RISC vs. CISC Still Matters

In the early 1980s the computer industry was gripped by a great new idea. It was called Reduced Instruction Set Computing, or RISC, an acronym coined by David Patterson, a computer architecture researcher and professor at the University of California at Berkeley. The term originally applied to the concept of designing computers with a simplified and regular instruction set architecture (ISA) that could be implemented efficiently with minimal overhead (control logic) and be amenable to such performance enhancing techniques as instruction processing pipelining and high processor clock rate.

In addition, the ISA would be deliberately designed to be an efficient and transparent target for optimizing compilers. In fact the compiler would be a crucial element as it absorbed various vital but complex and burdensome functions traditionally supported in hardware, such as synthesizing complex program functions and addressing modes from sequences of the more elementary arithmetic and logic functions directly performed by digital circuitry. Unlike most earlier computers, RISC architects fully expected their progeny would only be programmed in high level languages (HLLs), so being a good compiler target was the goal rather than being conducive to assembly language programming.

The primary commandment of the RISC design philosophy is *no instruction or addressing mode whose function can be implemented by a sequence of other instructions should be included in the ISA unless its inclusion can be quantitatively shown to improve performance by a non-trivial amount, even after accounting for the new instruction's negative impact on likely hardware implementations in terms of increased data path and control complexity, reduction in clock rate, and conflict with efficient implementation of existing instructions*. A secondary axiom was that a RISC processor shouldn't have to do anything at run time in hardware that could instead be done at compile time in software. This often included opening up aspects of instruction scheduling and pipeline interlocking to the compilers code generator that were previously hidden from software by complex and costly control logic.

By the mid-1980's the benefits of RISC design principles were widely understood and accepted. Nearly every major computer and microprocessor vendor developed new processors based on the RISC principles and the resulting designs were all remarkably similar. These ISAs had large general purpose register files with 32 addressable registers (one of which always read as all zeroes), uniformly sized instructions 32-bits in length, few instruction formats, only one or two addressing modes, and complete separation between instructions that compute and instructions that access memory (loads and stores). Soon the term RISC became synonymous with computer designs that shared some or most of these attributes.

When the term RISC was introduced a second term was created, Complex Instruction Set Computing, or CISC, which was basically a label applied to the existing popular computer

architectures such as the IBM S/370, DEC VAX, Intel x86, and Motorola 680×0. Compared to the remarkably similar ISAs of the self-proclaimed RISC architectures, the CISC group was quite diverse in nature. Some were organized around large general purpose register files while others had just a few special purpose registers and were oriented to processing data *in situ* in memory. In general, the CISC architectures were the product of decades of evolutionary progress towards ever more complex instruction sets and addressing modes brought about by the enabling technology of microcoded control logic, and driven by the pervasive thought that computer design should close the "semantic gap" with high level programming languages to make programming simpler and more efficient.

In some ways CISC was a natural outgrowth of the economic reality of computer technology up until the late 1970's. Main memory was slow and expensive, while read only memory for microcode was relatively cheap and many times faster. The instructions in the so-called CISC ISAs tend to vary considerably in length and be tightly and sequentially encoded (i.e. the instruction decoder had to look in one field to tell if a second optional field or extension was present, which in turn would dictate where a third field might be located in the instruction stream, and so on).

For example, a VAX-11 instruction varied in length from 1 to 37 bytes. The opcode byte would define the number of operand specifiers (up to 6) and each had to be decoded in sequence because there could be 8, 16, or 32 bit long displacement or immediate values associated with each specifier. This elegant scheme is a delight for VAX assembly language programmers, because they could use any meaningful combination of addressing modes for most instructions without worrying if instruction X supported addressing mode Y. However, it would become a major hurdle to the construction of high performance VAX implementations within a decade after its introduction.

Other CISC architectures, like x86, had a simpler and less orthogonal set of addressing modes but still included features that contributed to slow, sequential instruction decode. For example, an x86 instruction opcode could be preceded by an optional instruction prefix byte, an optional address size prefix byte, an optional operand size prefix byte, and an optional segment override prefix byte. Not only are these variable length schemes complex and slow, but are also susceptible to design errors in processor control logic. For example, the recent "FOOF" bug in Intel Pentium II processors was a security hole related to the "$F0_{16}$" lock instruction prefix byte wherein a rogue user mode program could lock up a multi-user system or server.

To illustrate the large contrast between the instruction encoding formats used by CISC and RISC processors, the instruction formats for the Intel x86 and Compaq Alpha processor architectures are shown in Figure 1. In the case of x86 there is a lot of sequential decoding that has to be accomplished (although modern x86 processors often predecode x86 instructions while loading them into the instruction cache, and store instruction hints and boundary information as 2 or 3 extra bits per instruction byte). For the Alpha (and virtually every other classic RISC design) the instruction length is fixed at 32-bits and the major fields appear in the same locations in all the formats. It is standard practice in RISC processors to fetch operand data from registers (or bypass paths) even as the instruction opcode field is decoded.

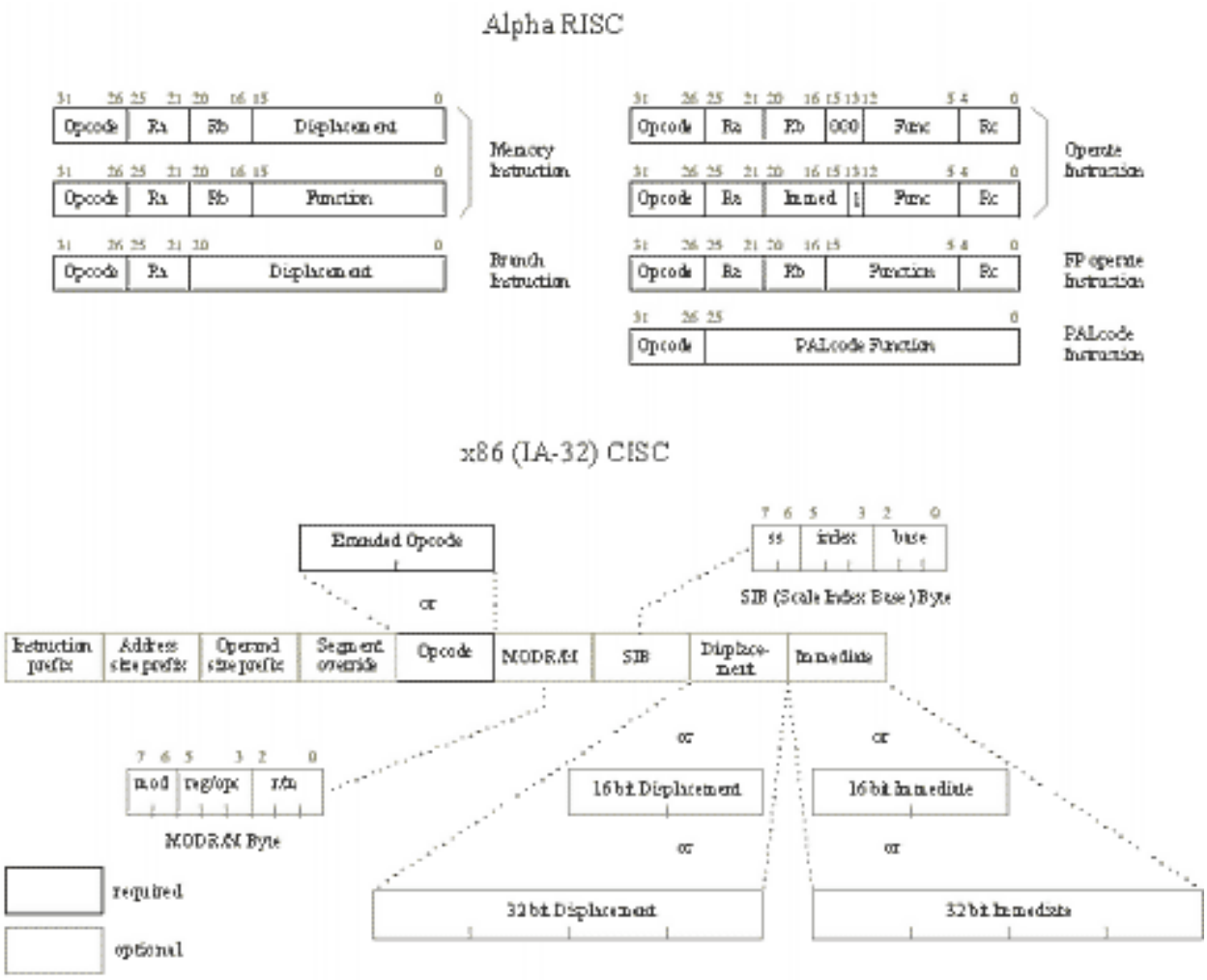

Figure 1 Instruction Encoding of Representative RISC and CISC

When RISC processors first appeared on the scene most CISC processors were microcoded monsters with relatively little instruction execution pipelining. Processors like the VAX, the 68020, and the Intel i386 for the most part processed only one instruction at a time and took, on the average, five to ten clock cycles to execute each one. The first RISC processors were fully pipelined, typically with five stages, and averaged between 1.3 and 2.0 clock cycles to execute an instruction. RISC-based microprocessors typically were more compact and had fewer transistors (no microcode) than their CISC counterparts, and could execute at higher clock rates. Although programs compiled for RISC architectures often needed to execute more native instructions to accomplish the same work, because of the large disparity in CPI (clocks per instruction), and higher clock rates, the RISC processors offered two to three times more performance. In Table 1 is a case study comparison of an x86 and RISC processor of the early RISC era (1987).

	Intel i386DX	**MIPS R2000**

Technology	1.5 um CMOS	2.0 um CMOS
Die Size	103 mm^2	80 mm^2
Transistors	275,000	115,000
Package	132 CPGA	144 CPGA
Power (Watts)	3.0	3.0
Clock Rate (MHz)	16	16
Dhrystone MIPs	5.7 [1]	13.9 [2]
SPECmark89	2.2 [1]	10.1 [2]
Note	[1] with 64 Kbyte external cache	[2] with 32 Kbyte external cache

In case study 1 the huge advantage of the RISC design concept for the upcoming era of VLSI-based microprocessors is clear. The MIPS R2000 processor is a smaller device built in an older semiconductor process with less than half the number of transistors as the Intel i386DX, yet it blows its CISC counterpart right out of the water in performance: more than twice the Dhrystone MIPS rating and more than four times the SPECmark89 performance (even with a smaller external cache).

The Empire Strikes Back

As can be imagined, the emergence of RISC, with its twin promises of simpler processor design and higher performance, had an energizing effect on the computer and microprocessor industry. The first CISC casualties of the RISC era, unsurprisingly, were in the computer markets most sensitive to performance and with the most portable software base. The VAX architecture was already facing stiff competition from mid-range systems based on standard high volume CISC microprocessors when RISC designs like MIPS, SPARC, and PA-RISC came along to administer the final blow. In the technical workstation market the Motorola 680X0 CISC family was easily overwhelmed by much faster RISC-based systems such as the ubiquitous SPARCstation-1.

The one market that RISC never got even a toehold in was the IBM-compatible PC market. Even the most popular RISC processors were manufactured in much smaller quantities than Intel x86 devices and could never effectively reach high volume PC price points without crippling their performance. Even if they could be built as cheaply as x86-based PCs, RISC processors couldn't tap into the huge, non-portable software installed base except under emulation, which more than wiped out the RISC performance advantage. And much credit must go to Intel Corporation. It aggressively invested in both developing complex and innovative new ways of implementing the hopelessly CISC x86 ISA, and ensuring these would be implemented in each new generation of CMOS processes one or two years before any RISC processor. The potent combination was sufficient to ensure that x86 processors were never behind RISC processors in integer performance by a factor of two or more. This was a sufficient factor to ensure the continued loyalty of independent software vendors (ISVs) offering PC-based applications. An uneasy peace settled in between the two solitudes of x86 PCs and RISC high-end servers and workstations until late 1995 when the Intel Pentium Pro (P6) processor appeared.

The launch of the Pentium Pro processor was the computer industry equivalent of a Pearl Harbor type surprise attack directly against the RISC heartland: technical workstations and servers. The Pentium Pro combined an innovative new out-of-order execution superscalar x86 microprocessor with a separate high speed custom SRAM cache chip in a multi-chip module (MCM)-like package. The biggest surprise of all was the fact that the 0.35 um version debuted simultaneously with the expected 0.5 um version and more than six months ahead of Intel's public product roadmap. This allowed the Pentium Pro to reach a clock speed of 200 MHz and integer performance levels that briefly eclipsed the fastest shipping RISC processor, the 0.5 um Alpha 21164.
The Pentium Pro's integer performance lead didn't last long and its floating point performance was still well behind nearly every RISC processor but this didn't reduce the psychological impact. Any hope that RISC microprocessor vendors had of being able to reach PC price points with their much smaller volume chips, while offering a large enough integer performance advantage (x86 processors already provided sufficient floating point for nearly all PC-type applications) to entice the market away from x86 was pretty well extinguished.

The last few years have not been kind to the leading high-end RISC processor architectures like MIPS, PA-RISC, SPARC, Alpha, and POWER/PowerPC. While low-end embedded RISC microprocessors were making huge headway in displacing the 680×0 family on the basis of high performance, low power, and compact die size, their larger, more complex, and lower volume brethren were under continuous attack in the low-end server and workstation market by x86 CISC processors. Today, 0.18 um x86 processors from two different vendors are yielding integer performance levels neck and neck with the fastest RISC processor, the Alpha EV67, and well ahead of all other RISC designs.
At the same time Intel is reaping the benefit of a long-term effort at convincing the marketplace that the difference between RISC and CISC processors was somehow shrinking. This first started when Intel released its i486 processor, and it was widely reported as having a "RISC integer unit". Despite the utter meaninglessness of this claim (does the 486 execute RISC integer instructions?), it was the thin edge of the wedge, and the beginning of a growing period of intellectual laziness within the computer trade press that has largely corrupted the terms RISC and CISC for most of the computer buying populace to this day.
The campaign to obfuscate the clear distinction between RISC and CISC moved into high gear with the advent of the modern x86 processor implementations employing fixed length control words to operate out-of-order execution data paths. These wide packets of control information would have been called microcode words a decade ago. However, x86 vendors have cleverly named them micro-ops (uops), RISC-ops (R-ops), or even RISC86 instructions in order to draw the inference that these are equivalent to the instructions of a RISC ISA. The popular technical press, like PC Magazine, latched onto this and routinely print side bars to CPU stories that explained how new processors like the Pentium II and the K7 are really just RISC processors with a bit of stuff in the first stage or two of the pipeline to handle that nasty old complex variable length x86 instruction set.

The primary fallacy of the “RISC and CISC have converged” school of thought is to ignore the distinction between an instruction set architecture (ISA) and the internal microarchitecture of an actual processor implementation. RISC and CISC refer to ISAs, which are abstract models of computer architectures as seen by the programmer. An ISA includes the programmer and compiler visible state of a computer, including all registers and flags, the encoding and semantics of all instructions, exception handling, and memory organization and semantics (little-endian vs big-endian, weakly-ordered vs strongly-ordered). An ISA *does not* tell computer engineers how an implementation must be realized.

Today’s modern x86 CISC microprocessor and high-end RISC microprocessor share a great deal of implementation details and are built using similar functional building blocks. The integer out-of-order execution back end of a Pentium II/III or K7 Athlon processor with its large group of renaming registers *does* closely resemble the integer data path of a RISC processor. This similarity is a major reason why x86 processors haven’t fallen hopelessly behind the performance of RISC microprocessors, but CISC processors still pay a large complexity tax. A modern x86 processor requires several extra pipe stages and about 40% larger instruction cache to analyze the variable length x86 instruction set and store pre-decode information. The x86 instruction decoders themselves consume one or two million transistors and are quite complex and prone to design errors that are only partially correctable using patchable microcode store. The modern x86 back-end execution engine (the so-called “RISC execution unit”) also has to devote extra resources to handle instruction dependencies related to condition codes and ensuring exceptions encountered, while processing micro-ops can be reported back as precise exceptions within the context of the originating x86 instruction.

The proponents and popularizers of the “RISC and CISC have converged” school of thought are so caught up in comparing chip organization and micro-architecture that they miss the big picture. The benefit of RISC ISA-based processor design comes in two separate packages. They focus on the first package: the ease of design of simplified and fast hardware. The era of 10 and 15 million transistor chips with three and four way superscalar issue and out-of-order execution has somewhat reduced (but not eliminated) this benefit, because in a sense all these chips, RISC and CISC alike, are damn complicated!

But the second benefit of RISC is the computational model – the ISA – it offers to the compiler. A RISC ISA offers a streamlined and simplified instruction set, and a generous set of general purpose registers. Most RISC designs do away with condition codes and instead rely on either storing Boolean control information in general purpose registers, atomically combining comparison and branch operations in single instructions, or a combination of both. In Figure 2 are the programmer’s visible register resources of the x86 and Alpha ISAs. The bottom line is that the x86 has 8 general purpose integer registers, while RISC processors have 32. Ironically, both modern x86 and RISC processors have even far more physical data registers in them than shown here to accomplish register renaming, a powerful design tool used to eliminate the effect of false dependencies between

instructions that would otherwise prevent out-of-order execution. However, it is the computational model seen by the compiler that is critical for the generation of ultra fast code.

The modern compiler is, in many ways, as complex and fascinating as the processors it creates code for. But the vital ingredient that allows a sophisticated compiler to excel is a large and unencumbered register set. A large register set facilitates such powerful optimization techniques as local and global variable register assignment, register-based parameter passing and function result return, and re-use of intermediate computational results from the calculation of common sub-expressions. In addition it is well known that because of loops, roughly 90% of program time is spent executing 10% of code. RISC ISAs, with their large register sets, support powerful loop-based optimizations such as array index address calculation strength reduction, software pipelining, and loop unrolling. Besides the large register sets, most RISC ISAs also incorporate three address instructions, that is, instructions that specify three registers – two source and one destination. The x86 and nearly every other CISC ISA use only one or two address instructions which means that extra move instructions are needed when it necessary not to overwrite either of two operand registers.

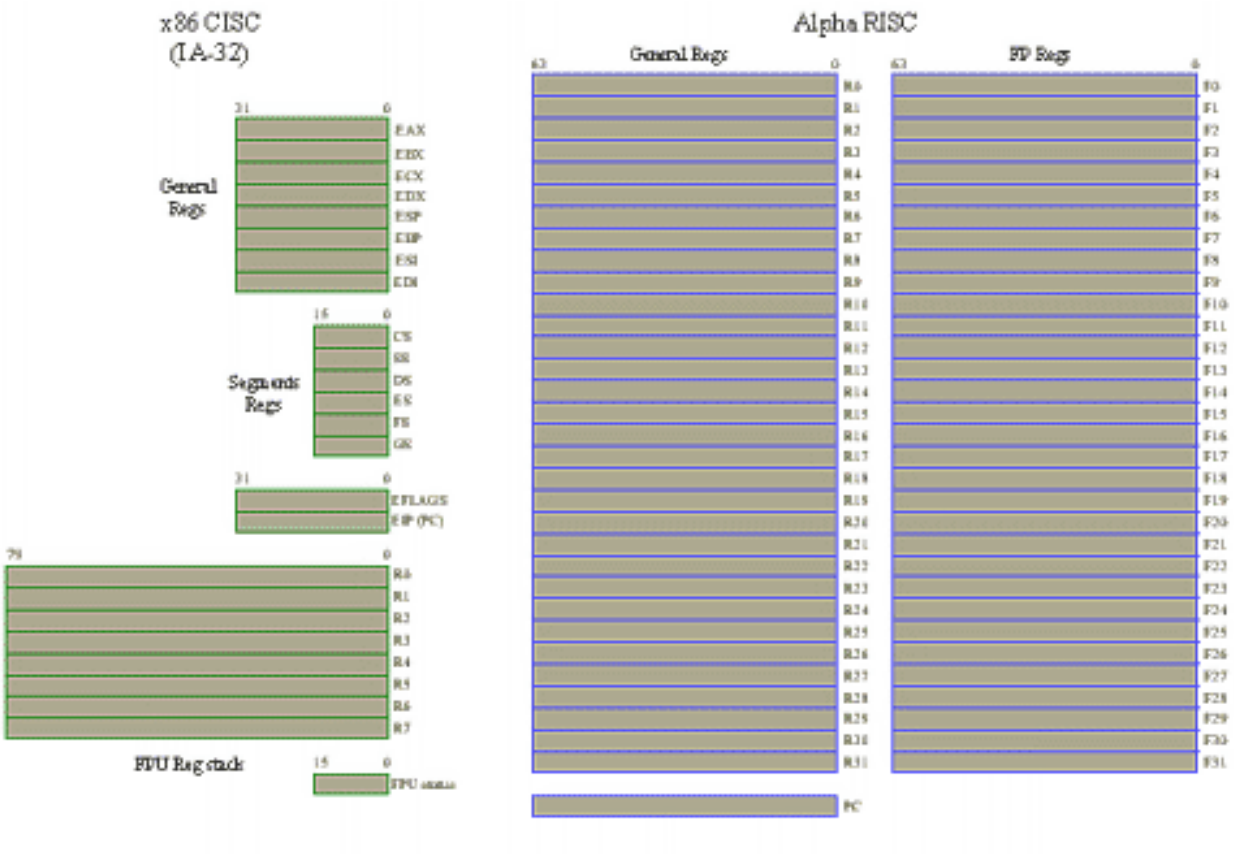

Figure 2 The Computational State: What the Compiler Has to Work With

The power and efficiency of the large register sets and three address instructions found in RISC ISAs is most clearly demonstrated when it comes to floating point performance. The x86 ISA uses an antiquated 8 element register stack computational model for its floating point instructions, which

is quite inferior to the familiar RISC ISA model with large floating point register sets and three address floating point instructions. Because of these architectural differences most high-end RISC processor families outperform even the best x86 processors by a factor of 2x, 3x, or more on floating point benchmark programs despite both having fully pipelined floating point execution units with similar latencies. This effect is so great that Advanced Micro Devices (AMD) has announced that it is adding a RISC style floating point computational model to its x86-64 architecture, a 64 bit extension of the existing x86 ISA to help close the performance gap with RISC processors.

The importance of architectural differences between RISC and CISC ISAs to compiled code performance is demonstrated in Figure 3. A simple subroutine was written in the C programming language to find the element in a list of integers with the most number of bits set. Along with the source code is the assembly language code equivalent generated by compilers for the HP PA-RISC processor and Intel x86. Both the PA-RISC and x86 compilers were run with maximum optimization for code speed.

C Source Code

```
most_ones (buff, n, pi, pnb)

int buff [], n, *pi, *pnb;
{
  int i, j, nb, t, i_best, nb_best;

  for (i = 0; i < n; i++)
  {
    nb = 0;
    t = buff [i];
    for (j = 0; j < BITS; j++)
    {
      nb += (t & 1);
      t >>= 1;
    }
    if (i == 0 || nb > nb_best)
    {
      i_best = i;
      nb_best = nb;
    }
  }
  *pi = i_best;
  *pnb = nb_best;
  return (nb_best);
}
```

PA-RISC Code

```
    ldi  0,r29
    comb,<=  r25,r29,L$0004
    stw  r2,-20(0,r30)
    ldi  31,r31
    ldi  0,r22
L$0015
    ldws,ma  4(0,r26),r21
    copy  r22,r20
L$0010
    extru  r21,31,1,r19
    extrs  r21,30,31,r21
    ldo  1(r20),r20
    comb,>=  r31,r20,L$0010
    addl  r22,r19,r22
    comiclr,=  0,r29,0
    comb,>=,n  r28,r22,L$0005
L$0013
    copy  r29,r2
    copy  r22,r28
L$0005
    ldo  1(r29),r29
    comb,>  r25,r29,L$0015
    ldi  0,r22
L$0004
    stw  r2,0(0,r24)
    stw  r28,0(0,r23)
    ldw  -20(0,r30),r2
    bv,n  0(r2)
```

x86 Code

```
    sub  esp,4
    push ebx
    push esi
    push edi
    xor  esi,esi
    cmp  DWORD PTR _n$[esp+12],esi
    push ebp
    jle  SHORT $L115
    mov  edx,DWORD PTR _buff$[esp+16]
    mov  eax,DWORD PTR _nb_best$[esp+20]
$L95: xor  edi,edi
    mov  ebx,DWORD PTR [edx+esi*4]
    mov  ecx,32
$L98: mov  ebp,ebx
    sar  ebx,1
    and  ebp,1
    add  edi,ebp
    dec  ecx
    jne  SHORT $L98
    test esi,esi
    je   SHORT $L102
    cmp  eax,edi
    jge  SHORT $L96
$L102: mov  eax,edi
    mov  DWORD PTR _i_best$[esp+20],esi
$L96: inc  esi
    cmp  esi,DWORD PTR _n$[esp+16]
    jl   SHORT $L95
    jmp  SHORT $L97
$L115: mov  eax,DWORD PTR _nb_best$[esp+20]
$L97:  mov  ecx,DWORD PTR _pi$[esp+16]
    mov  edx,DWORD PTR _i_best$[esp+20]
    mov  ebx,DWORD PTR _pnb$[esp+16]
    pop  ebp
    pop  edi
    mov  DWORD PTR [ecx],edx
    pop  esi
    mov  DWORD PTR [ebx],eax
    pop  ebx
    add  esp,4
    ret  0
```

Figure 3 Effect of Register-based Parameter Passing

Note: in terms of relative code density this program is not representative; it is generally accepted that most RISC processors have program code sizes at least 30% higher than x86 on average).
The RISC code benefits from register-based parameter passing to minimize memory traffic and superfluous instructions associated with stack based argument passing. The PA-RISC subroutine includes 23 instructions that total 92 bytes in size. The x86 subroutine uses 41 instructions that total 97 bytes in size. For a modern x86 processor whose instruction cache incorporated three bits of pre-decode and hint information per program byte this subroutine would occupy 45% more on-chip SRAM bits than the PA-RISC program would in a PA-8500.

The heated competition between Intel and AMD for the lucrative and high volume PC marketplace has pushed x86 CISC ISA-based microprocessors into 0.18 um CMOS processes well ahead of any RISC ISA-based processor. Besides higher clock rates, 0.18 um CMOS also permits the integration of large (256 Kbyte) and highly associative, low latency L2 caches within the processor. As a result, x86 processors have temporarily eclipsed the integer performance of virtually every RISC processor. However, as always, x86 processors are hopelessly behind nearly every non-embedded control RISC processor family in floating point performance.
There is no doubt about the power and influence of the x86 processor market within the semiconductor industry. Last year over 100 million x86 processors were sold bringing in well over $20 billion in revenue and $ billions in profit. Contrast that to, say, the Compaq Alpha processor, which might sell in the several hundred thousand devices per year range and bring in several hundred millions of dollars of revenue to Compaq’s semiconductor partners. This is why new x86 cores come to market at a much faster pace than high-end RISC processors and transition to newer and better semiconductor processes with less delay.
In Table 2 is a case study comparison of best-of-breed modern x86 and RISC processor designs built in approximately equivalent CMOS process technologies. This comparison is made more intriguing by the fact that several key designers of the EV67 core (Jim Keller and Dirk Meyer) left Compaq/DEC and went on to help design the K7. Also, the two processors share the same system bus architecture.

	AMD K7 Athlon	**Compaq Alpha EV67**
Technology	0.25 um CMOS	0.28 um CMOS
Die Size	184 mm^2	205 mm^2
Transistors	28.1 million	15.2 million
Package	240 SEC module	588 CPGA
Power (Watts)	50	75
Clock Rate (MHz)	700	700
SPECint95	31.7 [1]	39.1 [2]

SPECfp95	24.0 [1]	68.1 [2]
Note	[1] with 512 Kbyte external cache	[2] with 8 Mbyte external cache

Although it appears that the performance gap has mostly closed for integer code, faster 0.28 um EV67s (reportedly 833 MHz) are already in beta testing in systems at customer sites while faster K7s require a more advanced 0.18 um process. Unsurprisingly, the huge floating point performance gap is still present, although some of that is attributable to the disparity in L2 cache size. Also note that the K7 has almost twice as many transistors as the EV67 despite the fact both designs implement 64 Kbyte instruction and data caches. This reflects the CISC “complexity tax” imposed on modern x86 processor designs regardless of the similarities in the back end execution engine.

Conclusion

While the integer performance gap between the best RISC and CISC processors has closed over the last thirteen years, the deep and fundamental differences between the two architecture design concepts have not. The “RISC and CISC are converging” viewpoint is a fundamentally flawed concept that goes back to the i486 launch in 1992 and is rooted in the widespread ignorance of the difference between instruction set architectures and details of physical processor implementation. Modern out-of-order execution x86 and RISC processors *do* have very similar organization in their back end execution engines, both of may which contain 40 or more physical renaming registers. While RISC data paths are driven directly by RISC instructions, x86 data paths are similarly driven by sequences of simple, shallowly encoded microcode-like control words called micro-ops, or provocatively, RISC-ops.

Even if the physical implementation advantages RISC designs enjoy over CISC designs, like the x86, could be reduced to zero (a prospect which is demonstrably remote), it doesn’t change the fact that the ISA, the programming model targeted by compilers, is vastly different. The modern x86 processor might have 40 physical general purpose registers for renaming, but the compiler can only target the 8 GPRs visible in the ISA. It doesn’t matter that the modern x86 processor has a back-end execution engine that is controlled by RISC-like control words; these micro-ops are inaccessible from the outside world and the compiler cannot target them. The x86 compiler cannot perform many of the standard RISC compiler optimization techniques that strongly depend on large register sets, three address instruction formats, and the absence of non-register based dependencies between instructions.

This fundamental and inescapable nature of the difference between RISC and CISC computer design is the driving force behind both Intel’s development of the 64-bit RISC-like IA-64 family of processors to eventually replace x86, and AMD’s decision to add a RISC-like large flat floating point register file and three address floating point instructions to beef up the performance of the 64-bit extended x86-64 ISA for its upcoming K8 processor.