© The CodeBreakers-Journal, Vol. 1, No. 1 (2004)
http://www.CodeBreakers-Journal.com

64-Bit - Programming and Assembly Issues

Thorsten Schneider
University of Bielefeld, Germany

Abstract

The 64-Bit technology introduces several new and and complex tasks for software-developers. Even that the hardware developing industry claims that future software development should take care about the new introduced compiler systems, it is necessary to have a deep inside view on how the new underlaying 64-Bit assembly language works. This paper describes what 64-Bit means for future software devlopments, how 64-Bit influences assembly programming and how to port applications programmed under 32-Bit to 64-Bit. It is heavy based on the IA architecture and the Windows® operationsystem.

Keywords: *Assembly; 64-Bit; win64; Itanium; Athlon64; Porting win32 to win64; Windows®; Linux; EPIC*

I. Introduction

The introduction of the 64-Bit technology leads into several discussions if this change is really necessary [21]. Some arguments are to speed up computation time or to make calculations more precise. These affords lead into hard problems which have to be solved by upcoming compilers or operationsystems. Especially the necessary changes of the operationsystem infrastructure causes heavy problem porting older 32-Bit applications to 64-Bit compatibility. Examples for enhanced operatingsystem detection have been described by Kruse [15]. Since assembly programming is a comfortable task under 32-Bit systems (e.g. MASM, TASM, NASM, FASM and others), the question stays on how to port software and which changes of the architecture affects the current programming techniques.

II. The 64-Bit Architecture

The new 64-Bit Architecture introduces many innovations. Some of them are described by Jarp [14]:

- Rich Instruction Set
- Bundled Execution
- Predicated Instructions
- Large Register Files
- Register Stack
- Rotating Registers
- Modulo Scheduled Loops
- Control/Data Speculation
- Cache Control Instructions
- High-precision Floating-Point

Compared to the IA-32 architecture the new IA-64 architecture allows several advantages for the programmer. One of them is the clear and explicit programming technique which results in EPIC (Explicit Parallel Instruction Computing). EPIC is a Intel® IA-64 technique. Additional the programmer can code now more register-based, which means that everything is kept in registers as long as possible. The resulting assembly is code has a clear structure (example by Intel [9]):

```
.align 32
.section .text

.proc demo
demo:: ld4              r32 = [r32]
       mov              r55 = r24
       add              r44 = 32, r10
       mov              r17 = 3
       ;;
j_loop:
       cmp.le           p15,p0 = r32,r0
       add              r32 = -1,r32
       ;;
(p15) br.cond.spnt.few j_loop_end
       ;;
       add              r55 = r44, r43
       fadd             f6 = f4, f6
       nop.m            0
(p17) ld8              r40 = [r50]
       ;;
j_loop_end:
(p16) add              r50 = r40,r0
       nop.f            0
       br.ctop.sptk.few j_loop
       ;;
_done:
       br.ret.sptk.few  b0
       ;;
.endp
```

It is noticeable that Intel® offers SSE3 with the introduction of the Prescott processor. These instructions are included therefore in the ”real” 64-Bit processors of Intel® too (this information has been given by Kruse [15]. Kruse got this information from IDF 2004, San Francisco [8]). From this integration it is clear, that Prescott is nothing else than a 64-Bit processor which includes all remaining support but has not unlocked this support.

The AMD x86-64 differes from the IA-64 structure in important details (See figure 1). The developer has...

> *...access to eight additional GPRs, for a total of 16 GPRs. Furthermore, there are eight new SIMD registers, added for use in SSE/SSE2 code. So the number of GPRs and SIMD registers available to x86-64 programmers will go from eight each to sixteen each [21].*

III. The IA-64 Registers

64-Bit simply designate the number of bits that each of the processor’s general-purpose registers (GPRs) can hold [21]. This affects the number of instructions on a 64-Bit processor too.

In general the registers of a 64-Bit CPU are twice as wide as those in the 32-Bit CPU. As difference the instruction register (IR) size has the same size for 32-Bit and for 64-Bit processors

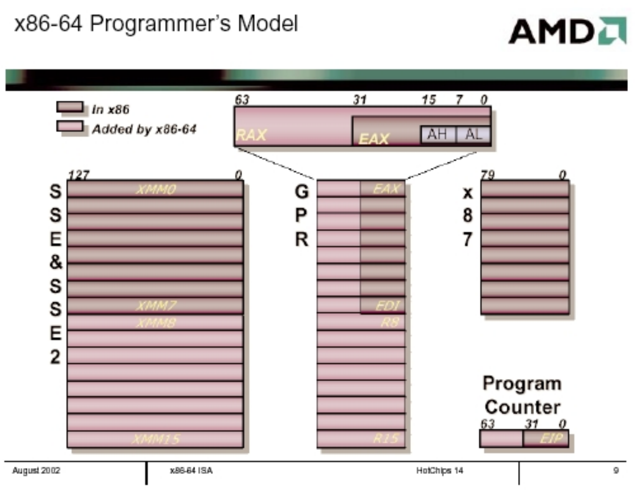

Fig. 1. AMD new programming model. Source: [21].

[21] [23] [22] [24]. So using of 64-Bit integer for example gives additional place to store data. A 64-bit architecture can (theoretically) address up to 18 million terabytes. Unfortunatly this can lead into wasting memory. Such problems have to be optimized in the future and are a problem of future compiler construction.

As one resulting example Win64 Pointers are eight bytes long, and not 4 as defined by Win32 systems. Another example is *that the size of LRESULT, WPARAM and LPARAM will all expand to 64 bits, so message handlers will have to be checked for inappopriate casts* [17]. Under the Intel® IA-64 structure the number of available registers have exploded. The assembly programmer should ignore most of them and can focus on a few sets of them. As Pietrek describes [19]:

> *To begin with, the IA-64 has 128 general-purpose registers, each 64 bits wide. These registers are conceptually similar to the general-purpose registers such as EAX on the x86. The IA-64 general-purpose registers are named with an r, followed by the register number. Thus, r0 is the first general-purpose register, and r127 is the last general-purpose register. The first 32 general-purpose registers (r0-r31) are static. That is, any code that refers to one of these registers will always be referring to the exact same register in silicon. All of the x86 registers act this way, and thus can be considered static. Some of the static registers have predefined meanings, and are usually referred to some other way than their r name. The global pointer and the stack pointer are two of the most important registers that fit this category. The r12 register is used as the stack pointer and is thus called the sp register. The r1 register is the global pointer, and you’ll see it referred to as*

> *the "gp" register.*

and

> *In addition to the 32 static general-purpose registers, the IA-64 also has 96 dynamic general-purpose registers [...]. Dynamic means that a given register name doesn't always refer to the exact same physical register on the CPU. That is, a register such as r34 in one function is likely to be assigned to a completely different physical register than r34 in another function.*

For the dynamic registers he states:

> *What's the point of dynamic registers? Everything about the IA-64 is focused on speed. Keeping values in registers and out of memory is one way to keep things running quickly. If a parameter is passed on the stack, and if that stack location isn't in the memory cache, the CPU might waste dozens of clock cycles just to read the parameter from the slow main memory. In contrast, registers can always be accessed in a single clock cycle. The dynamic registers exist so that each function can have its own set of up to 96 registers to work with. Inside a function, registers r32 through r127 are all essentially reserved for just that function. Hopefully, all of the function's local variables and parameters can be stored in these 96 registers. Of course, a function may not need all 96 registers, but they are available nonetheless. [...] In addition to the 128 general-purpose registers, the IA-64 also has 128 floating-point registers. They're named f0 through f127 (somebody probably gets paid a lot of money to come up with these names). The floating-point registers are 82 bits in length, allowing them to hold up to a C++ long double. As with the integer registers, certain floating-point registers have predefined meanings. For instance, the f0 register is always set to 0.0, while the f1 register always holds the value 1.0. [...] The last set of registers you need to know here are the branch registers. The IA-64 defines eight branch registers, named b0 through b7. These are 64-bit registers that contain the address of a code location that the CPU can transfer control to. On the IA-64, all control transfers take the form of a branch. The br.call instruction is equivalent to the x86 CALL; the br.ret instruction is like the x86 RET; and a simple br instruction is like an x86 JMP.*

An example on how to do Parameter Passing on the IA-64 has been descibed by Pietrek in another article [20]. An inside view on the Intel® 64-Bit architecture has been described in detail by Jarp [14].

```
~ registers                  assembly indirect
                             mnemonic access
----------------------------------------------
application                  ar       n
branch                       b        n
control                      cr       n
CPU identification           cpuid    y
data breakpoint              dbr      y
instriction breakpoint       ibr      y
data TLB translation         dtr      y
floating point               f        n
general                      r        n
instruction TLB translation  itr      y
protection key               pkr      y
performance monitor config   pmc      y
performance monitor data     pmd      y
predicate                    p        n
region                       rr       y
```

Fig. 2. Example for understanding the new registers.

```
AR0-AR7     equ AR.KR0-AR.KR7   kernel registers
AR8-AR15                        RESERVED
AR16        equ AR.RSC          register stack
                                configuration
AR17        equ AR.BSP          backing store pointer
                                (read only)
AR18        equ AR.BSPRESTORE   backing store pointer
                                mem stores
AR19        equ AR.RNAT         RSE NaT  collection
AR20                            RESERVED
AR21        equ AR.FCR          IA-32 floating-point
                                control
AR22,AR23                       RESERVED
AR24        equ AR.EFLAG        IA-32 EFLAG
AR25        equ AR.CSD          IA-32 code segment
                                descriptor ||
                                compare and store data
AR26        equ AR.SSD          IA-32 stack segment
                                descriptor
AR27        equ AR.CFLG         IA-32 combined CR0 and CR4
AR28        equ AR.FSR          IA-32 floating point status
AR29        equ AR.FIR          IA-32 floating point instr.
AR30        equ AR.FDR          IA-32 floating point data
AR31                            RESERVED
AR32        equ AR.CCV          compare and exchange
                                compare value
AR33-AR35                       RESERVED
AR36        equ AR.UNAT         user NaT collection
AR37-AR39                       RESERVED
AR40        equ AR.FPSR         floatint point status
AR41-AR43                       RESERVED
AR44        equ AR.ITC          interval time counter
AR45-AR63                       RESERVED / AR48- IGNORED
AR64        equ AR.PFS          previous function state
AR65        equ AR.LC           loop count
AR66        equ AR.EC           epilog count
AR67-AR127                      RESERVED / AR112- IGNORED
```

Fig. 3. Application register example: thinking with names.

IV. Porting Applications for Win64

Since there are hard changes between the architectures of 32-Bit and 64-Bit there evolve several problems for software developers in porting their software. We put our focus here on the new Win64 operationsystem. The Microsoft platform SDK ships with a pre-beta release of an IA-64 compiler, which allows it to test compile code for Win64.

Therefore Win64 offers 4 different porting options for existing applications:

1) 64-bit Full Port
2) Small Address Space with 64-bit Pointers
3) Small Address Space with 32-bit Pointers
4) Win32 Application

A detailed description of all 4 options including all necessary steps for porting applications from Win32 to Win64 are described by Intel® Corporation [13]:

> *Win64 uses the LLP64 (or P64) uniform data model in which pointers and long long are 64 bits, but int and long are 32 bits. To support this data model and to provide compatibility with the existing Win32 code, new data types have been defined in Win64. Some of them are fixed precision data types, for instance INT32 or INT64. Others change size depending on the architecture, for instance INT_PTR.*

In general simply compiling a Win32 application as Win64 application can cause several compiler errors. To prevent such a behaviour it is necessary to change for example the Win32 API Calls from:

```
LONG iVal = GetWindowLong(hWnd, GWL_HINSTANCE);
```

to

```
LONG_PTR iVal = GetWindowLongPtr(hWnd, GWLP_HINSTANCE);
```

The problem is here that four of the existing Win32 APIs that are used to set or get polymorphic window class data items have been changed [13]:

- Get/SetClassLong changed to Get/SetClassLongPtr
- Get/SetWindowLong changed to Get/SetWindowLongPtr

Concerning all those problems it is necessary to adapt the source code in a massive way:

```
#if defined(_WIN32)
 // stuff related to Win32
#if !defined(_WIN64)
 // Win32 without Win64 (regular Win32)
#else /* is _WIN64 also */
 // Win64 variant of Win32
#endif /* _WIN64 ? */
#elif defined(__unix) ||
 // various UNIXes
#else /* some other OS */
#error Unhandled OS;
#endif
```

Problems and solutions for adapting source code have been given by Chen [2].

V. Programming and Assembly under 64-Bit

One important change for the 64-Bit programmer is the *Global Pointer* and it's support code which has no equivalent on 32-Bit systems [18]. The Global Pointer is a preassigned value which makes it possible to access data within a load module. On the IA64, each instruction is 41 bits in length. The Global Pointer concept has been described by Pietrek [18].

VI. Assembly Optimization

Optimizing assembly code for 64-Bit has been described by Fletcher [7] using the example of the bubble sort algorithm:

```
void bubblesort(int count, int array[]) {
   for(int pass = 1; pass < count; pass++) {
      for(int i = 0; i < count-1; i++) {
         if(array[i] > array[i+1]) {
            int hold = array[i];
            array[i] = array[i+1];
            array[i+1] = hold;
         }
      }
   }
}
```

After cleaning up some name mangling the Intel® C++ Compiler produces the following assembly code [7]:

```
.proc bubblesort
bubblesort:
{ .mii
cmp4.ge.unc p9,p0=1,r32 //0: 20 7
mov r20=ar.lc //0: 19 2
add r19=1,r0 //0: 20 6
}
{ .mmb
add r16=4,r33 //0: 21 31
add r18=-1,r32 //0: 21 32
(p9) br.cond.dpnt .b1_1 ;; //0: 20 8
// Block 2: lentry Pred: 0 3 Succ: 3 7
// Freq 5.0e+001, Prob 0.50
}
.b1_2:
{ .mii
add r19=1,r19 //0: 20 15
cmp4.ge.unc p8,p0=r0,r18 //0: 21 13
sxt4 r17=r18 //0: 21 27
}
{ .mmb
mov r9=r16 //0: 20 12
mov r3=r33 //0: 20 11
(p8) br.cond.dpnt .b1_3 ;; //0: 21 14
// Block 7: prolog Pred: 2
// Succ: 17 Freq 5.0e+002, Prob 1.00
}
{ .mmi
add r11=-1,r17 ;; //0: 0 28
nop.m 0
sxt4 r10=r11 ;; //1: 0 29
}
{ .mii
nop.m 0
mov ar.lc=r10 //2: 0 30
nop.i 0 ;;
// Block 17: lentry lexit ltail collapsed Pred: 7 17
// Succ: 17 3 Freq 2.5e+003, Prob 0.80
}
.b1_17:
{ .mmi
ld4 r8=[r9] //0: 22 46
ld4 r2=[r3] //0: 22 45
nop.i 0 ;;
}
{ .mii
cmp4.le.unc p0,p6=r2,r8 //1: 22 54
nop.i 0
nop.i 0 ;;
}
{ .mmi
(p6) st4 [r3]=r8 //2: 24 52
(p6) st4 [r9]=r2 //2: 25 53
add r9=4,r9 //2: 21 50
}
{ .mib
nop.m 0
add r3=4,r3 //2: 21 49
br.cloop.sptk .b1_17 ;; //2: 21 51
// Block 3: lexit epilog ltail Pred: 2 17 Succ: 2 1
// Freq 5.0e+001, Prob 0.80
}
.b1_3:
{ .mib
cmp4.lt.unc p7,p0=r19,r32 //0: 20 16
nop.i 0
(p7) br.cond.dptk .b1_2 ;; //0: 20 17
// Block 1: exit epilog Pred: 0 3 Succ:
// Freq 1.0e+000, Prob 1.00
}
.b1_1:
{ .mib
nop.m 0
mov ar.lc=r20 //0: 29 9
br.ret.sptk.many b0 ;; //0: 29 10
}
.endp
```

Optimizing the code above needs several steps. One of such steps is eliminating instructions where three instructions can be eliminated when 2 assumptions can be made. For example

```
cmp4.ge.unc p9,p0=1,r32 //0: 20 7
(p9) br.cond.dpnt .b1_1 ;; //0: 20 8
mov r20=ar.lc //0: 19 2
```

can be optimized to

```
add r19=1,r0 //0: 20 6
add r16=4,r33 //0: 21 31
add r18=-1,r32 //0: 21 32
```

Using a *bundle* the above code results in the following template. It is important here that m and i is used for the *bundle template*.

```
{ .mmi
add r19=1,r0 //0: 20 6
add r16=4,r33 //0: 21 31
add r18=-1,r32 ;; //0: 21 32
}
```

One of the next steps should be the finding of outer loops which is described by Fletcher [7] in detail. A explicit detailed description of the assembly language for the Itanium architecture is provided by Intel® Corporation [10] [12] [11].

VII. Problems of IA-64

Since the IA-64 is not using an absolute addressing mode the global variables are accessed through the $r1$ register which synonym is the Global Pointer (See section (V)). The `addl` instruction provided by the IA-64 architecture allows only adding constants up to 22 bit. This results in a maximum size of 4 MB of global variables [3]. For solving this problem the Intel® C++ Compiler optimzes the code (See section (VI)). The IA-64 compiler solves this problem by splitting global variables into two categories, ”small” and ”large” where the difference between small and large can be set via compiler settings. This results into a difference how assembly code looks like. The small variable would look like [3]:

```
addl    r30 = -205584, gp;;
        // r30 -> global variable
ld4     r30 = [r30]
        // load a DWORD from the global variable
```

whereas the large variant result into the following larger code snippet where *the variable itself is allocated in a separate section of the file and a pointer to the variable is placed into the ”small” globals variables section of the module* [3]:

```
addl    r30 = -205584, gp;;
        // r30 -> global variable forwarder
ld8     r30 = [r30];;
        // r30 -> global variable
ld4     r30 = [r30]
        // load a DWORD from the global variable
```

Such difference between small and large global variables needs a explicit definition of variables even in C++ code. Using `extern BYTE b[];` will results in that the compiler uses the large global variables concept to be on the save side. This results in larger compiled code and possible inefficiency of the code. Additionaly the programmer should take care if he defines a short global variable when it’s is large. The programmer has to take care about this difference when using same definitions within different headers. Using `extern BYTE b;` in one file and `extern BYTE b[256];` in another one will result for sure in some confusion [3].

Another problem has been described by Chen [4]. The IA-64 introduces *another possible bad consequence of a mismatched function signature*.

One example of such a bogous code is [5] [4] [25] [6]:

```
void MyCritSectProc(LPVOID /*nParameter*/)
{ ... }

hMyThread = CreateThread(NULL, 0,
              (LPTHREAD_START_ROUTINE) MyCritSectProc,
              NULL, 0, &MyThreadID);
```

Another example by Microsoft® MSDN misdeclares both: the return value and the input parameter. The result is a crash on Win64 [1]. More examples are described by Chen [4].

VIII. Conclusions and Future Work

64-Bit is a necessary evolution in processor architecture. Nevertheless the problems of porting software from 32-Bit to 64-Bit will cause problems under Win64. As well the differences between the Intel® and the AMD® Architectures seem to result in software adaption problems which have been described before in the mid eighties of the last century. It seems to me more important than before to stay in touch with assembly code since the different approaches of compilers and vendors produce not always good and reliable code. Future work should focus on detailed compiler analysis to pinpoint the absolute differences in resulting assembly code and in evaluation of the performance of the different compiler dependant assembly codes which has been done by Loughran before for Win32 environments [16].

References

[1] Asche, Ruediger R.: *Putting DLDETECT to Work*. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndllpro/html/msdn_dldwork.asp, 2004.

[2] Chen, Bill: *Intel Itanium Intel Itanium Porting Methodologies*. http://mail.gnu.org/archive/html/tsp-devel/2003-05/pdf00000.pdf, 2004.

[3] Chen, Raymond: *ia64 - misdeclaring near and far data*. http://blogs.msdn.com/oldnewthing/archive/2004/01/20/60603.aspx, 2004.

[4] Chen, Raymond: *Uninitialized garbage on ia64 can be deadly*. http://blogs.msdn.com/oldnewthing/archive/2004/01/19/60162.aspx, 2004.

[5] Clipcode: *Clipcode Snippets for the Clipcode Systems SDK - Synchronisation*. http://www.clipcode.net/content/win32_4.htm, 2004.

[6] Devi, M.: *Mutex*. http://phoenix.liunet.edu/m̃devi/win32/Mutex.htm, 2004.

[7] Fletcher, Jason A.: *Optimizing Itanium Processor Family Assembly Code*. http://www.intel.com/cd/ids/developer/asmo-na/eng/53764.htm, 2004.

[8] Intel: *IDF 2004, San Francisco*. http://www.intel.com/idf/, 2004.

[9] Intel: *Intel Corporation Website*. http://www.intel.com, 2004.

[10] Intel: *Intel® Itanium® Architecture Software Developer's Manual - Volume 1: Application Architecture, Revision 2.1*. http://www.intel.com/design/itanium/manuals/iiasdmanual.htm, 2004.

[11] Intel: *Intel® Itanium® Architecture Software Developer's Manual - Volume 1: Instruction Set Reference, Revision 2.1*. http://www.intel.com/design/itanium/manuals/iiasdmanual.htm, 2004.

[12] Intel: *Intel® Itanium® Architecture Software Developer's Manual - Volume 1: System Architecture, Revision 2.1*. http://www.intel.com/design/itanium/manuals/iiasdmanual.htm, 2004.

[13] Intel: *Quick Start Guide for Porting Applications for Win64*. http://h21007.www2.hp.com/dspp/files/unprotected/Itanium/Windows/Win64_Rev4.pdf, 2004.

[14] Jarp, Sverre: *A Detailed Tutorial of the IA-64 architecture*. http://sverre.home.cern.ch/sverre/IA64_1.pdf, 1999.

[15] Kruse, T: *Processless Applications - Remotethreads on Microsoft Windows® 2000, XP and 2003*. CodeBreakers-Journal, 1(1), 2004.

[16] Loughran, Steve: *Code for Speed - Writing High Performance Win32 Code*. http://www.iseran.com/Win32/CodeForSpeed/, 2004.

[17] Loughran, Steve: *Frequently Asked Questions about Windows Programming*. http://www.iseran.com/Win32/FAQ, 2004.

[18] Pietrek, Matt: *Programming for 64-bit Windows*. MSDN Magazine, November 2000, 2000. http://msdn.microsoft.com/msdnmag/issues/1100/hood/.

[19] Pietrek, Matt: *IA-64 Registers*. MSDN Magazine, June 2001, 2001. http://msdn.microsoft.com/msdnmag/issues/01/06/hood/.

[20] Pietrek, Matt: *IA-64 Registers, Part 2*. MSDN Magazine, July 2001, 2001. http://msdn.microsoft.com/msdnmag/issues/01/07/hood/.

[21] Stokes, Jon: *An Introduction to 64-bit Computing and x86-64*. http://www.arstechnica.com/cpu/03q1/x86-64/x86-64-1.html, 2004.

[22] Stokes, Jon: *The Pentium 4 and the G4e: an Architectural Comparison*. http://arstechnica.com/cpu/01q2/p4andg4e/p4andg4e-1.html, 2004.

[23] Stokes, Jon: *Understanding the Microprocessor - Part I: Basic Computing Concepts*. http://arstechnica.com/paedia/c/cpu/part-1/cpu1-1.html, 2004.

[24] Stokes, Jon: *Understanding the Microprocessor - Part II: Pipelining and Superscalar Execution*. http://www.arstechnica.com/paedia/c/cpu/part-2/cpu2-1.html, 2004.

[25] Wild, Chris: *Single Thread/Process Global Variable*. http://www.cs.odu.edu/w̃ild/windowsNT/Fall97/version3.htm#SLIDE6, 2004.