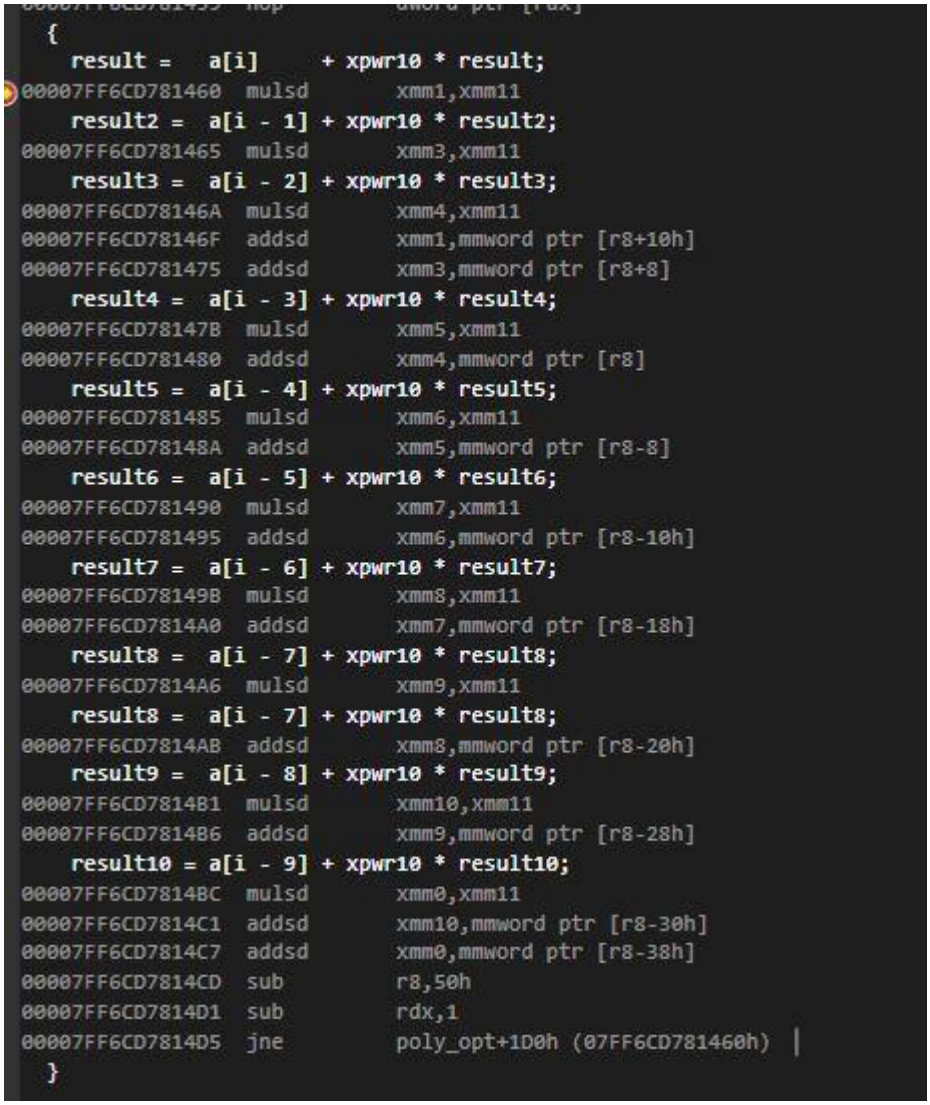
x64 vs x86 Perfomance

Firstly, here is the analysis of the x64 assembly code.

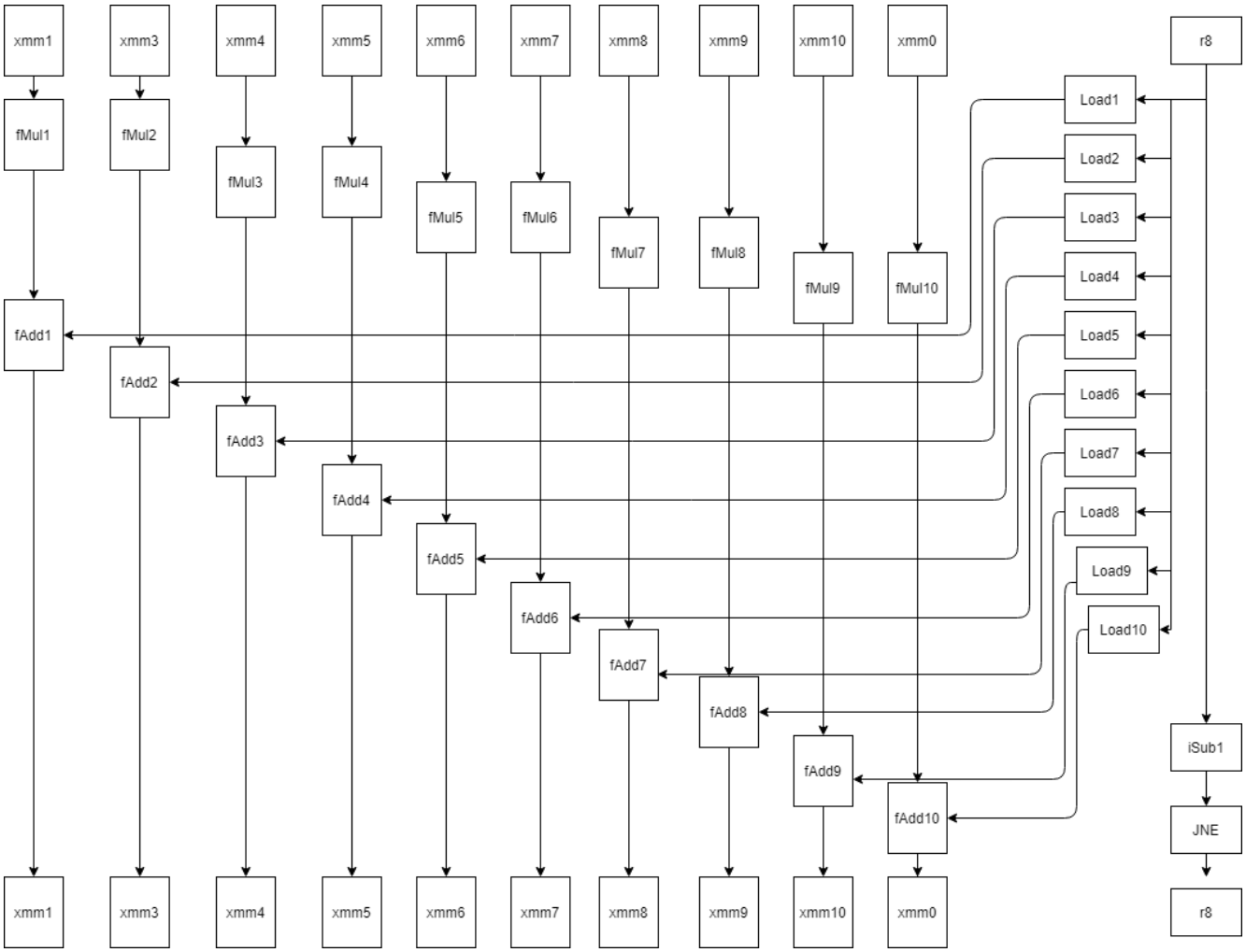


And here is the assembly code for the x86 code.

```
Viewing Options
  {
    result =   a[i]     + xpwr10 * result;
00B414C0  mulsd       xmm3,mmword ptr [esp+10h]
    result2 =  a[i - 1] + xpwr10 * result2;
00B414C6  mulsd       xmm7,mmword ptr [esp+10h]
    result3 =  a[i - 2] + xpwr10 * result3;
00B414CC  mulsd       xmm6,mmword ptr [esp+10h]
00B414D2  addsd       xmm3,mmword ptr [ebx+10h]
    result4 =  a[i - 3] + xpwr10 * result4;
00B414D7  mulsd       xmm5,mmword ptr [esp+10h]
    result5 =  a[i - 4] + xpwr10 * result5;
00B414DD  mulsd       xmm4,mmword ptr [esp+10h]
    result6 =  a[i - 5] + xpwr10 * result6;
00B414E3  mulsd       xmm1,mmword ptr [esp+10h]
    result7 =  a[i - 6] + xpwr10 * result7;
    result8 =  a[i - 7] + xpwr10 * result8;
    result9 =  a[i - 8] + xpwr10 * result9;
00B414E9  mulsd       xmm2,mmword ptr [esp+10h]
    result10 = a[i - 9] + xpwr10 * result10;
00B414EF  mulsd       xmm0,mmword ptr [esp+10h]
00B414F5  movsd       mmword ptr [esp+30h],xmm3
00B414FB  movsd       xmm3,mmword ptr [esp+60h]
00B41501  mulsd       xmm3,mmword ptr [esp+10h]
    result10 = a[i - 9] + xpwr10 * result10;
00B41507  addsd       xmm7,mmword ptr [ebx+8]
00B4150C  addsd       xmm6,mmword ptr [ebx]
00B41510  addsd       xmm5,mmword ptr [ebx-8]
00B41515  addsd       xmm3,mmword ptr [ebx-20h]
00B4151A  addsd       xmm4,mmword ptr [ebx-10h]
00B4151F  addsd       xmm1,mmword ptr [ebx-18h]
00B41524  addsd       xmm2,mmword ptr [ebx-30h]
00B41529  addsd       xmm0,mmword ptr [ebx-38h]
00B4152E  movsd       mmword ptr [esp+60h],xmm3
00B41534  movsd       xmm3,mmword ptr [esp+68h]
00B4153A  mulsd       xmm3,mmword ptr [esp+10h]
00B41540  addsd       xmm3,mmword ptr [ebx-28h]
00B41545  sub         ebx,50h
00B41548  movsd       mmword ptr [esp+68h],xmm3
00B4154E  movsd       xmm3,mmword ptr [esp+30h]
00B41554  sub         edx,1
00B41557  jne         poly_opt+1F0h (0B414C0h)
00B4155D  movsd       mmword ptr [esp+58h],xmm1
00B41563  movsd       xmm1,mmword ptr [esp+70h]
00B41569  movsd       mmword ptr [esp+50h],xmm4
00B4156F  movsd       xmm4,mmword ptr [esp+78h]
00B41575  movsd       mmword ptr [esp+48h],xmm5
00B4157B  movsd       xmm5,mmword ptr [esp+80h]
00B41584  movsd       mmword ptr [esp+40h],xmm6
00B4158A  movsd       xmm6,mmword ptr [esp+88h]
00B41593  movsd       mmword ptr [esp+38h],xmm7
00B41599  movsd       xmm7,mmword ptr [esp+90h]
  }
```

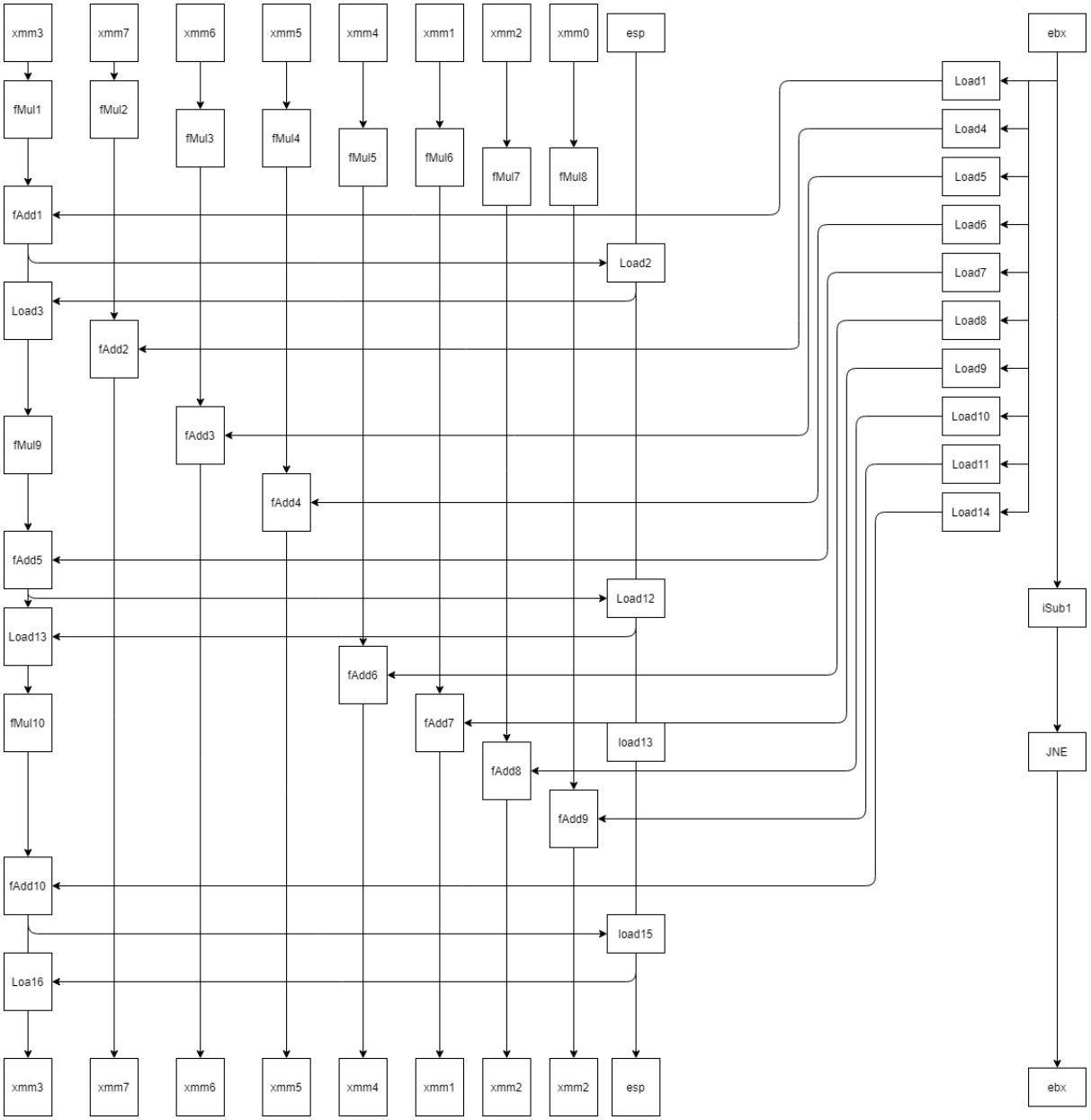
At first glance, you can tell there are already more instructions for the x86 assembly code.

Here is a flow chart to explain the x64 assembly code.



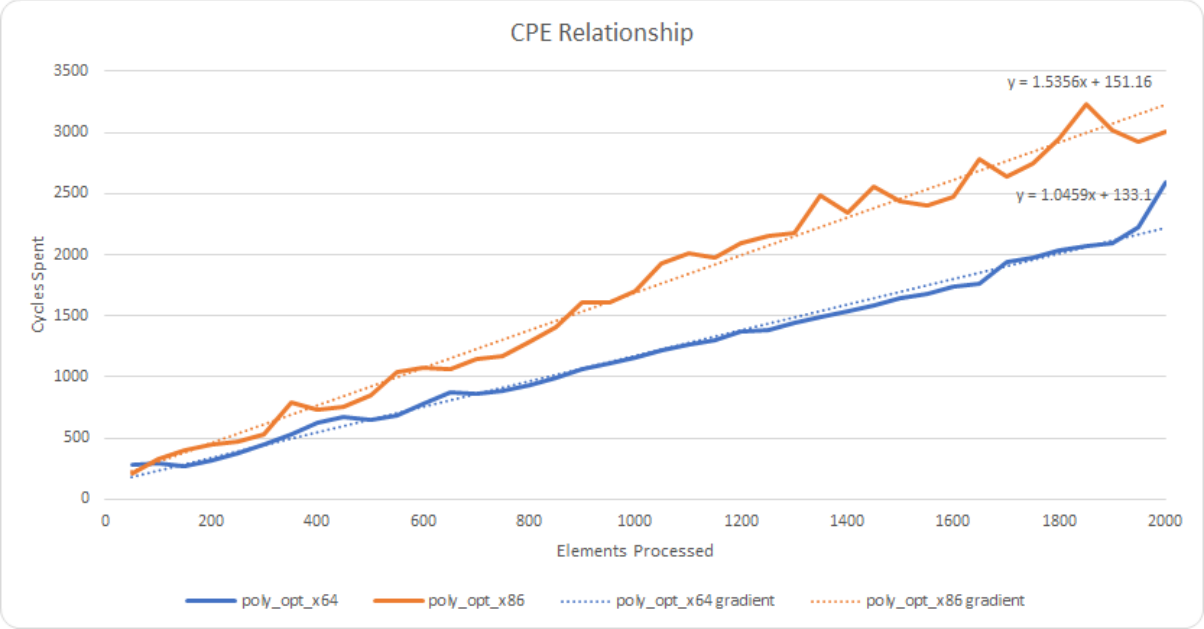
Ten xmm registers here are used as accumaltors. xmm2 is a local register not shown in the flow chart(because it's a local register) that actually stores the value for x. This allows multiplication of 2 registers to be carried out at the same time. Then the accumulators can do addition with the next elements after they get loaded.

Here is a flowchart of the x86 assembly code.



The flowchart is similar to the x64 assembly flowchart where it also multiplies with x and do addition with the next element. The biggest difference is that x86 does not have as many xmm registers. The consequence is that x is now stored in the stack, as well as 2 other accumulators. This results in the xmm3 register having to swap out the values for the 2 accumulators from the stack and finally restoring the original accumulator at the end for the next loop.

Lastly, here is a graph to show the difference between the cpe for x86 and x64.



As expected, x64 has a lower cpe as x86 requires one of the xmm registers to constantly swap in the 2 accumulators from the stack to perform the algorithm, which results in more cycles.