The BigHex Machine Assembly Examples

Samuel Russell

February 22, 2017

1 Adding Two Constants

First you must load the two constants in to the two registers (areg and breg) and then perform the interegister add operation.

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

The first line loads the number one in to areg, the second line loads two in to breg and the third adds together the value of areg and breg and puts the answer in to areg.

Note that each instruction goes on a new line and has at least one space at the beginning.

2 Adding Two Values from Memory

A data portion is allocated in memory and the address of it (using a label) is used to load the value in to the registers before the add operation (used as before) and store back to memory afterwards.

```
LDAM   val_one
LDBM   val_two
OPR    ADD
STAM   val_out

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

Note we can also use hexadecimal representation to define the values.

3 Summing the Integers from 1 to N

For this operation we will create a loop using the conditional and unconditional branch instructions. We will need to keep track of the running total and a variable to count the number of iterations. Since we only have two registers we must store these values in our main memory and use loads and stores as we manipulated them. We use labels to mark lines of code to jump to.

```
start:
LDAM sum
LDBM counter
OPR ADD
STAM sum
LDAM counter
LDBC 1
OPR SUB
STAM counter
BRZ end
BR start

end:
LDAM sum

counter:
DATA 10
sum:
DATA 0
```

This code can be made shorter by utilising the fact that the loop counter is already in breg when we need to decrement it.

```
start:
LDAM sum
LDBM counter
OPR ADD
STAM sum
LDAC -1
OPR ADD
STAM counter
BRZ end
BR start

end:
LDAM sum

counter:
DATA 10
sum:
DATA 0
```

4 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).

```
start:
LDAM sum
LDBM val_a
OPR ADD
STAM sum
LDAM val_b
LDBC 1
OPR SUB
STAM val_b
BRZ end
BR start

end:
LDAM sum

val_a:
DATA 0x2
val_b:
DATA 0x10
sum:
DATA 0
```

5 Factorial

We can then use this multiplication code we have written to write a program that calculates the factorial of a given number. (remember $n! = (n) * (n-1)...(2) * (1)$)

```
#jump straight to code
BR start
#Allocate space to store variables
val_a:
DATA 0x0
val_b:
DATA 0x0
mult_sum:
DATA 0x0
count:
DATA 4
total:
DATA 1

#this bit loads the parameters for our multiplier from our total variables
start:
LDAM total
STAM val_a
LDAM count
STAM val_b
LDAC 0
STAM mult_sum
# this is the call to multiplier
BR mult_start
#this bit receives the code from multiplier,
#then checks if we need to go round again.
check_finish:
LDAM mult_sum
STAM total
LDAM count
LDBC 1
OPR SUB
STAM count
BRZ end
BR start

#holds the processor in an infinite loop
end:
LDAM total
BR end

#this is the multiplier code as before
mult_start:
LDAM mult_sum
LDBM val_a
OPR ADD
STAM mult_sum
LDAM val_b
LDBC 1
OPR SUB
STAM val_b
BRZ mult_end
BR mult_start
#now we return to the previous code
mult_end:
BR check_finish
```

This piece of code calculates factorial 4. It works well but some improvements could be made to make it run faster.