The Alpha AXP, part 1: Initial plunge

devblogs.microsoft.com/oldnewthing/20170807-00

August 7, 2017

Raymond Chen

Since the Itanium series was such a smash hit (two whole people read it!), here's another series for a now-defunct processor architecture which Windows once supported. The next who-knows-how-many days will be devoted to an introduction to the Alpha AXP processor, as employed by Win32.

The Alpha AXP follows in the traditional RISC philosophy of having a relatively small and uniform instruction set. The first Alpha AXP chip was dual-issue, and it eventually reached quad-issue. (There was an eight-issue processor under development when the Alpha AXP project was cancelled.) This series will focus on the original Alpha AXP architecture because that's what Windows NT for Alpha AXP ran on, and it will largely ignore features added later.

The Alpha AXP is a 64-bit processor. It does not have "32-bit mode"; the processor is always running in 64-bit mode. If the destination of a 32-bit operation is a register, the answer is always sign-extended to a 64-bit value. (This is known as the "canonical form" for a 32-bit value in a 64-bit register.) This one weird trick lets you close one eye and sort of pretend that it's a 32-bit processor. An Alpha AXP program running on 32-bit Windows NT still has full access to the 64-bit registers and can use them to perform 64-bit computations. It could even use the full 64-bit address space, if you were willing to jump through some hoops.

Each instruction is a 32-bit word, aligned on a 4-byte boundary. Unlike other RISC processors of its era, the Alpha AXP does not have branch delay slots. If you don't know what branch delay slots are, then consider yourself lucky.

Memory size terms in the Alpha AXP instruction set are *byte*, *word* (two bytes), *longword* (four bytes), and *quadword* (eight bytes).[1] In casual conversation, *longword* and *quadword* are usually shortened *long* and *quad*.

The Alpha AXP defines certain groups of instructions which are optional, such as floating point. If you perform an instruction which is not implemented by the processor, the instruction will trap into the kernel, and the kernel is expected to emulate the missing instruction, and then resume execution.

Registers

There are 32 integer registers, all 64 bits wide. Formally, they are known by the names *r0* through *r31*, but Win32 assigns them the following mnemonics which correspond to their use in the Win32 calling convention.

Register	Mnemonic	Meaning	Preserved?	Notes
r0	*v0*	value	No	On function exit, contains the return value.
r1…r8	*t0…t7*	temporary	No	
r9…r14	*s0…s5*	saved	Yes	
r15	*fp*	frame pointer	Yes	For functions with variable-sized stacks.
r16…r21	*a0…a5*	argument	No	On function entry, contains function parameters.
r22…r25	*t8…t11*	temporary	No	
r26	*ra*	return address	Not normally	
r27	*t12*	temporary	No	
r28	*at*	assembler temporary	Volatile	Long jump assist.
r29	*gp*	global pointer	Special	Not used by 32-bit code.
r30	*sp*	stack pointer	Yes	
r31	*zero*	reads as zero	N/A	Writes are ignored.

The *zero* register reads as zero, and writes to it are ignored. But it goes further than that: If you specify *zero* as the destination register for an instruction, the entire instruction may be optimized out by the processor! This means that any side effects *may or may not occur*. There are a few exceptions to this rule:

- Branch instructions are never optimized out. If a branch instructions specifies *zero* as the register to receive the return address, the branch is still taken, but the return address is thrown away.
- Load instructions are always optimized out. If a load instruction specifies *zero* as the destination register, the processor will never raise an exception. Instead, these "phantom loads" are used as prefetch hints to the processor.

Whereas the behavior of the *zero* register is architectural, the behavior of the other registers are established by convention.

Win32 requires that the *gp*, *sp*, and *fp* registers be used for their stated purpose throughout the entire function. (If a function does not have a variable-sized stack frame, then it can use *fp* for any purpose.) Some registers have stated purposes only at entry to a function or exit from a function. When not at the function boundary, those registers may be used for any purpose.

Register marked with "Yes" in the "Preserved" column must be preserved across the call; those marked "No" do not.

The *ra* register is marked "Not normally" because you don't normally need to preserve it. However, if you are a leaf function that uses no stack space and modifies no preserved registers, then you can skip the generation of unwind codes for the leaf function, but you must keep the return address in *ra* for the duration of your function so that the operating system can unwind out of the function should an exception occur. (Special rules for lightweight leaf functions also exist for Itanium and x64.)

What does it mean when I say that the *at* register is volatile?

Direct branch instructions can reach destinations up to 4MB from the current instruction. When the compiler generates a *bsr* instruction (branch to subroutine), it typically doesn't know how far away the destination is. The compiler just generates a *bsr* instruction with a fixup and hopes for the best. It is the linker who knows how far away the destination actually is, and if it turns out the destination is too far away, the linker changes

```
        ....
        BSR     toofaraway
        ....
```

to

```
        ....
        BSR     trampoline
        ....

trampoline:
        ... set the "at" register equal to the
        ... address of "toofaraway."
        JMP     (at)            ; register indirect jump
```

The linker inserts the generated trampoline code between functions, which also has as a consequence that a single function cannot be larger than 8MB.

Anyway, this secret rewriting means that any branch instruction can potentially modify the *at* register. In between branches, you can use *at*, but you cannot rely on its value remaining the same once a branch is taken. In practice, the compiler just avoids using the *at* register altogether.

The *gp* register is not used by 32-bit code. I don't know for sure, but I'm guessing that in 64-bit code, it serves the same purpose as the Itanium *gp* register.

Note that some register names, like *a0* look like hex digits. The Windows debugger resolves them in favor of hex values, so if you do `? a0` thinking that you're getting the value of the *a0* register, you're going to be disappointed. To force a symbol to be interpreted as a register name, put an at-sign in front: `? @a0`.

Even more confusing is that the Windows debugger's disassembler does not put the `0x` prefix in front of numbers, so when you see an `a0`, you have to use the context to determine whether it is a number or a register. For example,

```
LDA     a0, a0(a0)
        ^^  ^^ ^^
  register  |  register
          number
```

The first parameter to `LDA` and the parameter inside the parentheses must be a register, so the outer `a0`'s refer to the register. The thing just outside the parentheses must be a constant, so the middle `a0` is the number 160. Yes, it's confusing at first, but the uniform instruction set means that these rules are quickly learned, and you don't really notice it once you get used to it.

Another point of confusion is that the conventional placeholder names for registers in instructions are `Ra`, `Rb` and `Rc`. This should not be confused with the *ra* register.

There are thirty-two floating point registers. Formally, they are known as *f0* through *f31*, but Win32 assigns the following mnemonics:

Register	Mnemonic	Preserved?	Meaning
f0		No	Return value
f1		No	Second return value (for complex numbers)
f2…f9		Yes	
f10…f15		No	
f16…f21		No	First six parameters

f22…f30		No	
f31	*fzero*	N/A	Reads as zero. Writes are ignored.

There are four floating point formats supported. Two are the usual IEEE single and double precision formats. Two are special formats for backward compatibility with the DEC VAX. That's about all I'm going to say about floating point.

Finally, there are some special registers.

Register	Mnemonic	Meaning
pc	*fir*	program counter
lock_flag		For interlocked memory access
phys_locked		For interlocked memory access
fpcr		Floating point control register

Why is the program counter called *fir*? Because that stands for "faulting instruction register".

Clearly named by somebody wearing kernel-colored glasses.

These special registers are not directly accessible. To retrieve the program counter, you can to issue a branch instruction and save the "return address" into the desired destination register. We'll learn more about the *lock_flag* and *phys_locked* when we study interlocked memory access.

Note that there is no flags register.

I repeat: There is no flags register.

Here's what a register dump looks like in the Windows debugger:

```
  v0=00000000 00000016   t0=00000000 00000000   t1=00000000 00000000
  t2=00000000 00000000   t3=00000000 00000009   t4=00000000 00000001
  t5=00000000 0006f9d0   t6=00000000 00000008   t7=00000000 00000000
  s0=00000000 00000001   s1=00000000 00000000   s2=00000000 00081eb0
  s3=00000000 77fc0000   s4=00000000 00081dec   s5=00000000 77fc0000
  fp=00000000 7ffde000   a0=00000000 750900c8   a1=00000000 00000001
  a2=00000000 00000009   a3=00000000 0006f9d0   a4=00000000 00000001
  a5=00000000 00000001   t8=00000000 0000004c   t9=00000000 00000001
 t10=00000000 0000004c  t11=ffffffff c00ea124   ra=00000000 77f4df08
 t12=00000000 00000001   at=00000000 77f548f0   gp=00000000 00000000
  sp=00000000 0006f9e0 zero=00000000 00000000 fpcr=08000000 00000000
softfpcr=00000000 00000000  fir=77f63bf4
 psr=00000003
mode=1 ie=1 irql=0
```

I never needed to know what `softfpcr` is. The `psr` is the processor status register, the `mode` is 1 for user mode and 0 for kernel mode, `ie` is the interrupt enable flag, and `irql` is the interrupt request level.

The calling convention is simple. As noted in the tables above, parameters are passed in registers, with excess parameters spilled onto the stack. There is no home space. The return address is passed in the *ra* register, and the stack must be kept aligned on a 16-byte boundary. Exception dispatch is done by unwind tables stored in a separate section of the image.

Okay, that's the register set and calling convention. Next time, we'll look at integer operations.

Exercise: The x64 calling convention reserves home space so that the register-based parameters can be spilled onto the stack and remain contiguous with the other stack-based parameters, so that the entire parameter pack can be enumerated with the `va_start` family of macros. Why doesn't this requirement apply to the Alpha AXP?

[1] The term *octaword* was introduced later, but we are focusing on the Alpha AXP classic architecture.

Raymond Chen

Follow