Assembly Optimization Tips

by

Mark Larson

The most important thing to remember is to TIME your code. Trying different tricks might or migh
up your code. So it is very important to time your code to see if you do get a speedup as you try e

Beginner

1. **Freeing up all 8 CPU registers for use in your code.**

```
    push ebx
    push esi
    push edi
    push ebp                    ; has to be done before changing ESP

    ;load up ESI and EDI and your other registers with the vales passed
    ;in on the stack. This has to be done before freeing up ESP.

    movd mm0,esp                ; no pushing/popping past this point.
    xor  ebx,ebx                ; So how do you save?  A variable.
    mov  esp,5

inner_loop:
    mov   [eax_save],eax        ; eax_save is a global variable. Can't be
    movzx eax,word ptr [fred_2] ; local because that requires EBP to point to it.
    add   ebx,eax
    mov   eax,[eax_save]        ; restore eax

    movd  esp,mm0               ; has to be done before you do the POPs

    pop   ebp
    pop   edi
    pop   esi
    pop   ebx
    ret
```

2. **Maximize register usage.**

   Most high level compilers generate a lot of memory accesses to variables. I usually get around that l
   keep most of them in registers. Having all 8 cpu registers free can really come in handy.

3. **Complex instructions**

   Avoid complex instructions (lods, stos, movs, cmps, scas, loop, xadd, enter, leave). Complex instru
   instructions that do multiple things. For instance stosb writes a byte to memory and also increments
   stopped making these fast with the original Pentium because they were trying to make it more RISC
   REP and the string instructions is still fast. That is the only exception to the case.

4. **Don't use INC/DEC on P4**

   On the P4 use ADD/SUB in place of INC/DEC. Generally it is faster. ADD/SUB runs in 0.5 cycles. INC
   1 cycle.

5. **Rotating**

   Avoid rotate by a register or rotate by an immediate value of anything but a 1.

6. **Eliminate unnecessary compare instructions**

   Eliminate unnecessary compare instructions by doing the appropriate conditional jump instruction ba
   flags that are already set from a previous arithmetic instruction.

```
        dec     ecx
        cmp     ecx,0
        jnz     loop_again

;gets changed to
        dec     ecx
        jnz     loop_again
```

7. **LEA is still really cool, except for on the P4 it tends to be slow.**

   You can perform multiple math operations all in one instruction and it does not affect the flags regis
   can put in in between one register being modified and a flags comparison jump on the next line.

```
top_of_loop:

    dec   eax
    lea   edx,[edx*4+3]     ; multiply by 4 and add 3. Does not affect flags
    jnz   top_of_loop       ; so the next instruction doesn't get hosed.
```

8. **ADC and SBB.**

   Most compilers don't really make good use of ADC and SBB. You can get good speeds ups with that.
   64-bit numbers together, or adding big numbers together. Keep in mind that on the P4 ADC and SB
   As a work around you can use "addq", and use MMX to do this. So the second optimization suggesti
   to use MMX to do the adding or subtracting. You just have to have a processor that supports MMX.

```
    add    eax,[fred]
    adc    edx,[fred+4]

    ; the above 2 statements are the same as the below 3 statements

    movd   mm0,[fred]       ; Get 32-bit value in MM0
    movd   mm1,[fred+4]     ; Get 32-bit value in MM1
    paddq  mm0,mm1          ; This is an unoptimized way to do it. You would
                            ; really pre-read MM0 and MM1 a loop in advance.
                            ; I did it this way for ease of understanding.
```

9. **ROL, ROR, RCL, and RCR and BSWAP.**

   It is a cool trick to switch from Big Endian to Little Endian using BSWAP. Also you can use it for tem
   storage of a 16-bit or 8-bit value in the upper half of the register. Likewise you can use ROL and RO
   8-bit and 16-bit values. It's a way to get more "registers". If all you are dealing with are 16-bit valu
   turn your 8 32-bit registers into 16 16-bit registers. Which gives you a lot more registers to use. RC
   can also easily be used for counting the number of bits that are set in a register. Keep in mind that
   RCL, RCR and BSWAP are all slow on the P4. The rotate instructions are about twice as fast as BSW
   have to use one or the other on the P4 use the rotate ones.

```
    xor   edx,edx           ; set both 16-bit registers to 0
    mov   dx,234            ; set the first 16-bit register to 234
    bswap edx               ; swap it so the second one is ready
    mov   dx,345            ; set the second 16-bit register to 345
    bswap edx               ; swap to the first one
    add   dx,5              ; add 5 to the first one
    bswap edx               ; swap to the second one
    add   dx,7              ; add 7 to it
```

10. **String instructions.**

    Most compilers don't make good use of the string instructions (scas, cmps, stos, movs, and lods). S to see if that is faster than some library routine can be a win. For instance I was really surprised wh at strlen() in VC++. In the radix40 code it ran in 416 cycles for a 100 byte string!!! I thought that w slow.

11. **Multiply to divide.**

    If you have a full 32-bit number and you need to divide, you can simply do a multiply and take the t half as the result. This is faster because multiplication is faster than division. (thanks to pdixon for t

12. **Dividing by a constant.**

    There's some nice information now how to divide by a constant in Agner Fog's pentopt.pdf documen program that you pass in the number you want to divide by, and it will print out the assembler code will dig it up later and post it. Here is a link to Agner's document. Agner's Pentopt PDF (http://www.agner.org/assem/)

13. **Unrolling.**

    This is a guideline. Unrolling falls in the General Optimiation category but I wanted to add a footnote set up my Unrolling with a macro that unrolls an EQUATE value amount. That way you can try differ and see which is best easily. You want the unrolling to fit in the L1 code cache (or trace cache). Usi equate makes it easy to try different unroll amounts to find the fastest one.

```
UNROLL_AMT       equ   16   ; # of times to unroll the loop
UNROLL_NUM_BYTES equ    4   ; # of bytes handled in 1 loop iteration

        mov     ecx,1024
looper:
offset2 = 0
REPEAT UNROLL_AMT
        add     eax,[edi+offset2]
offset2 = offset2 + UNROLL_NUM_BYTES
        add     edi,UNROLL_AMT * UNROLL_NUM_BYTES   ; we dealt with 16*4 bytes.
        sub     ecx,UNROLL_AMT  ; subtract from loop counter the # of loops we unrol
        jnz     looper
```

14. **MOVZX.**

    Use MOVZX to avoid partial register stalls. I use MOVZX a lot. A lot of people XOR the full 32-bit reg But MOVZX does the equivalent thing without having to have an extra XOR instruction. Also you had XOR enough in advance to give it time to complete. With MOVZX you don't have to worry about that

15. **Using MOVZX to avoid a SHIFT and AND instruction**

    I ran across this bit of C code I was trying to speed up using assembler. The_array is a dword array. trying to get a different byte from a dword in the array passed upon which pass this is over the data variable that goes from 0 to 3 for each byte in a particular dword.

```
        unsigned char c = ((the_array[i])>>(Pass<<3)) & 0xFF;

; I got rid of the "pass" variable by unrolling the loop 4 times.
; So I had 4 of these each one seperated by lots of C code.
        unsigned char c = (the_array[i])>>0) & 0xFF;
        unsigned char c = (the_array[i])>>8) & 0xFF;
        unsigned char c = (the_array[i])>>16) & 0xFF;
        unsigned char c = (the_array[i])>>24) & 0xFF;
```

What if I can get rid of the SHIFT and the AND using assembler? That would save me 2 instructions. mention the fact that the P4 is very slow when doing SHIFT instructions (4 cycles!!!). So try to avoi where possible. SO taking just the second to last line that shifts right 16 as our example

```
; esi points to the_array

        mov     eax,[esi]
        shr     eax,16
        and     eax,0FFh
```

; So how do we change that to get rid of the AND and SHR?
; We do a MOVZx with the 3rd byte in the dword.

```
      movzx   eax,byte ptr [esi+2]          ;unsigned char c = (the_array[i])>>16) & 0xFF;
```

16. **Align, align, align.**

    It is really important to align both your code and data to get a good speed up. I generally align code boundaries. For data I align 2 byte data on 2 byte boundaries, 4 byte data on 4 byte boundaries, 8 b 8 byte boundaries, 16 byte data on 16 byte boundaries. In general if you don't align your SSE or SS a 16-byte boundary you will get an exception. You can align your data in VC++ if you have the proc They added support for both static data and dynamic memory. For static data you use __declspec(al alignes on a 4 byte boundary.

17. **BSR for powers of 2.**

    You can use BSR to count the highest power of 2 that goes into a variable.

18. **XORing a register with itself to zero it.**

    This is an oldie, but I am including it anyway. It also has a side benefit of clearing dependencies on t That is why sometimes you will see people use XOR in that fashion, before doing a partial register ac prefer using MOVZX to doing it that way because it is trickier to do using a XOR (read my above cor about in #12 above talking about MOVZX) . On the P4 they also added support for PXOR to break de in that fashion. I think the P3 does the same thing.

19. **Use XOR and DIV.**

    If you know your data can be unsigned for a DIVISION, use XOR EDX, EDX, then DIV. It's faster tha IDIV.

20. **Try to avoid obvious dependencies.**

    If you modify a register and then compare it to some value on the very next line, instead try and pu other register modification in between. Dependencies are any time you modify a register and then re write it shortly afterwards.

```
   inc edi
   inc eax
   cmp eax,1    ; this line has a dependency with the previous line, so it will stal
   jz  fred

;shuffling the instructions around we can help break up dependencies.
   inc eax
   inc edi
   cmp eax,1
   jz  fred
```

21. **Instructions to avoid on P4.**

    On P4's try to avoid the following instructions, adc, sbb, rotate instructions, shift instructions, inc, de any instruction taking more than 4 uops. How do you know the processor running the code is a P4? (

22. **Using lookup tables.**

On the P4 sometimes you can get around the long latency instructions that I listed previously by doi tables. Thankfully on P4's they come with really fast memory. So having to do a lookup table doesn' performance as much if it isn't in the cache.

23. **Use pointers instead of calculating indexes.**

A lot of times in loops in C there will be multiplications by non-powers of 2 numbers. You can easily this by adding instead. Here is an example that uses a structure.

```
typedef struct fred
{
   int fred;
   char bif;
} freddy_type;

freddy_type charmin[80];
```

The size of freddy_type is 5 bytes. If you try and access them in a loop the compiler will generate cc multipling by 5 for each array access!!!! (Ewwwwwwwwwwwwww). So how do we do it properly?

```
for ( int t = 0; t < 80; t++)
{
   charmin[t].fred = rand(); // the compiler multiplies by 5 to get the offset, EWWW
   charmin[t].bif = (char)(rand() % 256);
}

; in assembler we start with an offset of 0, that points to the first data item.
; And then we add 5 to it each loop iteration to avoid the MUL.

   mov   esi,offset charmin
   mov   ecx,80
fred_loop:
   ;... perform operations on the FRED and BIF elements in freddy_type
   add   esi,5                    ;make it point to the next structure entry.
   dec   ecx
   jnz   fred_loop
```

The MUL removal applies to loops as well. I have seen people do multiplies in loops as part of increm variable or for terminating condition. So try doing addition instead.

24. **Conform to default branch predictions.**

Try to set up your code such that backward conditional jumps are usually taken, and forward conditi are almost never taken. That has to do with branch prediction. The static branch predictor uses that to guess if a conditional jump is taken or not. So have a loop that has a backwards conditional jump And then have special exit conditions from that same loop that executes a forward jump that only ex certain condition that doesn't often occur.

25. **Eliminate branches**

Eliminate branch where possible. This might seem obvious, but I have seen some people use too ma in their assembler code. Keep it simple. Use as few branches as possible.

26. **Using CMOVcc to remove branches**

I have yet to see the CMOVcc instructions actually be faster than a conditional jump. So I recommer conditional jumps over CMOVcc. It might be faster in the case where your jumps aren't easily guess branch prediction logic. So if that is the case with you, benchmark it and see.

27. **Local vs. Global variables**

Use local variables for a procedure over using a global variable. If you use local variables you'll get l
misses.

28. **Address Calculation**

Compute address calculations before you need them. Let's say you have to do some funky stuff to g
particular address. Such as multiplying by 20. You can pre-compute that before you get to the point
where you need it.

29. **Smaller registers**

Sometimes using smaller registers will give you a speed up. I did this on the radix40 code. If you ch
below code to use EDX it runs slightly slower.

```
        movzx          edx,byte ptr [esi]             ;get the data from the ascii
        test           dl,ILLEGAL                     ;bit 7 set?  if so do ILLEGAl
        jnz            skip_illegal
```

30. **Instruction Length**

Try and keep your instructions to 8 bytes or less.

31. **Use registers to pass parameters**

Try passing parameters in registers instead of on the stack where possible. If you have 3 variables t
have to push onto the stack as paramaters, that is at least 6 memory reads and 3 memory writes. Y
read each variable from memory into a CPU register and then push it on the stack. That is 3 memory
there. Then the 3 pushes onto the stack make 3 writes. Then why would you push parameters you'd
So figure at least 3 (maybe more) reads from the stack of the pushed data.

32. **Don't pass big data on the stack**

Don't pass 64-bit data or 128-bit data (or bigger) on the stack. Instead pass a pointer to the data.

Intermediate

1. **Adding to memory faster than adding memory to a register**

This has to do with the number of micro-ops the instruction takes. Give preference to doing an add w
memory over adding memory to a register.

```
        add     eax,[edi]       ;don't do this if possible
        add     [edi],eax       ;This is preferred
```

2. **Instruction selection**

Try and pick instructions with the fewest micro-ops and shortest latencies.

3. **Handling an unaligned byte data stream that needs to be dword aligned**

Parsing a byte array a dword at a time will get you performance hits due to the buffer not being alig
byte boundary. You can get around this by dealing with the first X bytes (0 to 3), until you come to
to a 4 byte boundary.

4. **Using CMOVcc to reset an infinite loop pointer**

If you are making multiple passes through an array, and want to reset it to the beginning when you
reached the end of the array, you can use CMOVcc.

```
        dec ecx             ; decrement index into array
        cmovz ecx,MAX_COUNT ; if we are at the beginning, then reset the index
```

```
                                ; to MAX_COUNT (the end).
```

5. **Multiplying by 0.5 by doing a subtraction**

   This probably won't work for everything, but multiplying by 0.5, or dividing by 2.0 in real4 (in floati
   you can just subtract 1 from the exponent. Won't work with 0.0. For real8, the subtract value is 001
   donkey posted this)

```
.data
        somefp real4    2.5
.code
        sub dword ptr [somefp],00800000h        ;divide real4 by 2.
```

6. **Self Modifying Code**

   The P4 optimization manual recommends avoiding self-modifying code. I have seen cases where it c
   faster. But as always you need to time it to verify in your case that it is faster.

7. **MMX, SSE, SSE2**

   Most compilers don't generate good code for MMX, SSE and SSE2. GCC and the Intel compiler have
   better at it. But hand tooled assembler is still a big win in this area.

8. **Using EMMS.**

   EMMS tends to be a really slow instruction on Intel processors. On AMD it is faster. Generally I don't
   per routine basis, because it is so slow. I very rarely use a lot of floating point in a program that I al
   a lot of MMX in (and vice versa). So I usually wait to do the EMMS before doing any floating point. If
   lot of floating point and very little MMX, then do the EMMS at the end of all the MMX routines you ca
   any). But adding it in every routine that does MMX just makes the code run slow.

9. **Converting to MMX,SSE, or SSE2**

   Can your code be convert to MMX, SSE, or SSE2? If so you can get a big speed up by doing stuff in

10. **Prefetching data.**

    This is underutilized a lot. If you are processing a huge array (256KB and up), using the "prefetch"
    on P3 and up processors can speed up your code anywhere from 10-30%. You can actually get a de
    performance if you don't use it right. Unrolling works well with this, because I unroll to the number c
    fetched with this instruction. On a P3 it is 32, but on a P4 it is 128. That means you can easily unroll
    to handle 128 bytes at a time on a P4 and get the benfit from unrolling and prefetching. It is not alw
    case that if you unroll it for 128 bytes that you will get the best speed up. So try different variations

```
UNROLL_NUM_BYTES equ    4                       ; # of bytes handled in one
                                                ; iteration of the loop.
UNROLL_AMT       equ    128/UNROLL_NUM_BYTES    ; We want to unroll the loop such
                                                ; that we handle 128 bytes per loop.

        mov     ecx,1024
looper:
offset2 = 0
REPEAT UNROLL_AMT
        prefetchnta [edi+offset2+128] ; prefetch 128 bytes into the L1 cache before w
        add     eax,[edi+offset2]
offset2 = offset2 + UNROLL_NUM_BYTES
        add     edi,UNROLL_AMT * UNROLL_NUM_BYTES ;we dealt with 16*4 bytes.
        sub     ecx,UNROLL_AMT        ; subtract from loop counter the # of loops we
        jnz     looper
```

11. **Cache Blocking**

    Let's say you have to call multiple procedures on this big array in memory. It is better to break it up
    that fit into the cache to reduce cache misses. For example if you were doing 3D code, the first proc

translate your coordinates, the second might scale and the third might rotate. So instead of going th
whole huge array in one fell swoop. You break off a "chunk" of the data that fits into the cache, and
3 procedures, and then go to the next chunk and repeat.

12. **TLB Priming**

    The TLB is the Translation Lookaside Buffer. The TLB is cache that is used to improve performance o
    translation of a virtual memory address to a physical memory address by providing fast access to pa
    entries. Having it not in the TLB cache forces a cache misse which slows the code down. The trick is
    a data byte from the next page before you have to read it. I will show an example later on in anothe
    tips.

13. **Intermix your code to break dependencies.**

    In C code the C compiler treats different chunks of code as seperate. To break dependencies, when y
    the assembler level you can intermix them

14. **Parallelization.**

    Most Compilers don't take advantage of the fact you have 2 pipelines for ALU stuff, which is the maj
    what people use. On the P4 you have it even sweeter. You can execute 4 ALU instructions in 1 cycle
    right. If you break things up into doing stuff in parallel it also helps break up dependencies. So you k
    with one stone. Assume this piece is in a loop.

```
looper:
        mov     eax,[esi]
        xor     eax,0E5h        ;dependency with the line above it.
        add     [edi],eax       ;dependency with the line above it.
        add     esi,4
        add     edi,4
        dec     ecx
        jnz     looper

;So how do we 'parallelize' it and reduce dependencies?
looper:
        mov     eax,[esi]
        mov     ebx,[esi+4]
        xor     eax,0E5
        xor     ebx,0E5
        add     [edi],eax
        add     [edi+4],ebx
        add     esi,8
        add     edi,8
        sub     ecx,2
        jnz     looper
```

15. **Avoiding memory accesses**

    Re-structing the code to avoid memory accesses (or other I/O). One method is to accumulate a valu
    register before writing it to memory. Here is an example of that below. In this example assuming we
    3 bytes from the source array to the destination array which is an array of dwords. The destination a
    zeroed.

```
        mov     ecx,AMT_TO_LOOP
looper:
        movzx byte ptr eax,[esi]
        add     [edi],eax
        movzx byte ptr eax,[esi+1]
        add     [edi],eax
        movzx byte ptr eax,[esi+3]
        add     [edi],eax
        add     edi,4
        add     esi,3
        dec     ecx
        jnz     looper
```

We can accumulate the result in a register, and then only do one write to memory.

```
        mov     ecx,AMT_TO_LOOP
looper:
        xor     edx,edx                 ;zero out register to accumulate the result i
        movzx byte ptr eax,[esi]
        add     edx,eax
        movzx byte ptr eax,[esi+1]
        add     edx,eax
        movzx byte ptr eax,[esi+3]
        add     edx,eax
        add     esi,3
        mov     [edi],edx
        add     edi,4
        dec     ecx
        jnz     looper
```

16. **When to convert a call to a jump**

    if the last statement in a routine is a call consider converting it to a jump to get rid of one call/ret.

17. **Using arrays for data structures**

    (This is non-assembler related, but it's a great one). You can use an array for data structures such a linked lists. By using an array the memory ends up being contiguous and you get a speed up due to misses.

Advanced

1. **Avoid prefixes**

   Try to avoid prefixes (prefixes get generated for a number of things including segment overrides, bı operand-size override, address-size override, LOCKs, and REPs). Prefixes make your instructions lon

2. **Grouping reads / writes in code**

   If there is a bunch of alternating between read and write transactions on the bus, look at grouping a more reads at a time and more writes at a time. Here is what we are trying to avoid:

```
        mov eax,[esi]
        mov [edi],eax
        mov eax,[esi+4]
        mov [edi+4],eax
        mov eax,[esi+8]
        mov [edi+8],eax
        mov eax,[esi+12]
        mov [edi+12],eax

;Grouping the reads and writes together this gets converted to
        mov eax,[esi]
        mov ebx,[esi+4]
        mov ecx,[esi+8]
        mov edx,[esi+12]
        mov [edi],eax
        mov [edi+4],ebx
        mov [edi+8],ecx
        mov [edi+12],edx
```

3. **Making use of execution units to make your code run faster**

Choose instructions that execute on different execution units. If you do this properly the time to exe
code will be the throughput time and not the latency time. For most instructions the throughput time

4. **Interleaving 2 loops out of sync**

   You can unroll a loop twice, and instead of running each instruction after each other, you can run th
   sync. Why is this useful? 2 reasons. First, sometimes you have instructions that have to use a certai
   and have a long latency such as MUL or DIV. That creates a dependency on EDX:EAX for the two MU
   instructions in a row. Second, sometimes some instructions just have really long latencies. So you w
   and place a number of instructions after it from the other loop to help delay until it returns the resul
   MMX, SSE, and SSE2 instructions on the P4 fall into that category. Here is an example loop: A1 ; ins
   loop 1 D2 ; instruction 4 loop 2 B1 ; instruction 2 loop 1 A2 ; instruction 1 loop 2 C1 ; instruction 3 l
   instruction 2 loop 2 D1 ; instruction 4 loop 1 C2 ; instruction 3 loop 2

5. **Different tricks you can do with masks created by comparison instructions using MMX/SS**

   With MMX and SSE and SSE2 you can generate masks when doing comparisons. This can be helpful
   cases when looking for a pattern in a file, such as a line feed. So you can use it to search for pattern
   math operations.

   You can use MMX, SSE, and SSE2 masks that get generated by a compare instruction to control doi
   only part of a MMX or SSE register. The following piece of code only adds a 9 to the dword parts of a
   register if it has a 5 in it.

```
; if (fredvariable == 5)
;       fredvariable += 9;
    movq     mm5,[two_fives]        ;mm5 has 2 DWORD 5's in it.
    movq     mm6,[two_nines]        ;mm6 has 2 DWORD 9's in it.
    movq     mm0,[array_in_memory]  ;get value
    movq     mm1,mm0                ;get backup copy
    pcmpeqd  mm1,mm5                ;mm1 now has an FFFFFFFF in each dword location
                                    ; in MM1 with a 5 all other locations have 0.
    pand     mm1,mm6                ;zero out the locations in MM6 that don't
                                    ; have 5's in them for MM0.
    paddd    mm0,mm1                ;add 9 ONLY to the locations in MM0 that
                                    ; have a 5 in them.
```

6. **PSHUFD and PSHUFW.**

   On the P4 MMX, SSE and SSE2 move instructionns are slow. You can get around this by doind "pshu
   SSE and SSE2 and "pshufw" for MMX. It is 2 cycles faster. There is one caveat. It has to do with wha
   the opcode goes down. So without getting too technical, sometimes it is faster to use the slower "MC
   than to replace it with a "PSHUFD". So time your code.

```
        pshufd  xmm0,[edi],0E4h     ;copy 16 bytes at location EDI into XMM0.
                                    ; The 0E4h makes it a straight copy.
        pshufw  mm0,[edi],0E4h      ;copy 8 bytes at location EDI into MM0.
                                    ; The 0E4h makes it a straight copy.
```

7. **Write directly to memory - bypass the cache.**

   Another optimization dealing with memory. If you have to write to a lot of memory (256KB and up)
   to write directly to memory bypassing the cache. If you have a P3 you can use "movntq" or "movntp
   does an 8 byte write, the second a 16-byte. The 16-byte write needs to be 16-byte aligned. On the P
   get "movntdq", which does 16-bytes also, but needs to be 16-byte aligned. This trick applies to both
   fills and memory copies. Both do a write operation. Here is some sample code. I personally would do
   registers in parallel to help break up some of the latencies for the P4 MOVDQA instruction. However
   understanding, I did not do that.

```
        mov     ecx,16384           ;write 16384 16-byte values, 16384*16 = 256KB.
                                    ; So we are copying a 256KB array
        mov     esi,offset src_arr  ;pointer to the source array which has to be
                                    ; 16-byte aligned or you will get an exception.
        mov     edi,offset dst_arr  ;pointer to the destination array which has to be
                                    ; 16-byte aligned or you will get an exception.
looper:
        movdqa  xmm0,[esi]          ;works on P3 and up
        movntps [edi],xmm0          ;Works on P3 and up
        add     esi,16
        add     edi,16
        dec     ecx
        jnz     looper
```

8. **Handle 2 cases per loop for MMX / SSE / SSE2.**

On the P4 the latencies are usually so long on the MMX, SSE and SSE2 instructions that I always ha per loop, or read a loop in advance. Or more than 2 cases if I have enough registers. All the various including MOVD) instructions on P4 are slow. So adding 2 32-bit arrays of numbers together is going slower on the P4 than the P3. A faster way would be to do two per loop, where you pre-read the loo values MM0 and MM1 before the FRED label. You just have to have special handling if you have an o of array elements. Just check that at the end, and if so add code for that one extra dword. Here is th does not read a value in advance. I think converting this to read a value in advance is easy. So that not posting both. This is how you would avoid ADC on a P4, which is a slow instruction, to add two a together.

```
    pxor    mm7,mm7     ; the previous loops carry stays in here.
fred:
    movd    mm0,[esi]   ; esi points to src1
    movd    mm1,[edi]   ; edi points to src2, also to be used as the destination.
    paddq   mm0,mm1     ; add both values together
    paddq   mm0,mm7     ; add in remainder from last add.
    movd    [edi],mm0   ; save value to memory
    movq    mm7,mm0
    psrlq   mm7,32      ; shift the carry over to bit 0.
    add     esi,8
    add     edi,8
    sub     ecx,1
    jnz     fred
    movd    [edi],mm7   ; save carry
```

9. **Pre-reading MMX or XMM register to get around long latency**

Pre-reading an SSE2 register before you need it will give a speed up. That is because MOVDQA take a P4. That is really slow. So because it has such a long latency, I want to read it in advance of wher make sure it doens't stall. Here is an example. movdqa xmm1,[edi+16] ;read in XMM1 before we us cycles on P4, not including the time to get it from cache. por xmm5,xmm0 ;do an OR with XMM0 wh previously read. Takes 2 cycles on the P4. pand xmm6,xmm0 ;do an AND with XMM0 which was pre read. Takes 2 cycles on the P4. movdqa xmm2,[edi+32] ;read in XMM2 before we use it, takes 6 cy not including the time to get it from cache. por xmm5,xmm1 ;do an OR with XMM1 which was previo Takes 2 cycles on the P4. pand xmm6,xmm1 ;do an AND with XMM1 which was previously read. Tak on the P4.

10. **Accumulating a result in a register or registers to avoid doing a slow instruction**

Accumulating a result in a register or registers to avoid doing a slow instruction. I did this to speed u compare/read loop written in SSE2. The slow instruction was PMOVMSKB. So instead of executing it I accumulated a result in a register. And then every 4KB of memory read, I would do a PMOVMSKB. good speed up. The example below will also demonstrate using PREFETCH and TLB Priming. Their ar the below code. The inner loop is unrolled to 128 bytes(the number of bytes fetched by PREFETCH c The outer loop is unrolled to 4KB, so that it can do TLB Priming. If you are using a system that does 4KB page size, you'd have to do modify the code appropriately. On the system I tested this on (a D with 6.4 GB/s of max memory bandwidth, I was able to do a read and a compare at 5.55 GB/s (in a non-Windows environment. Under windows it will run slower). I left out the code that at label "comp for 2 reasons. 1) The cut/paste of code is already big. 2) It doesn't demosntrate any techniques I wa The code at "compare_failed" simply does a REP SCASD to find the failing address after PCMPEQD fi nearest 4KB block. This one has a HUGE code example, so I put it last in case you fall asleep reading

```
read_compare_pattern_sse2 proc near

                mov         edi,[start_addr]        ;Starting Address
                mov         ecx,[stop_addr]         ;Last addr to NOT test.
                mov         ebx,0FFFFFFFFh          ;AND mask
                movd        xmm6,ebx                ;AND mask
                pshufd      xmm6,xmm6,00000000b     ;AND mask
                movdqa      xmm0,[edi]              ;Get first 16 bytes
                mov         eax,[pattern]           ;EAX holds pattern
                pxor        xmm5,xmm5               ;OR mask
                movd        xmm7,eax                ;Copy EAX to XMM7
                pshufd      xmm7,xmm7,00000000b     ;Blast to all DWORDS
outer_loop:
                mov         ebx,32                  ;128 32 byte blocks
                mov         esi,edi                 ;save start of block

if DO_TLB_PRIMING
                mov         eax,[edi+4096]          ;TLB priming
endif ; if DO_TLB_PRIMING

fred_loop:
                movdqa      xmm1,[edi+16]           ;read 16 bytes
                por         xmm5,xmm0               ;OR into mask
                pand        xmm6,xmm0               ;AND into mask

                movdqa      xmm2,[edi+32]           ;read 16 bytes
                por         xmm5,xmm1               ;OR into mask
                pand        xmm6,xmm1               ;AND into mask

                movdqa      xmm3,[edi+48]           ;read 16 bytes
                por         xmm5,xmm2               ;OR into mask
                pand        xmm6,xmm2               ;AND into mask

                movdqa      xmm0,[edi+64]           ;read 16 bytes
                por         xmm5,xmm3               ;OR into mask
                pand        xmm6,xmm3               ;AND into mask

                movdqa      xmm1,[edi+80]           ;read 16 bytes
                por         xmm5,xmm0               ;OR into mask
                pand        xmm6,xmm0               ;AND into mask

                movdqa      xmm2,[edi+96]           ;read 16 bytes
                por         xmm5,xmm1               ;OR into mask
                pand        xmm6,xmm1               ;AND into mask

                movdqa      xmm3,[edi+112]          ;read 16 bytes
                por         xmm5,xmm2               ;OR into mask
                pand        xmm6,xmm2               ;AND into mask

                por         xmm5,xmm3               ;OR into mask
                prefetchnta [edi+928]               ;Prefetch 928 ahead
                pand        xmm6,xmm3               ;AND into mask

                add         edi,128                 ;Go next 128byteblock
                cmp         edi,ecx                 ;At end?
                jae         do_compare              ;No, jump

                movdqa      xmm0,[edi]              ;read 16 bytes

                sub         ebx,1                   ;Incr for inner loop
                jnz         fred_loop

do_compare:
                pcmpeqd     xmm5,xmm7               ;Equal?
```

```
pmovmskb    eax,xmm5                ;Grab high bits in EAX
cmp         eax,0FFFFh              ;all set?
jne         compare_failed          ;No, exit failure

mov         edx,0FFFFFFFFh          ;AND mask
pxor        xmm5,xmm5
pcmpeqd     xmm6,xmm7               ;Equal?

pmovmskb    eax,xmm6                ;Grab high bits in EAX
cmp         eax,0FFFFh              ;All Set?
jne         compare_failed          ;No, exit failure

movd        xmm6,edx                ;AND mask
pshufd      xmm6,xmm6,00000000b     ;AND mask

cmp         edi,ecx                 ;We at end of range
jb          outer_loop              ;No, loop back up

jmp         compare_passed          ;Done!!! Success!!!
```

11. **Prefetch distance and location within the loop**

    You will notice in the previous example that I prefetch 928 bytes ahead instead of 128, when 128 is
    of bytes fetched on a P4. Why is that? Well Intel recmomends prefetching 128 bytes (2 cache lines)
    the start of your loop. I found both statements (doing at the beginning of the loop and prefetching 1
    ahead) to be wrong. I don't prefetch at the beginning of the loop nor do I prefetch 128 bytes ahead.
    when I was playing with the code I found that I could make it run faster by moving the PREFETCH in
    around in the loop and changing the offest for how far ahead it prefetches. So being the geek I am I
    to try all combinations of the location of the prefetch instruction in the loop and offsets to begin pref
    code takes an assembler file and moves the "prefetch" instruction around inside a loop, and also mo
    offset to begin prefetching. Then a batch file compiles the just modified code, and runs a benchmark
    benchmark over several hours to try the different combinations (I started at a prefetch distance of 3
    went up to 1024, in increments of 32). On the system I wrote the code on 928 ahead instead of 128
    fastest. And prefetching almost at the end of the loop was fastest (the prefetchnta instruction is abo
    above the do_compare: label)

# # #

visitors since 09 jul 2004:
020242 020240 000852