The BigHex Machine Assembly Specification

Samuel Russell

February 20, 2017

1 Design Considerations

- The assembly language is designed to be very close to the actual machine instructions, with the assembler only handling the layout of the instructions and the data in memory.
- The language is relatively flexible in terms of syntax and should be quite consistent with other languages.

2 Labels

Labels must label an instruction. They are written with an identifier followed by a colon. The identifier can be include letters, numbers, '-' and '_'. They can be on the same line or above the instruction.
Example

```
loop_start:
  INSTRUCTION OPERAND
  INSTRUCTION OPERAND
loop-end : INSTRUCTION OPERAND
```

An address can be specified in square brackets as so...

```
program_start[0x10]:
  INSTRUCTION OPERAND
next-bit [20] :  INSTRUCTION OPERAND
```

If an address is specified, the assembler will place that label and the following instructions at said location by padding with zeros from the normal location. The assembler does not do any clever reorganisation of code, so the address must be larger than they would be normally and they will push all subsequent code up as well.

You can define more than one label one after another and they will be given the same value (unless the second has an address specified. Although that would produce weird behaviour and should not be done).

Good Example:

```
program_start:
loop_start:
  INSTRUCTION OPERAND
```

3 Instructions

All instructions have 1 operand like the machine's actual instructions. In fact most of the instructions are exactly the instructions of the machine. Instructions are represented by...

1. then the acronym (capital letters),
2. then at least one space,
3. then the operand,
4. then a new line (with no trailing spaces).

Eg.

```
BR 1
```

or

```
OPR ADD
```

The instructions are as follows (taken from the machine code specification). The oreg value is provided by the operand.

Access to constants and program addresses is provided by instructions which either load values directly or enable them to be loaded from a location in the program:

```
LDAM: areg <- mem[oreg] load from memory
LDBM: breg <- mem[oreg] load from memory
STAM: mem[oreg] <- areg store to memory

LDAC: areg <- oreg load constant
LDBC: breg <- oreg load constant
LDAP: areg <- pc + oreg load address in program
```

Access to data structures is provided by instructions which combine an address with an offset:

```
LDAI: areg <- mem[areg + oreg] load from memory
LDBI: breg <- mem[breg + oreg] load from memory
STAI: mem[breg + oreg] <- areg store to memory
```

Branching, jumping and calling The branch instructions include conditional and unconditional relative branches. A branch using an offset in the stack is provided to support jump tables.

```
BR: pc <- pc + oreg branch relative unconditional
BRZ: if areg = 0 then pc <- pc + oreg branch relative zero
BRN: if areg < 0 then pc <- pc + oreg branch relative negative
BRB: pc <- breg branch absolute
```

Inter-register operandless instructions using OPR

```
OPR ADD: areg <- areg + breg add the registers values
OPR SUB: areg <- areg - breg subtract the registers values
```

Operands

Operands can be written as either:

- decimal format with negative value being prefixed by a '-' sign.
- unsigned hexadecimal format prefixed by '0x' to indicate this.
- the OPR command's sub instructions can be written by using the mnemonics ADD and SUB.

Data Sections

Slightly differently, to define a data word (16bits) to be stored to memory. we use

```
DATA {initial value of the 16bit word}
```

It is suggested you label this word with a label and refer to it using that since the assembler might move it (for example to align it to fit in 1 16 bit word) See example in code snippet below.

Using labels as Operands for Instructions

The following instructions also accept labels as operands.

`LDAM, LDBM, STAM, LDAC, LDBC, BR, BRZ, BRN, LDAP`

Instead of writing a static number you can signify a label by typing the identifier.

- Use with LDAM, LDBM and STAM will load and store the data to/from the address of the label, be it a Data Section or 2 instructions.
- Use with LDAC and LDBC will load the address of the label in to the register. This might be the base of an array which you could then use for memory access with indirection.
- For use with BR, BRZ, BRN and LDAP the assembler calculates the relative jump to the specified label and use that in the machine code.

4 Comments

Any line that starts with '#' will be ignored so can be used for comments. Eg.

```
#this is a comment
```

5 Example Program

```
#This program maintains a stack while calling a function that returns 5
 BR       L3
sp:
 DATA     0x7CFE
L3:
 LDAP     L5
 BR       L4
L5:
 LDAI     1
 BR       -2
L4:
L2:
 LDBM     sp
 STAI     0
 LDAC     -1
 OPR      ADD
 STAM     sp
 LDAC     5
 LDBM     sp
 STAI     2
 LDAC     1
 OPR      ADD
 STAM     sp
 LDBI     1
 BRB      0
```

6 Notes

- The assembler doesn't like trailing white-space.
- BR -1 probably won't do what you think, since the negative value needs an extra instruction to produce it. (So should use -2)

- You can write new-lines and comments between lines of code, but not between a label and the line of code it is labelling.
- Due to needing prefixes for large values/addresses it is best to place data sections in the low parts of memory to reduce code size.