Assembly Language, Abstract Machine, and Instruction Set

Assembly language definition

In the assembly language there is no distinction between upper-case and lower-case letters.

A program consists of a sequence of lines, as follows.

```
<line> ::=
     ( <label> ':' )? <instruction>? ( ';' <comment> )?
```

Every line has three optional parts: an instruction, a label, followed by '`:`', and a comment, preceded by '`;`'.

```
<instruction> ::=
     <opcode> ( <operand> ( ','? <operand> )? ( ','? <operand> )? )?
```

An instruction is an opcode with between 0 and 3 operands, separated by spaces or commas.

```
<operand> ::=
       <constant>
     | <register>
     | <label>
```

Each operand may be at most 10 characters long.

A <constant> is any (possibly signed) integer or real constant, as accepted by the `"%d"` or `"%f"` format of C. E.g., `1234` or `-9.876`.

A <register> is the letter '`R`' or '`r`' followed by one or more digits. E.g., `R1` or `r987654321`.

A <label> is any alphanumeric string that is not a register or a constant. Every label used as an operand must be defined, by appearing as a label on some line, and no label may be defined more than once. A label refers to the instruction on the same line, or to the next instruction if there is no instruction on the same line as the label. This makes it possible for more than one label to refer to the same instruction.

The possible opcodes, and their operands, are listed in the instruction table below.

Abstract machine and instruction set

The abstract machine is based on the *Jouette* architecture described in Chapter 9 of the Appel book, with the following differences and refinements:

- The machine has an infinite number of registers (actually 1000000000 of them, because of the limited operand length). It is quite possible to use registers `R101`, `R123456789`, etc., but probably more sensible to use `R0`, `R1`, etc.
- `R0` is not initialized to 0 as described in the book. You need to explicitly set `R0` to 0 if you want to use it this way, e.g., by `XOR R0,R0,R0`.

- Real (floating point) versions of most instructions are included.
- All integer and real numbers are 32-bit.
- Memory addresses used by `LOAD` and `STORE` are byte addresses, which must be a multiple of 4. These instructions move 4 bytes at a time, regardless of type.
- The new instruction `WRS` (to print a string) refers to the memory address of a string terminated by a 0 byte.
- Memory addresses (used by `LOAD`, `STORE`, and `WRS`) start at 0. Memory to be accessed by these instructions must be allocated by the pseudo-instruction `DATA`.

The following table shows the opcodes and operands of all instructions. `I` means an integer constant. `F` means a real constant. `Ri`, `Rj`, and `Rk` represent registers, and `L` represents a label.

Instruction	Effect	Comments	Reference
`ADD Ri,Rj,Rk`	`Ri <- Rj + Rk`	Integer addition	Appel, p177
`SUB Ri,Rj,Rk`	`Ri <- Rj - Rk`	Integer subtraction	Appel, p177
`MUL Ri,Rj,Rk`	`Ri <- Rj * Rk`	Integer multiplication	Appel, p177
`DIV Ri,Rj,Rk`	`Ri <- Rj / Rk`	Integer division	Appel, p177
`XOR Ri,Rj,Rk`	`Ri <- Rj ^ Rk`	Bitwise XOR	New
`ADDR Ri,Rj,Rk`	`Ri <- Rj + Rk`	Real addition	New
`SUBR Ri,Rj,Rk`	`Ri <- Rj - Rk`	Real subtraction	New
`MULR Ri,Rj,Rk`	`Ri <- Rj * Rk`	Real multiplication	New
`DIVR Ri,Rj,Rk`	`Ri <- Rj / Rk`	Real division	New
`ADDI Ri,Rj,I`	`Ri <- Rj + I`	Integer addition: register and constant	Appel, p177
`SUBI Ri,Rj,I`	`Ri <- Rj - I`	Integer subtraction: register and constant	Appel, p177
`MULI Ri,Rj,I`	`Ri <- Rj * I`	Integer multiplication: register and constant	New
`DIVI Ri,Rj,I`	`Ri <- Rj / I`	Integer division: register and constant	New
`XORI Ri,Rj,I`	`Ri <- Rj ^ I`	Bitwise XOR: register and constant	New
`MOVIR Ri,F`	`Ri <- F`	Real constant moved to register	New
`ITOR Ri,Rj`	`Ri <- Rj`	Integer to real conversion (`Rj` is integer; `Ri` is real)	New
`RTOI Ri,Rj`	`Ri <- Rj`	Real to integer conversion (`Rj` is real; `Ri` is integer)	New
`RD Ri`	`Read Ri`	Reads integer from stdin	New
`RDR Ri`	`Read Ri`	Reads real from stdin	New
`WR Ri`	`Write Ri`	Writes integer to stdout	New

WRR Ri	Write Ri	Writes real to stdout	New
WRS I	Write M[I]...	Writes string (from address I to next 0 byte) to stdout	New
LOAD Ri,Rj,I	Ri <- M[Rj + I]	Loads memory contents to register	Appel, p177
STORE Ri,Rj,I	M[Rj + I] <- Ri	Stores register contents in memory	Appel, p177
JMP L	goto L	Jumps to label L	New
JUMP Ri	goto Ri	Jumps to the instruction whose address is stored in the register	Appel, p201
IADDR Ri,L	Ri <- L	Store address L in the register	New
BGEZ Ri,L	if Ri ≥ 0 goto L	If register's contents (integer) non-negative jump to L	Appel, p201
BGEZR Ri,L	if Ri ≥ 0 goto L	If register's contents (real) non-negative jump to L	New
BLTZ Ri,L	if Ri < 0 goto L	If register's contents (integer) negative jump to L	Appel, p201
BLTZR Ri,L	if Ri < 0 goto L	If register's contents (real) negative jump to L	New
BEQZ Ri,L	if Ri = 0 goto L	If register's contents (integer) zero jump to L	Appel, p201
BEQZR Ri,L	if Ri = 0 goto L	If register's contents (real) zero jump to L	New
BNEZ Ri,L	if Ri ≠ 0 goto L	If register's contents (integer) non-zero jump to L	Appel, p201
BNEZR Ri,L	if Ri ≠ 0 goto L	If register's contents (real) non-zero jump to L	New
NOP		No operation	New
HALT		Stop execution	New
DATA I		A pseudo-instruction. Used by the assembler to allocate one byte in data memory initialized to the value I (in range 0..255).	New

Department of Computer Science

University of Bristol
Department of Computer Science
Merchant Venturers Building
Woodland Road
Bristol BS8 1UB UK
+44 (0)117 331 5663