VIEWPOINT

RISC-like Design Fares Well for x86 CPUs

Argument for Native Design Is Overblown; Advantages Nonexistent

By Mike Johnson, AMD

A recent article by Mark Bluhm and Ty Garibay of Cyrix *(see **0912VP.PDF**)* purported to argue certain advantages of a "native design," presumably represented by Cyrix's M1 (6x86), over the "RISC-like" designs from all of Cyrix's main competitors. Although this is mostly a religious debate—and heaven knows it's about time for another version of the RISC-vs-CISC debate, now that the original one is largely decided—this article is misleading in several respects.

Stalking a Paper Tiger

Bluhm and Garibay base most of their argument on the complexity of a RISC-like processor they postulated, shown in Figure 1. This design, however, is unlike any existing RISC-like design that has been announced. The design in the figure is a composite of features found in AMD's K5 and Intel's Pentium Pro, neither of which is a two-issue design, as implied by the figure. This figure appears intended to show such designs in the worst possible light. It certainly misrepresents the issue bandwidth available in the processors being criticized, which is reason enough to complain. But there are many other inaccuracies. I'll point these out relative to the AMD-K5.

First, the K5 doesn't have anything corresponding to the "Xbar3" shown in the figure, because the K5 uses distributed reservation stations. The authors state that a centralized reservation station is a feature of a RISC-based design, whereas "distributed reservation stations can be added transparently to the native design." Certainly, they can't really believe this. Only Pentium Pro has this organization, and the K5 has distributed reservation stations that are just as "transparent" as anyone's.

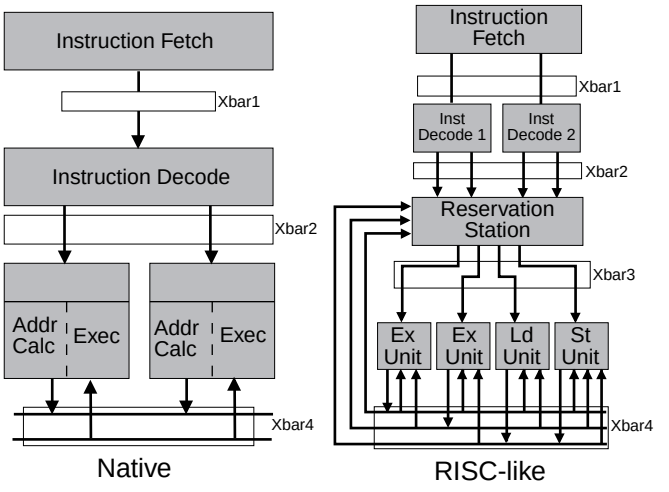

Figure 1. Cyrix compared a native x86 processor to a RISC-like processor of its own invention that does not resemble actual RISC-like x86 designs. (Source: Cyrix)

This argument is overblown on another count. Interconnect is a major concern in any implementation, but the authors have grossly inflated the interconnect in the RISC-based design by treating all global interconnect as "crossbars." The AMD-K5 implements most of these crossbars using global shared buses. For example, a single decode position has a single, dedicated bus to communicate commands and data to all function units, each of which is connected to the same bus. There are four such buses, because there are four decode positions. Since each bus connects to six function units, the authors would conclude that this represents 24 "interconnects."

This line of reasoning inflates the cost of interconnect from one that is on the order of the number of decoders to one that is on the order of the number of decoders times the number of function units. I can think of no justification for expressing the cost in these terms, except to make it look as large as possible. Certainly there is additional cost for each function unit that is attached, but this is comparable to the cost of local interconnect, which the authors dismiss in their own design.

Native Design Is Also Complex

Let's turn our attention to the other side of this argument: the advantages of a native design. Here, too, there are inconsistencies in the authors' position. It is not possible to achieve any appreciable level of parallelism in the native design shown in Figure 1 because there is insufficient communication between the native function units. For example, Figure 2 (taken from the original article, ***0912VP.PDF***) shows an example of the parallel execution of two complex instructions. This parallel execution, however, relies on the output of an add instruction feeding the second operation in a read-modify-write sequence (a register-to-memory subtract). The interconnect necessary for this communication isn't accounted for in Figure 1.

Furthermore, consider what would happen if the first instruction were merely a memory-to-register move (a more common way to start an x86 computational sequence). In this case, parallel execution would call for the load to directly feed the subtract, requiring communication from the address unit/data cache in the first pipeline to the execution unit in the second pipeline. This

communication might conceivably be handled by Xbar4, but this would cause conflicts for this interconnect with previous instructions that use Xbar4 for writeback.

The crossbars in the RISC-like version serve an essential purpose: communicating data between parallel computations. In the native design, the forwarding paths that appear within the native function units must be duplicated between function units. If this interconnect is not provided, the native dual-pipeline design doesn't even run as fast as a scalar pipelined design, because the forwarding paths that are available within a single pipeline aren't available between pipelines. Alternatively, a single pipeline must be used most of the time to gain access to the local forwarding paths, defeating the purpose of having dual pipelines. Parallel computation requires parallel communication, and the fact that these communication paths are omitted from Figure 1's native design suggests that either the figure is incorrect or the design is much slower than implied.

Most x86 Code Is RISC-like

Finally, native function units are used effectively only when there is an appreciable number of complex instructions in the dynamic instruction stream. Unfortunately, most dynamic x86 code, particularly so-called 32-bit code, maps directly to single RISC-like operations (ROPs). The reason is very simple, one that has been known for more than 20 years: compilers have a hard time generating complex instructions. Instead, they usually generate code sequences from basic instructions.

The timing diagram in Figure 3 shows an execution fragment of an MPEG decoder (Huffman decoding of an interblock frame) as executed by the K5. In this diagram, 24 x86 instructions require 12 cycles to execute—a rate of 2 x86 instructions per cycle. In this sequence, only 7 instructions are "complex," and those instructions all generate two-ROP sequences.

A native design organized around basic read-modify-write sequences would have hardware idle in its function units most of the time, because most instructions can't use the full capability of these units. In the RISC-like design, the function units are made available across the operations of all instructions, regardless of their origin.

In the native design, furthermore, a multicycle operation like multiplication (e.g., the fourth instruction in Figure 3) ties up the entire execution unit, and all its resources, for the duration of the calculation. In the RISC-like design, the multiplier is a separate function resource that doesn't block the execution of independent instructions. In this example, the K5 achieves fully overlapped execution with the multiplier. It's hard to imagine getting close to this throughput with a native design, because its resources aren't balanced to the needs of the instructions but instead are designed for and dedicated to the worst-case sequence (a read-modify-write).

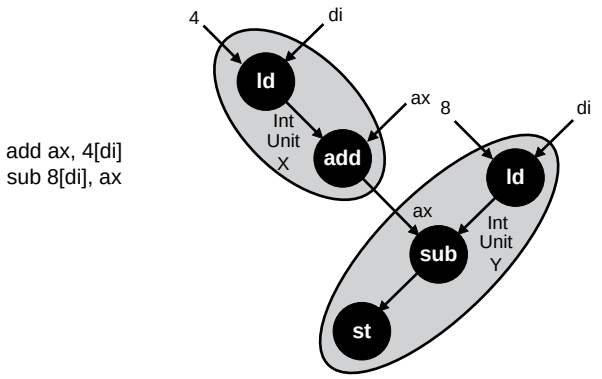

Figure 2. In a native x86 processor, the operations required to complete each instruction are all handled by a single execution unit. (Source: Cyrix)

Neither Choice Is Superior

So is a native design the wrong choice? It depends. I think Cyrix chose a native design to leverage its experience with simple, scalar pipelines. There is nothing at all wrong with making these sorts of tradeoffs in favor of reduced time-to-market. To claim all these other benefits is unwarranted, however, particularly because they don't exist. ♦

Mike Johnson is the chief architect of AMD's K5 and superscalar 29K processors. He is also the author of the book Superscalar Microprocessor Design.

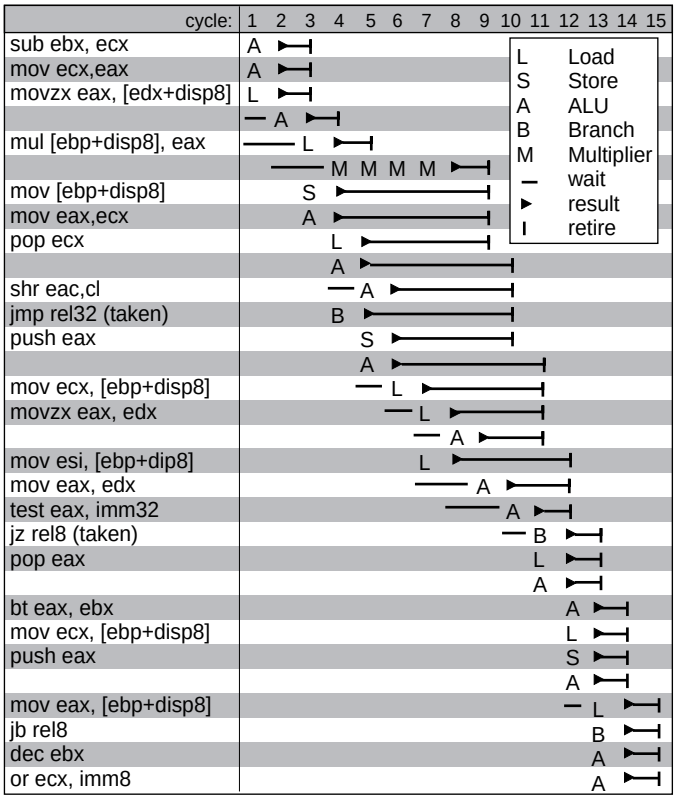

Figure 3. An execution fragment from AMD's K5 processor shows that RISC-like designs can achieve significant parallelism. (Source: Dave Christie, AMD)

