10 into register r0	`load 0x0a, r0;`	`8002 000a` instruction code (enabled) = 80 source = 0, destination = 2 data argument = 000a
Copy a value from the memory location currently selected by the *addr* register into register r3	`copy mem, r3;`	`84e5` instruction code (enabled) = 84 source = e, destination = 5
Call the program function at address a048	`call 0xa048;`	`8200 a048` instruction code (enabled) = 82 no source/destination specification address argument a048
Divide the value of register r0 by the value of register acc (result saved in register r0)	`div acc, r0;`	`8902` instruction code (enabled) = 89 source = 0, destination = 2
Return from a function call, stop execution before this instruction (breakpoint)	`ret;`	`0100` instruction code (disabled) = 01 no source/destination specification