The Big Hex Machine Assembly Worksheet

Samuel Russell

February 22, 2017

This document is available online at
github.com/bighexmachine/BigHexMachineAssembly/raw/master/Worksheet/bigHexMachineWorksheet.pdf
or tinyurl.com/bighexworksheet

1 Introduction to the Big Hex Machine

The Big Hex Machine is a two register machine with 32 kilobytes of RAM memory. However, it only supports 2 arithmetic operations: add and subtract. The other instructions move data to and from and registers, RAM and IO ports. However it is a bit big to have one each to play with. However, we have also build a on-line simulator so you can try out writing programs for it!

2 Information about the Instruction Set

The the machine has 16 instructions, that each take one operand (a parameter), although they might also use the data stored in the registers. One of the instructions is split in to 4 different instructions (each without an operand) by giving one of 4 input operands. Don't worry you won't need all of them and we'll take you through the ones you do but if your interested here is the specification.

For the following exercises X is a number written in decimal like 182 (2 units, 8 tens, 1 hundred) or in hexadecimal like 0x298 (8 units, 9 sixteens, 2 sixteen-squareds).

We write an instruction with a it's short name in captitals and then a space and then the operand/input value. For example to branch (move) one step forward in the program.

```
BR 1
```

3 Exercise 0

Open up a web browser, and visit tinyurl.com/bighexmachine. A page will appear that has 4 different sections:

- A view of the memory. Each word (addressable slot) in memory is represented by 4 hexadecimal digits.
- A view of the register values. The RA (register A) and RB (register B) are the important ones, but RP (program counter) also keeps track of where in the program you are.
- A program control panel. This is where you can start and stop your program. It might be useful to step through one instruction at a time to see what is going on. But do remember to reset each time to restart the program other wise the computer will try and go from where yo left of last time.
- A disassembler which translates the computer's binary in memory back to human readable format. You will see every row says LDAM just because that happens to be the instruction represented by 00.

Click on 'edit', in the memory section. On the left you can directly edit memory values, but we are going to use the area on the right to write our assembly programs.
When you have written some assembly code in the box you can click 'assemble' to translate the text in to machine code and then 'accept' to go back to the previous page to run and test your code. Read on to learn about how to write the assembly code!

4 Exercise 1: Inputting Values

These are your basic instructions to put values in to the registers.

```
LDAC X : Loads the value X in to register A
LDBC X : Loads the value X in to register B
```

Try writing something like the following, in the text box.

```
LDAC 10
LDBC 0xA
```

Then click 'assemble' to translate the text in to machine code and then accept to go back to the previous page to run and test your code. To do this click 'reset' then 'step' twice to execute each instruction. Notice the instructions in the disassembler and the values change in the register viewer. Try changing the above code the load different values in to the registers.

5 Exercise 2: Adding Two Constants

The OPR instruction is the instruction that is split in to 4 other instructions. One of these is ADD.

```
OPR ADD : Adds the value in A reg to the value in B reg and puts the result in A reg.
OPR SUB : Subtracts the value in A reg by the value in B reg and puts the result in A reg.
```

```
LDAC  1
LDBC  2
OPR    ADD
```

Again copy this in and have a play with different values and operations.

6 Exercise 3: Adding Two Values from Memory

These are some instructions to get values from memory.

```
LDAM X : Gets the value at memory address X and puts it in register A
LDBM X  : Gets the value at memory address X and puts it in register B
STAM X : Stores the value in register A in to memory at address X
```

We can explicitly allocate addresses in memory by the DATA statement. Furthermore, these data portions can be named using a label and so the address X can be replaced the label word. Here is an example. Give it a go and see how it behaves. Note labels are written with a colon at the end.

```
LDAM   val_one
LDBM   val_two
OPR    ADD
STAM   val_out

val_one: DATA    0x10
val_two: DATA    2
val_out: DATA    0
```

7 Exercise 3: Loops and conditions

This is where the real power of the computer comes; repeating stuff over and over, and making decisions!
This computer supports the unconditional branch (BR and BRB) which jumps in the program every time, and also conditional jumps (BRZ and BRN) which look at the value of the registers to decide to jump or not.

```
BR X : Jumps X+1 places forward in the program. (remember it normally jumps 1 ahead anyway).
BRZ X  : if A reg = 0 then it jumps as above but else jump goes to the next instruction like normal.
BRN X : if A reg < 0 then it jumps as above but else jump goes to the next instruction like normal.
BRB X : unconditionally jumps to position X.
```

Just like we can label data we can also label parts of the program. This makes jumping to the right place easier. Here's an example to try.

```
LDAC 10
start:
LDBC 1
OPR SUB
BRZ end
BR start
end:
BR end
```

8 Exercise 4: More Loops and conditions

You may have to many variables to fit in to registers. If this is the case you have to store them in memory. This is called spilling and is normally done all automatically.

```
start:
LDAM counter
LDBC 1
OPR SUB
STAM counter
BRZ end
BR start
end:
LDAM Lsum
counter:
DATA 10
```

9 Challenge 1: Summing

Using the code previously your challenge is to write a program to calculate the of sum the integers (whole numbers) from 1 to N. You can input the N using the LDAC/LDBC instructions.

10 Challenge 2: Multiplication

This machine has a very RISC architecture which means instructions can be decoded simply and the control logic is relatively small. This does mean however that there is no multiplication instruction and so it has to be done iteratively (this was true up until surprisingly recently).
Now can you create a program to multiple two numbers again setting the inputs with LDAC and/or LDBC.
Hint: This is quite similar to the previous code.

11 Challenge 2: Factorial

This is a classic mathematical problem. Remember n! = (n) * (n - 1)...(2) * (1).
We can then use this multiplication code we have written as a sub procedure (function) to write a program that calculates the factorial of a given number.