SUMMARY OF THE TTK-91 INSTRUCTION SET

This document is also available in finnish

In the symbolic machine language, the commands are of following form:

```
LABEL OPER Rj,M ADDR(Ri)

where  OPER    the symbolic name of the command
       Rj      the first operand (register R0..R7)
       M       addressing mode:
               =   immediate operand
                   direct addressing (empty, ie. not marked)
               @   indirect addressing
       ADDR    address part (memory address or immediate value)
       Ri      possible index register (register R0..R7)
```

If some part of the instruction is not significant, it can be left out. It is possible to add a label (symbolic address) in front of a command. The label may contain letters A-Ö, numbers 0-9 and an underscore _. The label must contain at least one non-numerical character. Only the first 8 characters are significant.

Almost all commands have the following forms available:

```
  OPER Rj,ADDR        direct memory addressing
  OPER Rj,Ri          direct register addressing
  OPER Rj,=ADDR       immediate operand
  OPER Rj,@ADDR       indirect memory addressing
  OPER Rj,@Ri         indirect memory addressing
  OPER Rj,ADDR(Ri)    indexed addressing
  OPER Rj,=ADDR(Ri)   indexed immediate operand
  OPER Rj,@ADDR(Ri)   indexed indirect memory addressing
```

Exceptions:

STORE 2nd operand is always destination address, it cannot be a register or constant

POP 2nd operand must always be a register

PUSHR 2nd operand is not significant

POPR 2nd operand is not significant

NOT 2nd operand is not significant

NOP Neither operand is significant

Branches 2nd operand is always the target address, it cannot be a constant. In the jump commands which examine the state register, the first operand is ignored.

TTK-91 symbolic machine language instructions

Data transfer instructions:

LOAD take the value of the 2nd operand and makes it the new value of register Rj

STORE store the value of Rj to the address specified by the 2nd operand

IN read an integer to register Rj from a peripheral appointed in the 2nd operand (from keyboard for example IN R1,=KBD)

OUT print the value of register Rj to the peripheral appointed in the 2nd operand (to screen for example OUT R1,=CRT)

Arithmetic ja logic instructions:

ADD (add) add the value of the 2nd operand to the value of register Rj

SUB (subtract) substract the value of the 2nd operand from the value of Rj

MUL (multiply) multiply the value of Rj by the the 2nd operand.

DIV (divide) divide the value of register Rj with the 2nd operand and store the result in Rj

MOD (modulo) divide the value of register Rj with the 2nd operand and store the remainder in Rj

AND (boolean AND) logical AND-operation.

OR (boolean OR) logical OR-operation.

XOR (boolean XOR) logical exclusive OR-operation.

NOT (boolean NOT) invert all bits in register Rj.

SHL (shift left) shift register Rj's bits left the amount stated in the 2nd operand. Fill the right end with 0-bits.

SHR (shift right) like SHL, but shift right.

SHRA (arithmetic shift right) perform an arithmetic right shift; as in SHR, but fills the left end with copies of the leftmost bit, thus keeping negative numbers negative.

COMP (compare) compare the value of the 1st operand to the value of the 2nd operand and set the result of the comparison to the bits of state register SR (L=less, E=equal, G=greater).

Branching instructions:

JUMP Unconditional jump to the target address stated in the 2nd operand.

JNEG (jump if negative) if Rj < 0, jump to the address stated in the 2nd operand, otherwise continue to the next instruction.

JZER (jump if zero) if Rj = 0

JPOS (jump if positive) if Rj > 0

JNNEG (jump if not negative) if Rj >= 0

JNZER (jump if if not zero) if Rj <> 0

JNPOS (jump if not positive) if Rj <= 0

JLES (jump if less) if the state register SR's bit L is set, jump to the address stated in the 2nd operand, otherwise continue from the next instruction (used with the COMP-instruction).

JEQU (jump if equal) if bit E is set

JGRE (jump if greater) if bit G is set

JNLES (jump if not less) if bit E or G is set

JNEQU (jump if not equal) if bit L or G is set

JNGRE (jump if not greater) if bit L or E is set

Stack instructions:

The first operand of the instructions, Rj, points to the cell on top of the stack. Usually the register SP (ie R6) is used as stack pointer.

PUSH Increase the value of stack pointer Rj by one and store the 2nd operand as the top cell of the stack.

POP Remove the top cell from stack and take it to the register stated in the 2nd operand (NOTICE: always a register). Reduce the value of stack pointer Rj by one.

PUSHR Push registers R0, R1, R2, R3, R4, R5 and R6 (SP) to the stack, in this order. Before pushing each register, increases the stack pointer Rj's value by one.

POPR Pop values to the registers R6 (SP), R5, R4, R3, R2, R1 and R0, in this order, from the stack. For each pop, first fetches the value from the top of the stack indicated by the register Rj, and then substracts one from Rj.

Subroutine instructions:

CALL Call a subroutine, ie transfer control to the address stated with the 2nd operand. Prior to transfering control to the subroutine, CALL will automatically push the return address and current frame pointer (FP, ie R7) to the stack. Rj points to the top of the stack.

EXIT Return from a subroutine to the next instruction. Restores FP and PC from the stack, to which Rj is pointing to. The 2nd operand is the number of the stack parameters passed to the subroutine (they will be removed from the stack).

System calls:

SVC (supervisor call) call an operanting system's service routine. The first operand Rj is the top of the stack and the second operand is number of service. See service numbers below:

- **HALT** Ends program execution.
- **TIME** Gives the time. Addresses transmitted with stack, where the hours, minutes and seconds are desired. (NOTICE: order!).
- **DATE** Gives the date. Addresses transmitted with stack, where the day, month and year are desired. (NOTICE: order!).
- **READ** Reads an integer. Address transmitted with stack, where the integer is desired to be read.
- **WRITE** Writes an integer. A printable value transmitted with stack.

Other:

NOP (no operation) does nothing, neither operand is significant

Compiler control instructions (fake instructions)

The compiler control instructions give instructions to the assembler's compiler. They ARE NOT actual instructions of the assembler.

label **EQU** value

The equation instruction EQU defines an integer value to a symbolic label. The label can be used in an instruction's ADDR-field, after which it will be handled like a "value" typed in the same place.

label **DC** value

Memory allocation instruction DC (data constant) reserves one

memory word for a constant, equates the address of the allocated memory cell and the symbolic address "label" and sets the number "value" as the contents of the reserved memory cell. The label can be used in an instruction's ADDR-field like a memory address.

label **DS** size

The data reservation instruction DS (data segment) allocates a memory block of specified size (in words) and equates the starting address of the allocated memory cell and the symbolic address "label". Is used for allocating space for global variables. A label can be used in an instruction's ADDR-field like a memory address.

option **DEF** string *(not available in TitoTrainer)*

This special instruction changes options for simulating the file system of a TTK-91 machine. 'String' should be an absolute directory path. Examples:

```
STDIN DEF /home/myuser/ttk91/stdin
STDOUT DEF C:\mydir\stdout
```

Available options are:

STDIN To set which file stdin data is read from (default is file `stdin` in the user's current working directory)

STODUT To set which file stdout data is written to (default is file `stdout` in the user's current working directory)

The DEF instruction is disabled in TitoTrainer for security reasons

---

Addressing modes

Immediate Operand

In immediate addressing the second operand is a constant embedded in the instruction word. Immediate operand is a signed 16-bit integer so the range of possible values is -32768...32767.

Example:

```
1) LOAD R1, =100        Load the number 100 to register R1.
```

Direct Addressing

Direct addressing is a scheme in which the address specifies which memory word or register contains the operand.

Examples:

```
2) LOAD R1, 100         Load the content of memory address 100 to register R1.
3) LOAD R1, R2          Load the content of register R2 to register R1.
```

Indirect Addressing

Indirect addressing is a scheme in which the address specifies which memory word or register contains not the operand but the address of the operand.

Examples:

```
4) LOAD R1, @100        Load the content of memory address stored at
                        memory address 100 to the register R1.
5) LOAD R1, @R2         Load the content of the memory address stored at
                        register R2 to register R1.
```

Indexed Addressing

Addresses have two parts: the number of an index register and a constant. The address of the operand is the sum of the constant and the contents of the index register. It contains indexed (direct) addressing, indexed immediate addressing and indexed indirect addressing.

Examples:

```
6) LOAD R1, 100(R2)     Load the content of the memory address which is
                        the sum of 100 and the content of register R2 to
                        register R1.
7) LOAD R1, =100(R2)    Load the sum of 100 and the value of register R2
                        to register R1.
8) LOAD R1, @100(R2)    Load the content of the memory address stored at
                        the memory address which is the sum of 100 and
                        the number in register R2 to the register R1.
```

Hardware addressing modes

It is worth noting that in actuality the TTK-91 instruction set has only three adressing modes instead of the eight presented above. In compiled code all adressing uses indexing and constants. Although you can omit the index register or the constant from your assembly code, the compiler automatically adds them to the compiled binary. For example:

```
"LOAD R1, =10"  ==  "LOAD R1, =10(R0)"
"LOAD R1, R2"   ==  "LOAD R1, =0(R2)"
"LOAD R1, @R2"  ==  "LOAD R1, 0(R2)"
"LOAD R1, @10"  ==  "LOAD R1, @10(R0)"
"LOAD R1, 10"   ==  "LOAD R1, 10(R0)"
```

In the above examples, note the use of register R0. When used as index register, the value of R0 is always zero. You can assign other values to R0 and perform ALU operations on it, but when used in

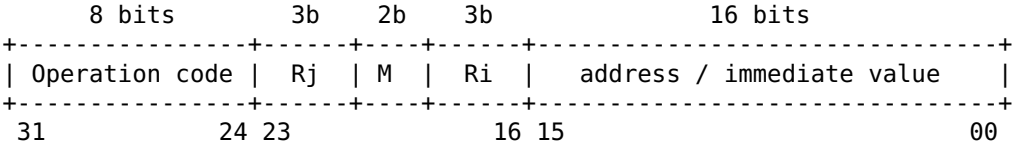
the second operand it is always treated as zero. When used as the *first* operand R0 behaves like any other register. For example consider the failed attempt to copy R0 with LOAD, and the workaround to this limitation:

```
a) LOAD R3, R0          Assign 0 to R3, regardless of the value of R0.

b) STORE R0, X          Store the value of R0 to memory location X
   LOAD R3, X           Load the value of memory location X to R3
```

TTK-91 instruction binary format

Instruction word layout



Operation codes

Operation	Binary	Decimal	Hexdecim.
NOP	0000 0000	0	00
STORE	0000 0001	1	01
LOAD	0000 0010	2	02
IN	0000 0011	3	03
OUT	0000 0100	4	04
ADD	0001 0001	17	11
SUB	0001 0010	18	12
MUL	0001 0011	19	13
DIV	0001 0100	20	14
MOD	0001 0101	21	15
AND	0001 0110	22	16
OR	0001 0111	23	17
XOR	0001 1000	24	18
SHL	0001 1001	25	19
SHR	0001 1010	26	1A
NOT	0001 1011	27	1B
SHRA	0001 1100	28	1C
COMP	0001 1111	31	1F
JUMP	0010 0000	32	20
JNEG	0010 0001	33	21
JZER	0010 0010	34	22
JPOS	0010 0011	35	23
JNNEG	0010 0100	36	24
JNZER	0010 0101	37	25
JNPOS	0010 0110	38	26
JLES	0010 0111	39	27
JEQU	0010 1000	40	28
JGRE	0010 1001	41	29

JNLES	0010 1010	42	2A
JNEQU	0010 1011	43	2B
JNGRE	0010 1100	44	2C
CALL	0011 0001	49	31
EXIT	0011 0010	50	32
PUSH	0011 0011	51	33
POP	0011 0100	52	34
PUSHR	0011 0101	53	35
POPR	0011 0110	54	36
SVC	0111 0000	112	70

Addressing modes

Binary	Dec	Addressing mode
00	0	indexed immediate
01	1	indexed direct
10	2	indexed indirect