Performance Comparison of Various Bit Width Operations

7/15/2012
Brandon Falk
Gamozo Labs

Summary

This document contains the performance analysis of numerous operations at various bit widths on an x86-64 system. The goal is to determine which bit widths should be used for optimum performance in certain scenarios.

Code

Throughout this document will be numerous performance tests on operations. The code used in these studies is as follows:

```
[bits 64]

section .code

global mainCRTStartup

mainCRTStartup:
      mov ecx, 100000
.lewp:
%rep 10000
      <operation>
%endrep
      dec ecx
      jnz .lewp

      ret
```

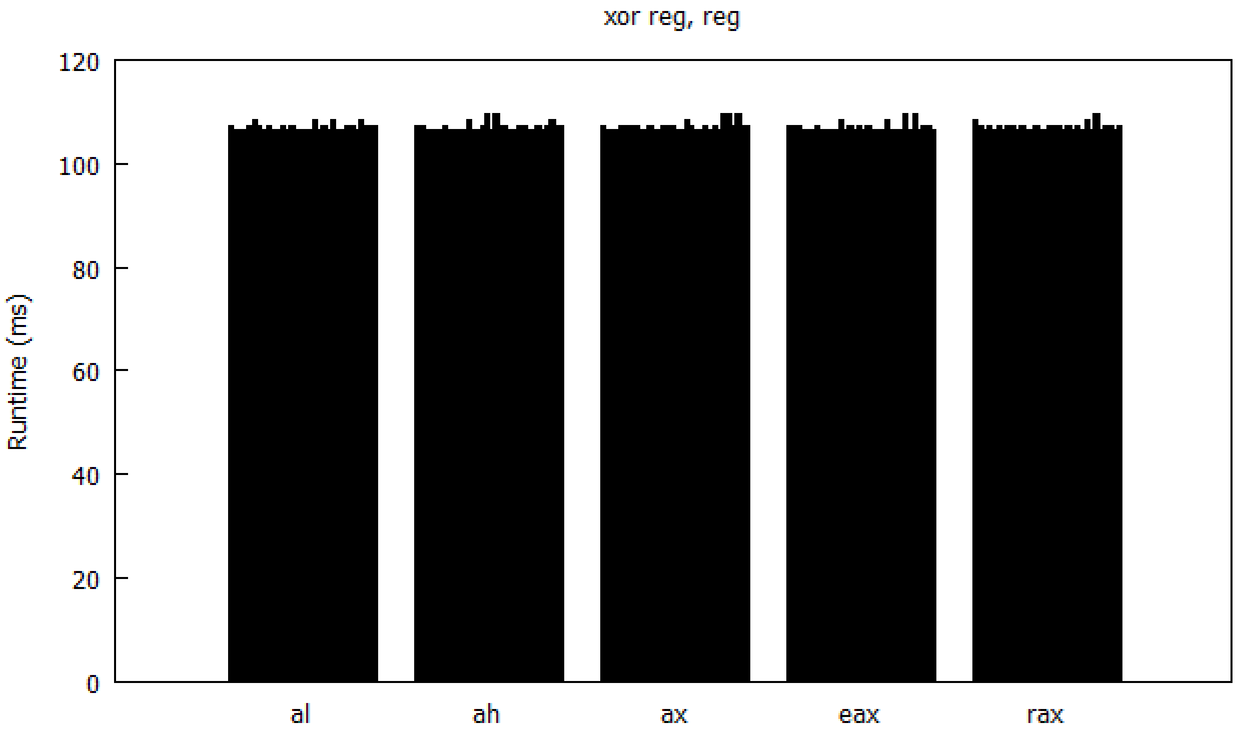
The operation is placed where `<operation>` is in the code. For those unfamiliar with NASM syntax, the `%rep` duplicates the contents between the `%rep` and `%endrep` for the number of times specified. Thus, `<operation>` is repeated 10,000 times, and then looped 100,000 times for a total of 1,000,000,000 operations.

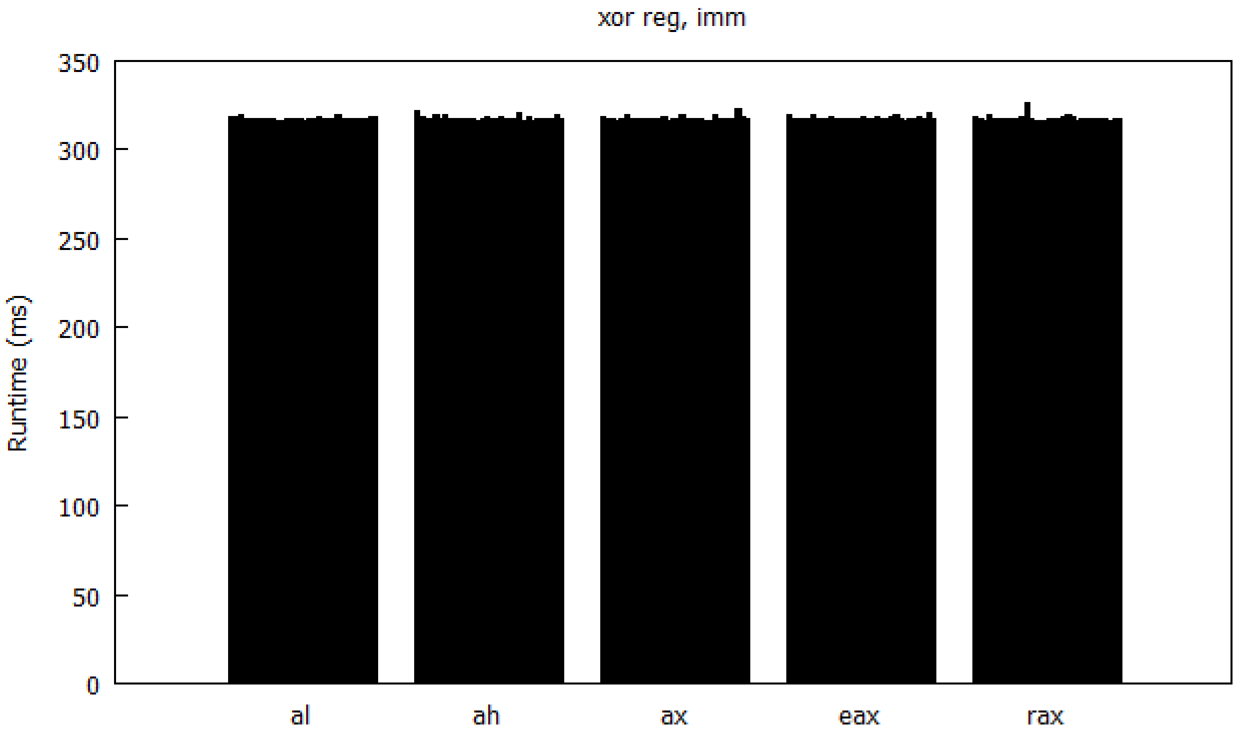
Benchmarking

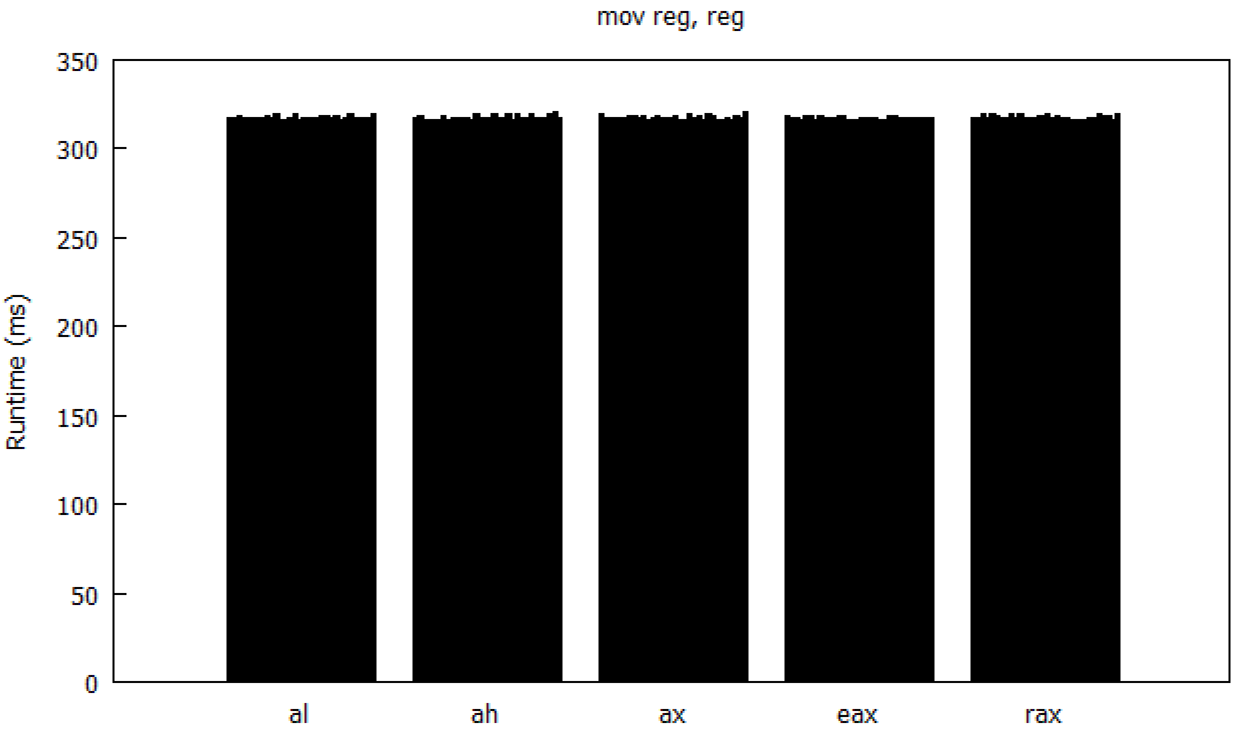
All of the benchmarking is done by my tool `putime` (svn co http://gamozolabs.com/putime). The statistic used to get the data in this document is the process run time. Each test is run 32 times to form the plot. Registers used are al, ah, ax, eax, and rax. All operations are done on an i7-970 CPU.

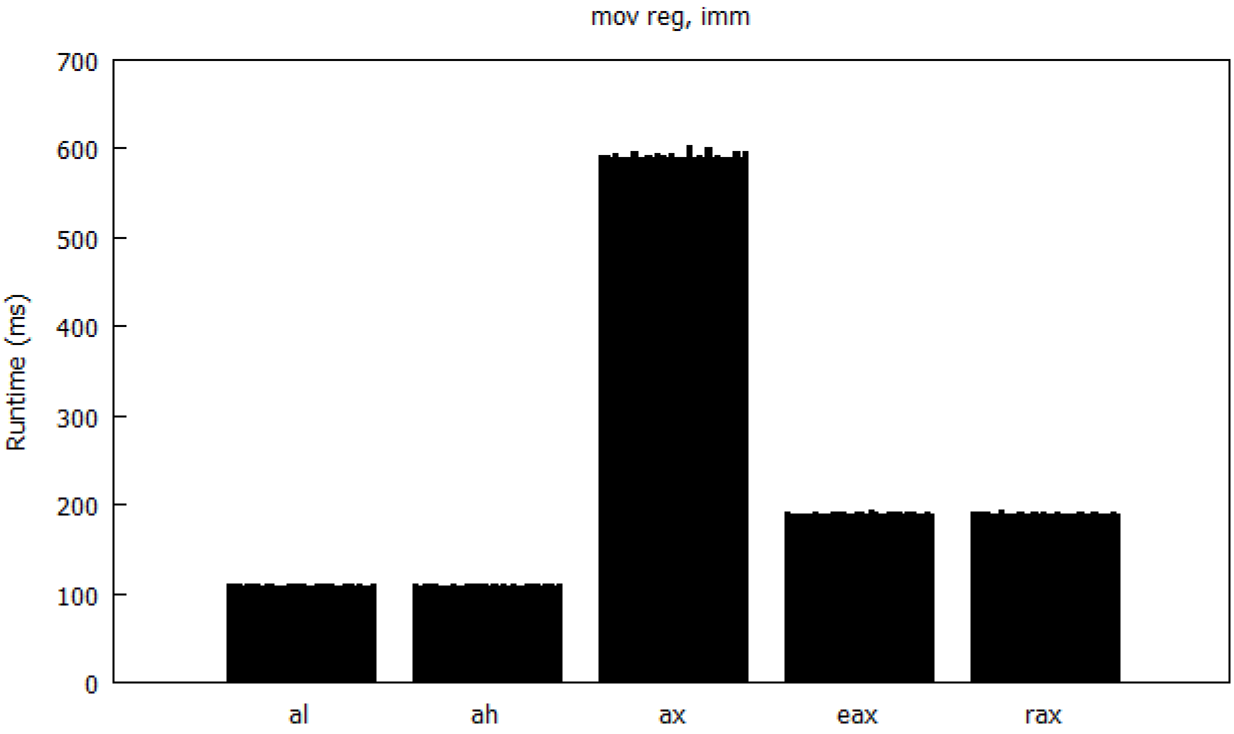
Coverage

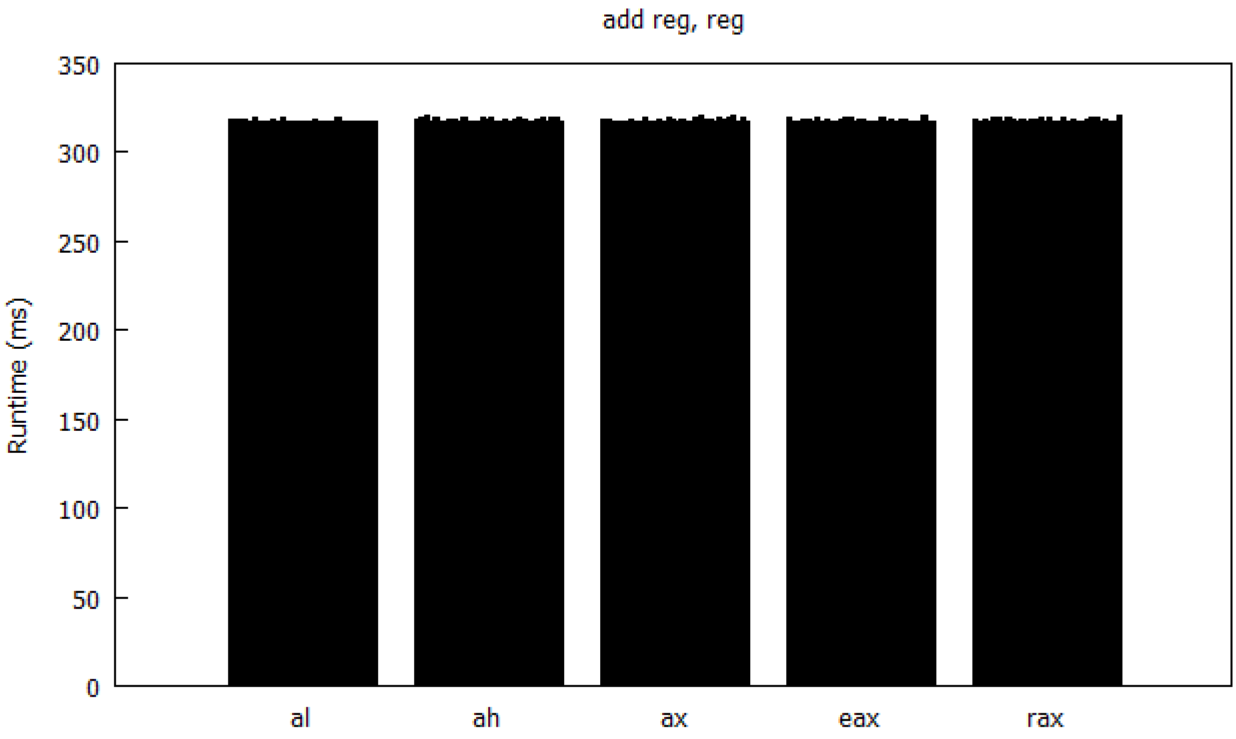
```
xor  reg, reg     shl  reg, 1
xor  reg, imm     ror  reg, 1
mov  reg, reg     test reg, reg
mov  reg, imm     test reg, imm
add  reg, reg     not  reg
add  reg, imm     neg  reg
and  reg, reg     cmp  reg, reg
and  reg, imm     cmp  reg, imm
inc  reg          mul  reg
dec  reg
```

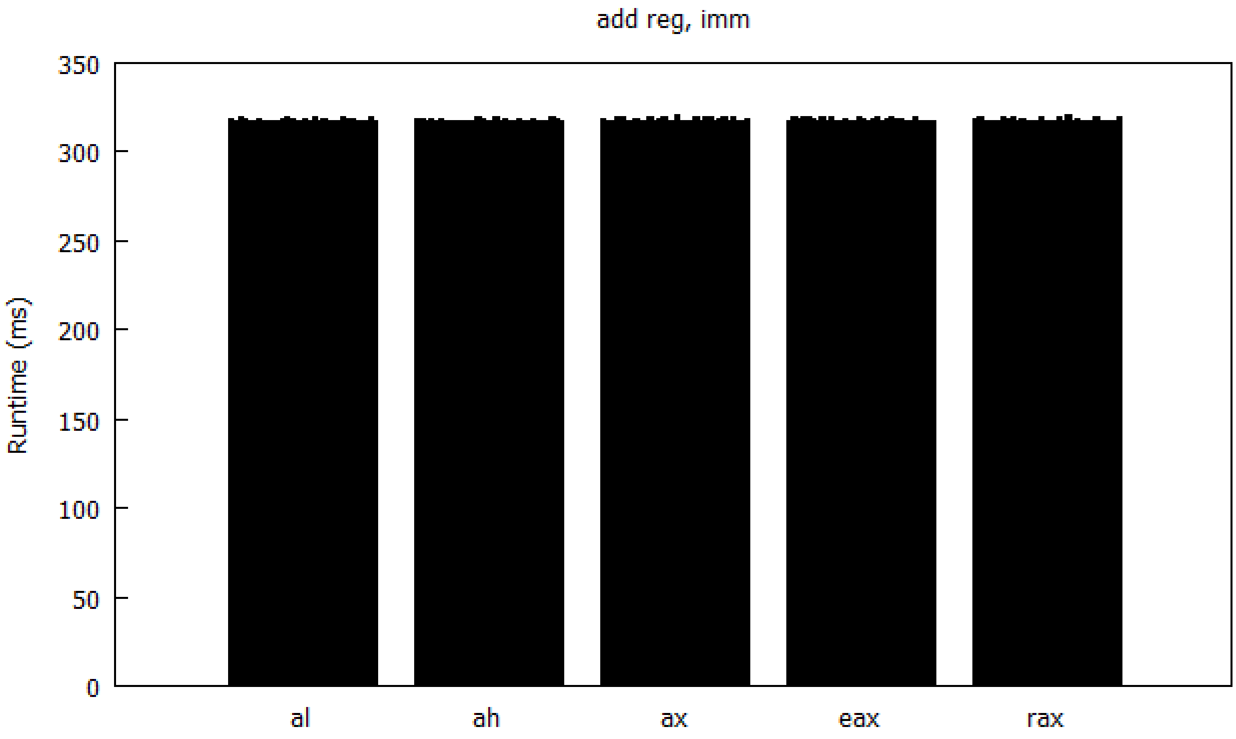

xor reg, reg
Runtime (ms)
0
20
40
60
80
100
120
al
ah
ax
eax
rax

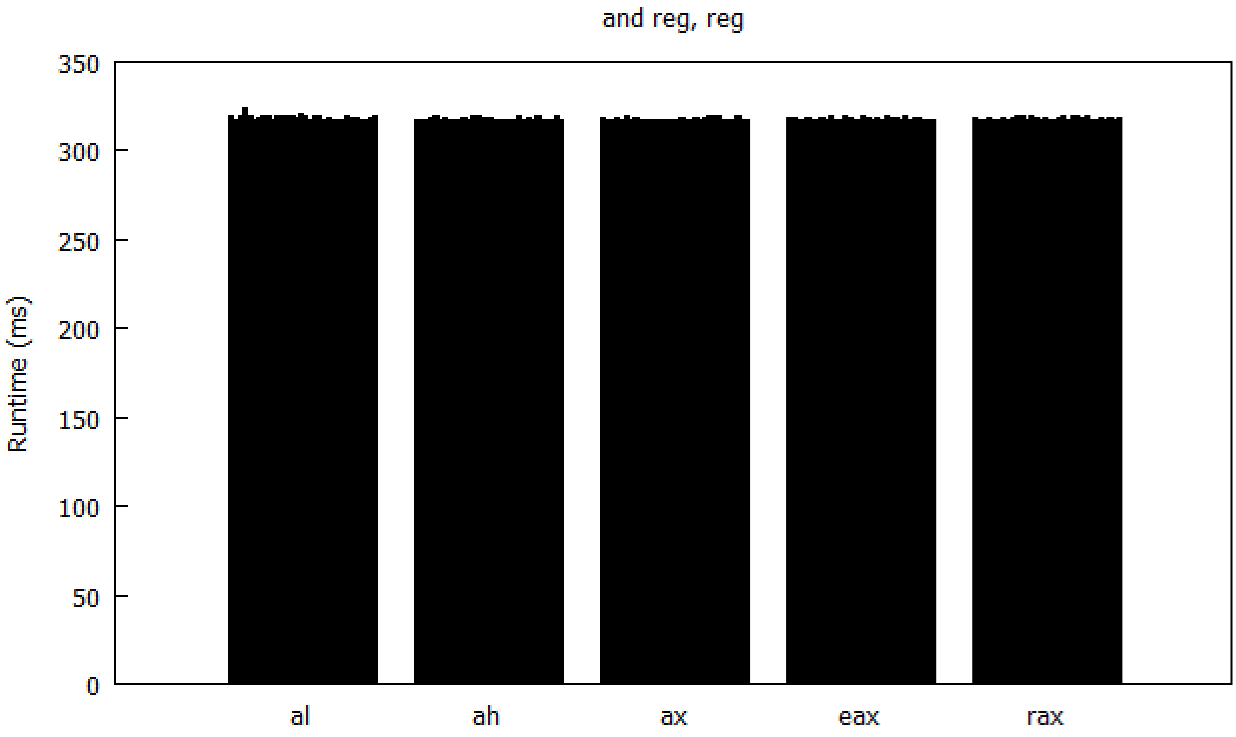

xor reg, imm
Runtime (ms)
0
50
100
150
200
250
300
350
al
ah
ax
eax
rax

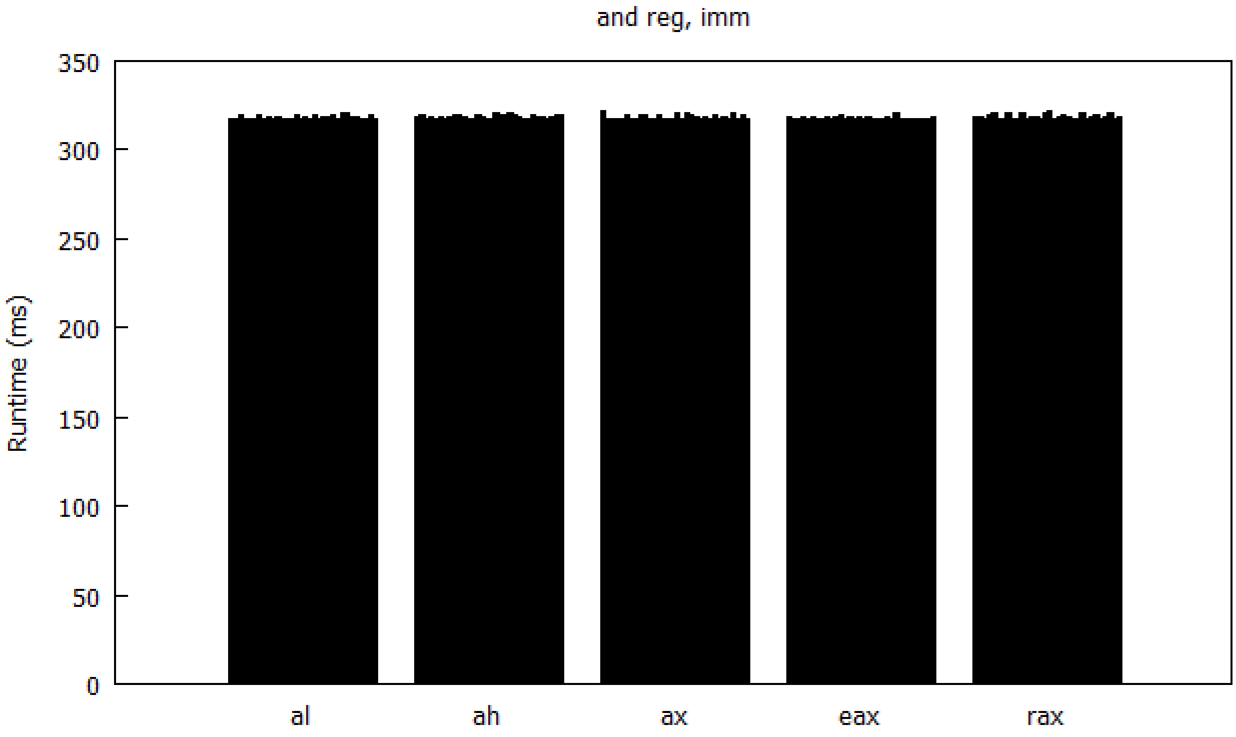

mov reg, reg
Runtime (ms)
0
50
100
150
200
250
300
350
al
ah
ax
eax
rax

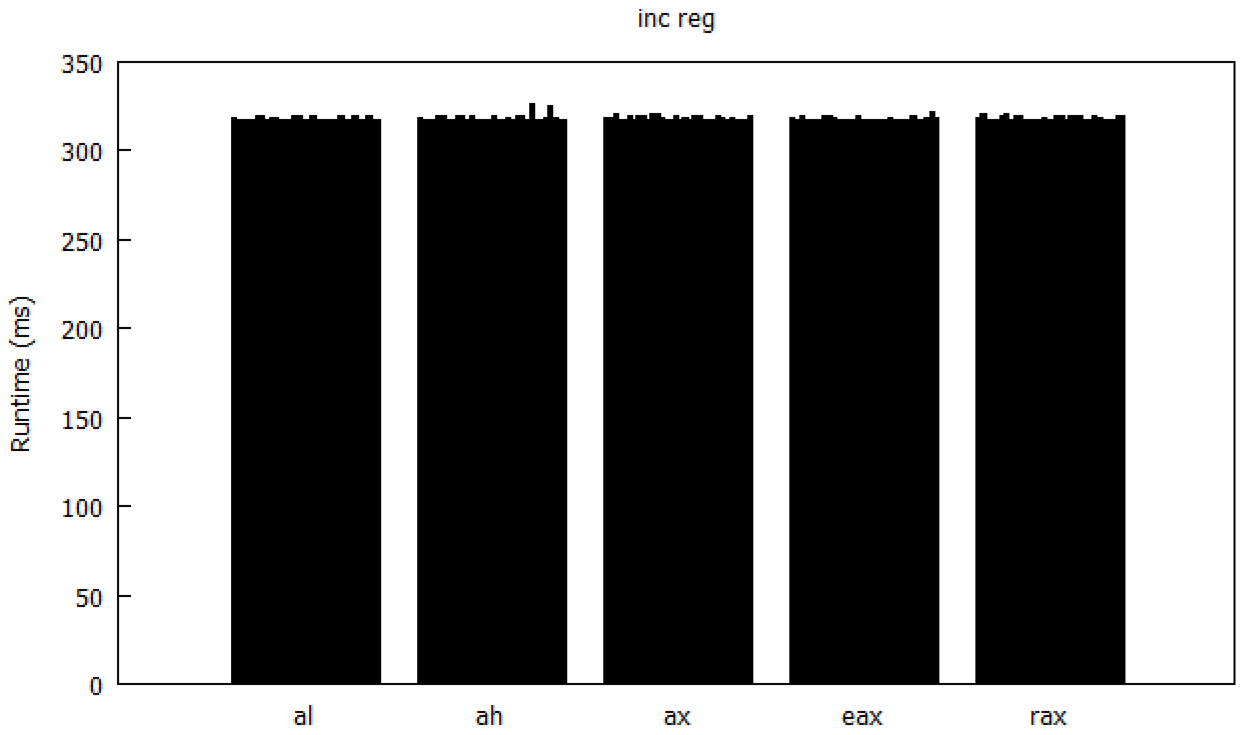

mov reg, imm
Runtime (ms)
0
100
200
300
400
500
600
700
al
ah
ax
eax
rax

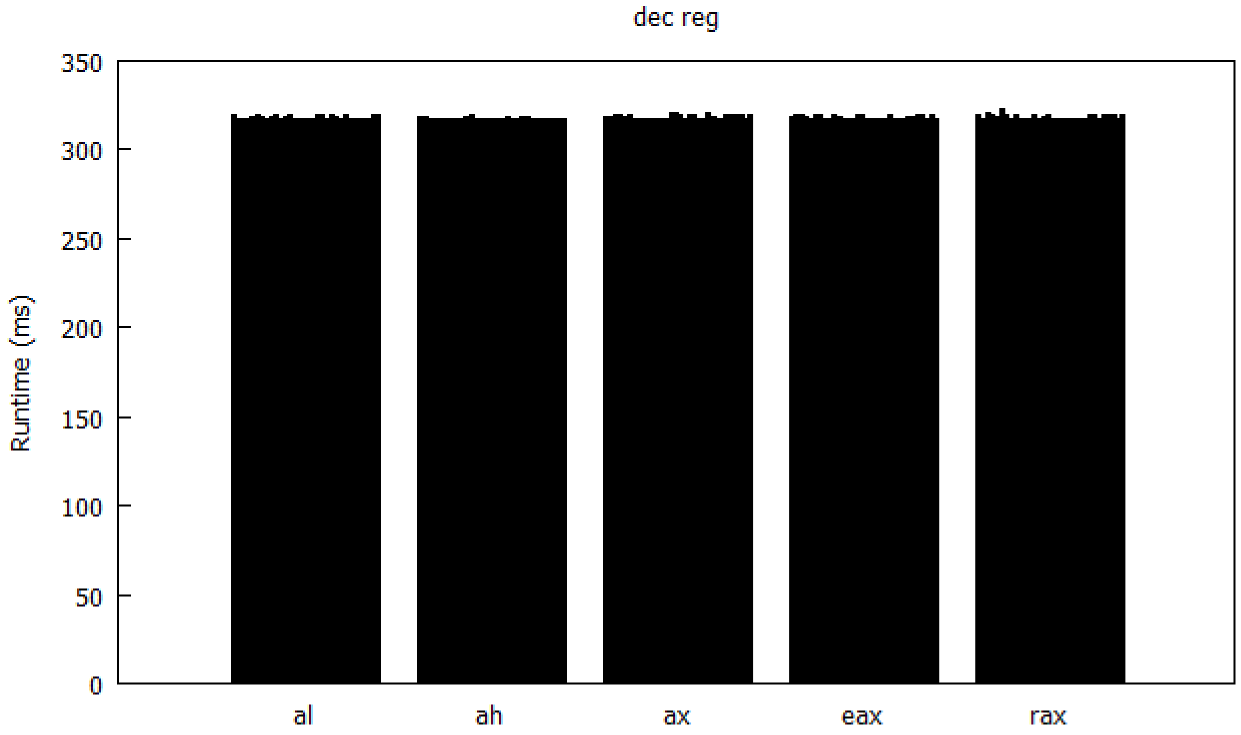

add reg, reg
Runtime (ms)
350
300
250
200
150
100
50
0
al
ah
ax
eax
rax

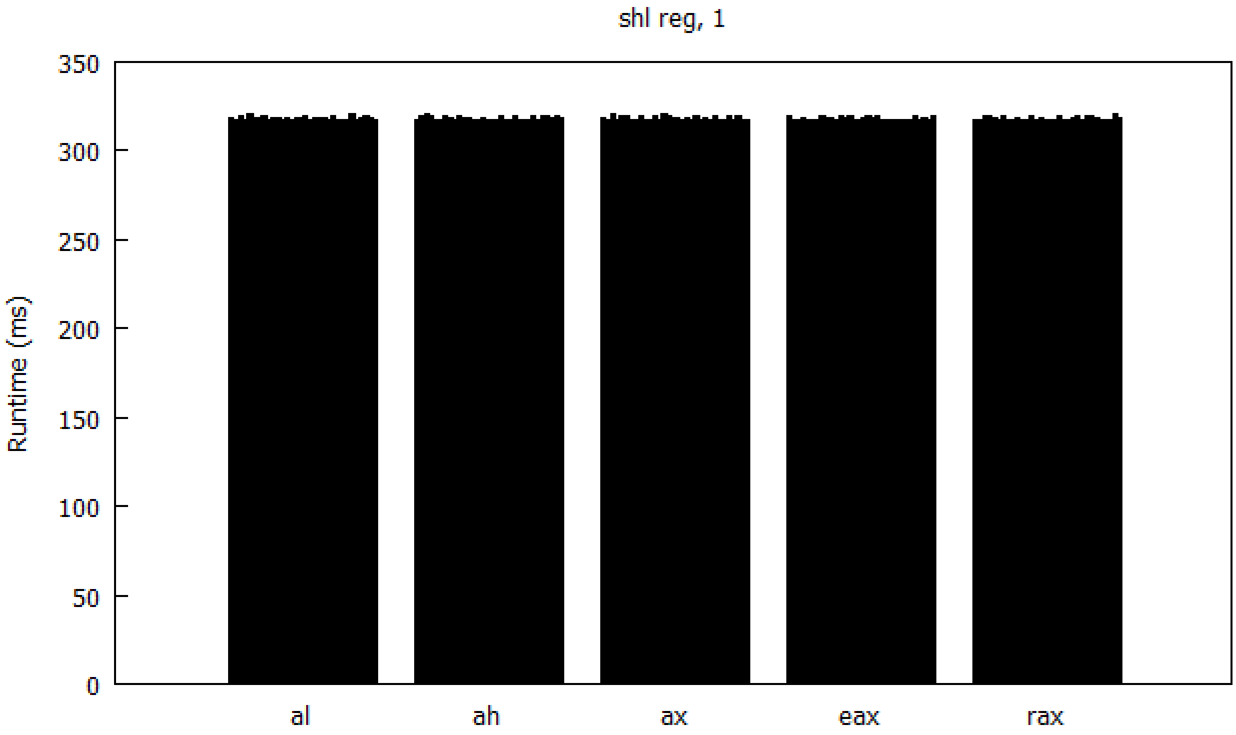

add reg, imm
Runtime (ms)
350
300
250
200
150
100
50
0
al
ah
ax
eax
rax

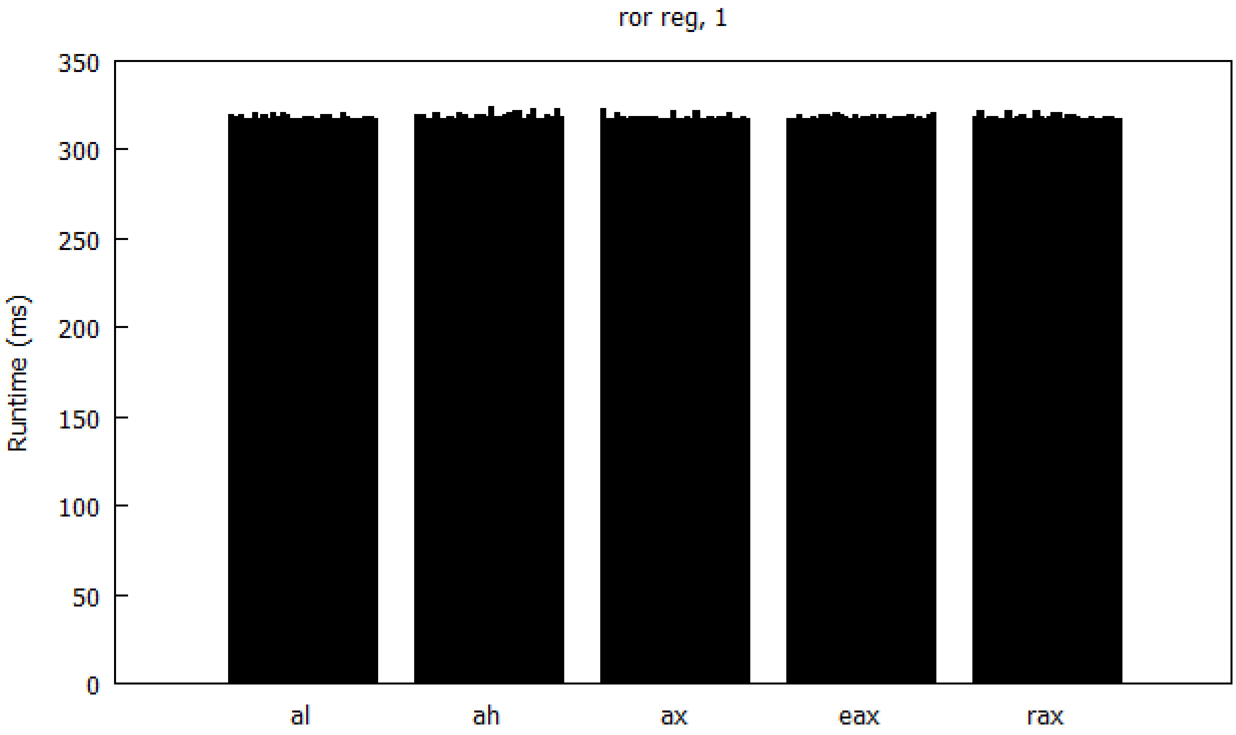

and reg, reg

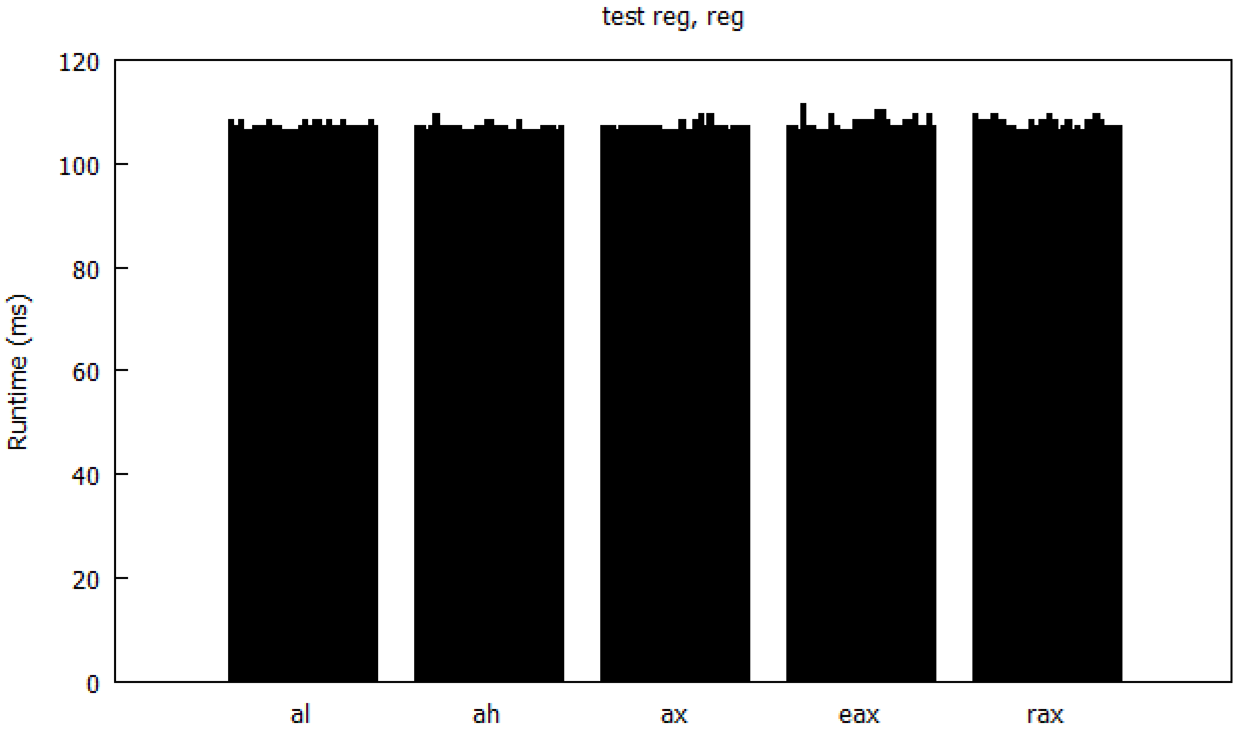

and reg, imm

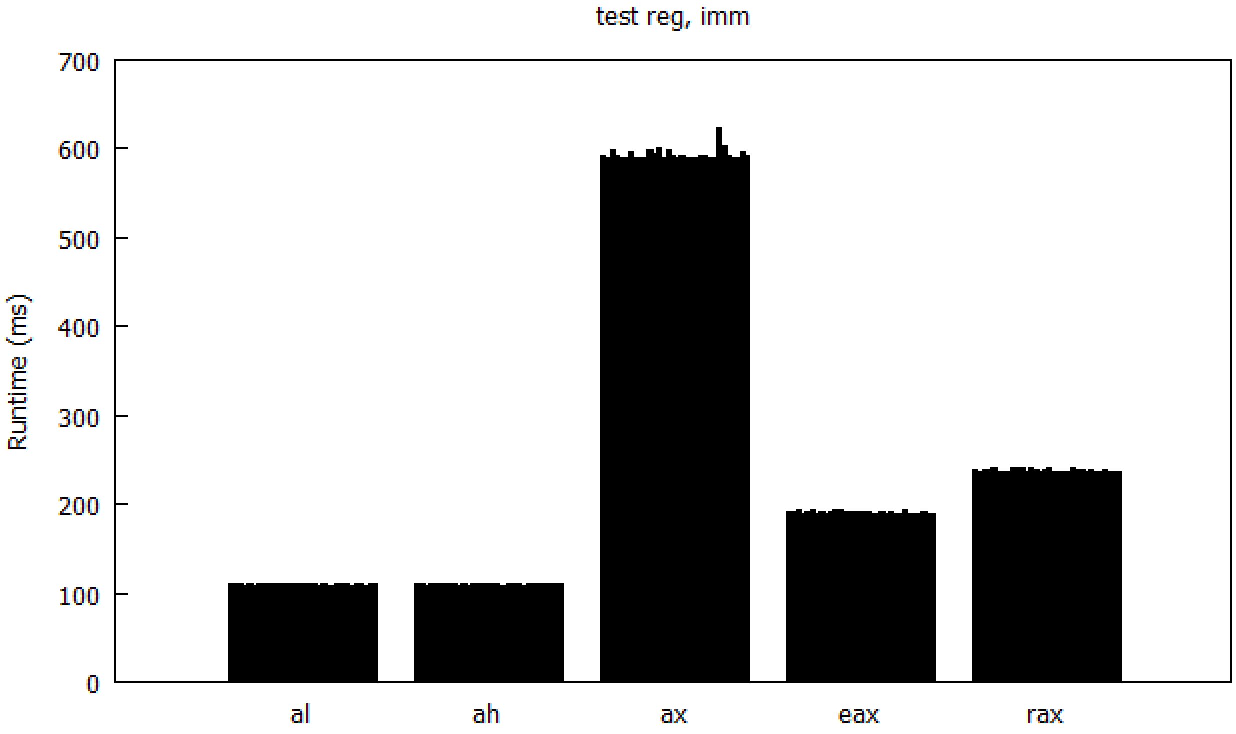

inc reg
Runtime (ms)
350
300
250
200
150
100
50
0
al
ah
ax
eax
rax

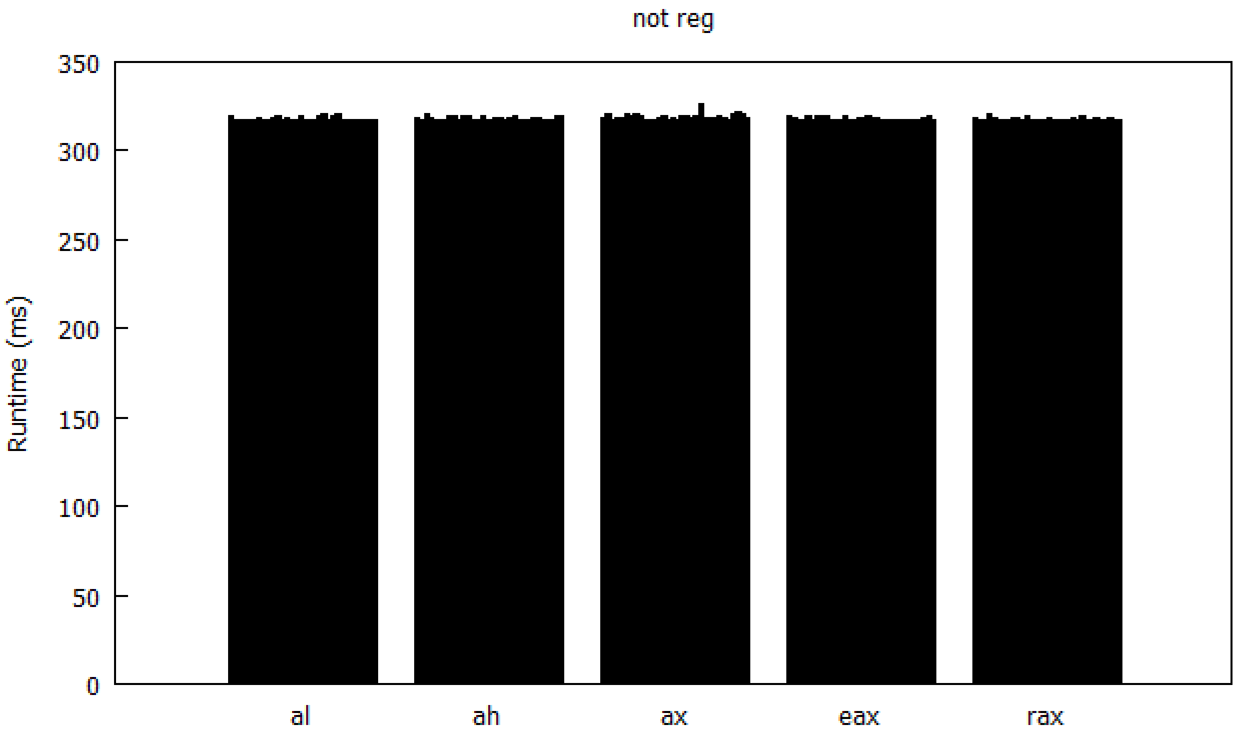

dec reg
Runtime (ms)
350
300
250
200
150
100
50
0
al
ah
ax
eax
rax

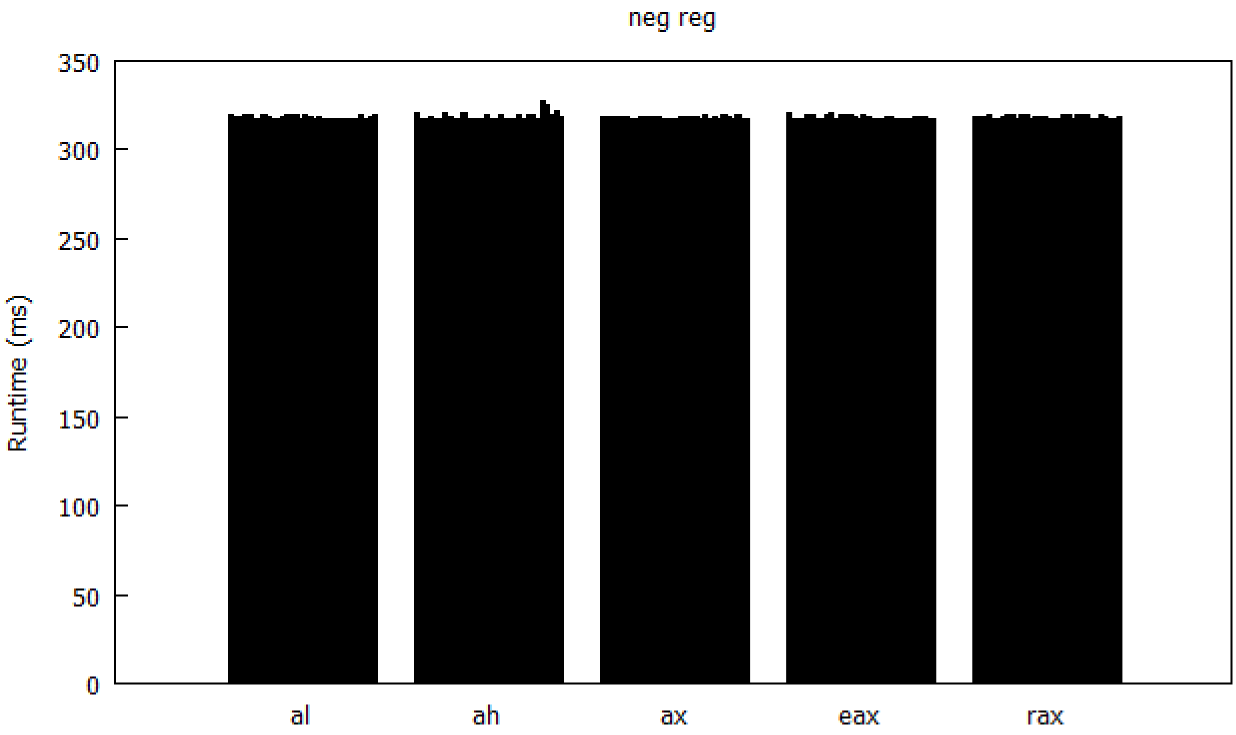

shl reg, 1
Runtime (ms)
0
50
100
150
200
250
300
350
al
ah
ax
eax
rax

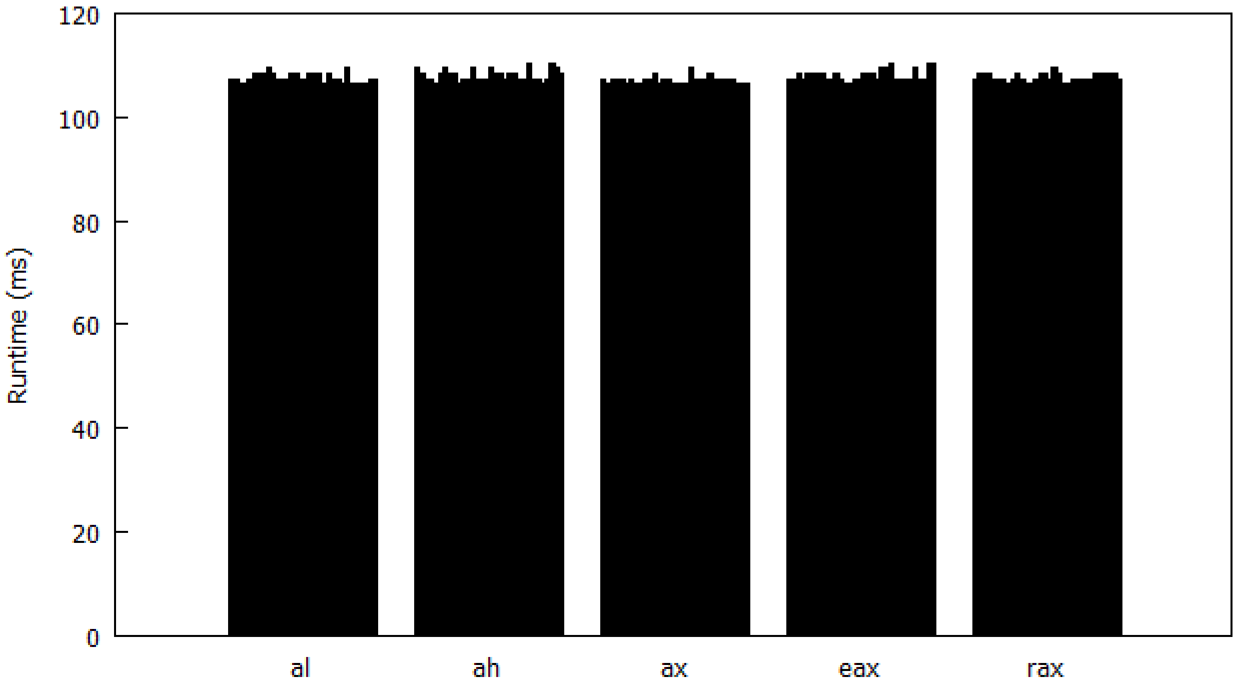

ror reg, 1
Runtime (ms)
0
50
100
150
200
250
300
350
al
ah
ax
eax
rax


test reg, reg
Runtime (ms)
0
20
40
60
80
100
120
al
ah
ax
eax
rax

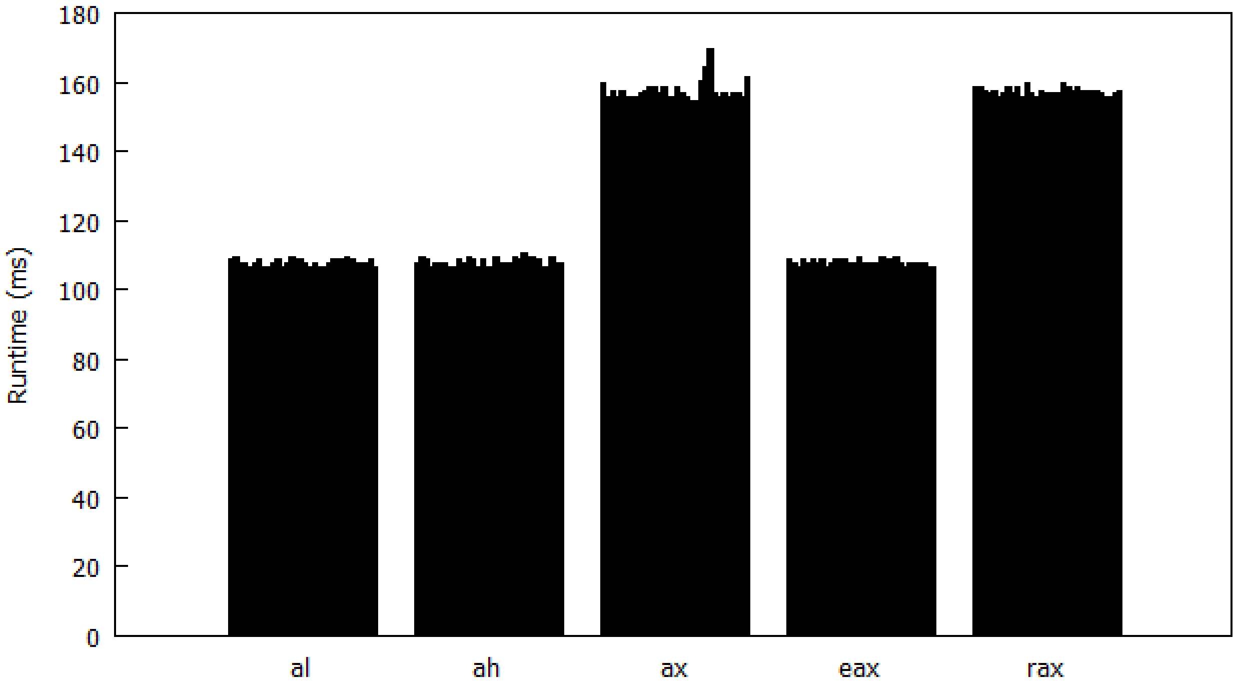

test reg, imm
Runtime (ms)
0
100
200
300
400
500
600
700
al
ah
ax
eax
rax


not reg
Runtime (ms)
350
300
250
200
150
100
50
0
al
ah
ax
eax
rax

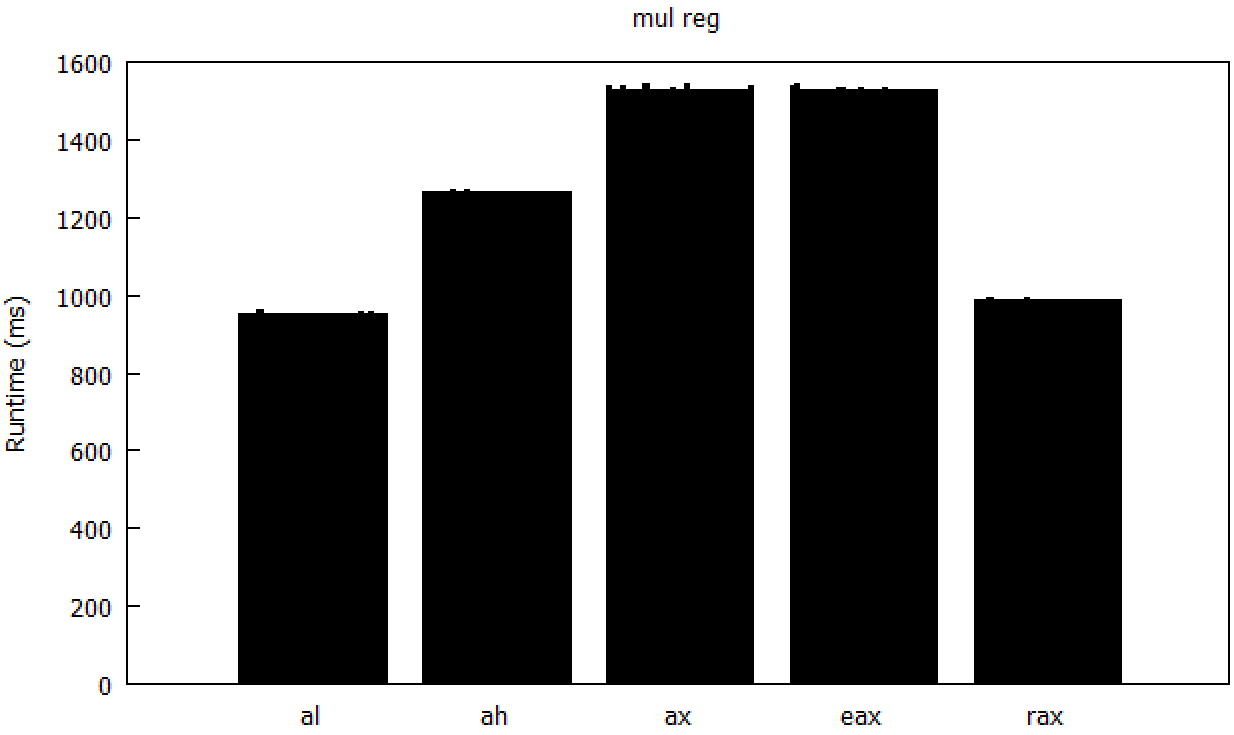

neg reg
Runtime (ms)
350
300
250
200
150
100
50
0
al
ah
ax
eax
rax



cmp reg, reg



cmp reg, imm



Conclusion

There was no affect in any register only operations, and only a few operations which bit-width did impact. These operations were *mov*, *test*, *cmp*, and *mul*. An interesting thing to note is that in the *mul* operation, whether you were using *al* or *ah* did actually seem to have an affect, however no other operations showed similarities to this. Another strange finding is that the *cmp* and *test* operations depended on the bit width, however *add* and *and* did not *(sub* produced the same results as *add*, and thus has been omitted). There must be a reason for this at hardware level with the temporary register.

All and all these findings were a bit less exciting than expected, however there were some very strange findings such as *mul* which made up for it.