VIEWPOINT

Going Native!

x86-Optimized Designs Have a Strong Growth Path

by Mark Bluhm and Ty Garibay, Cyrix

Several recently announced x86 processors utilize a design style that translates x86 instructions into primitive operations before executing them, a style we refer to as "RISC-like." These processors include Intel's P6, AMD's K5, and NexGen's Nx586. Intel and HP are rumored to be collaborating on the design of a processor using a second style based on VLIW principles. In contrast, Cyrix has chosen a third style of execution for its highest-performing processor, the M1. We refer to this style as "native."

Although the RISC-like design style can be used to build high performance x86 implementations, the native style delivers performance equivalent to or better than the RISC-like approach and, unlike the VLIW approach, retains compatibility with the existing code base. Furthermore, the native approach yields a simpler design that reduces cost and design time.

Native vs. RISC-like

A native design and a RISC-like design execute x86 instructions in very different ways. We can contrive a simple example to illustrate their differences:

```
1. ADD     ax, 4[di]          ; Add memory to register ax
2. SUB     8[di], ax          ; Subtract ax from memory
```

In this example, the SUB instruction is dependent on the result of the ADD instruction, so in either design approach these two instructions must execute sequentially.

In a RISC-like design, the two instructions are translated into a sequence of RISC-like operations:

```
1. LOAD    temp1, 4[di]       ; Load memory into temp1
2. ADD     ax', ax, temp1     ; Register ax' gets ax + temp1
3. LOAD    temp2, 8[di]       ; Load memory into temp2
4. SUB     temp3, temp2, ax'  ; temp3 gets temp2 - ax'
5. STORE   temp3, 8[di]       ; Store result to memory
```

This decomposition allows operations to be grouped by type (i.e., load/store, integer ALU, etc.) across instruction boundaries. As a result, the RISC-like design can more closely match the number and types of function units to the expected distribution of RISC-like operations.

This decomposition is not without cost, however. A direct consequence of allowing any operation, decomposed from x86 instructions, to execute within specific function units is that any function unit completing an operation may potentially modify an x86 register. To communicate new register values to dependent operations in other function units, all function units must be fully interconnected. In the worst case, each RISC-like operation must be assumed to be updating a programmer-visible register. Therefore, both temporary and programmer-visible results are passed between all function units to guarantee correct results.

In a native implementation using two integer units, these two instructions execute as shown in Figure 1. Each function unit in the native design is a full integer unit with all capabilities required to execute each of these simple x86 instructions in their entirety. The results that are passed between the two function units and eventually to memory are only those values that are programmer visible.

In contrast to the RISC-like approach, the native design treats each x86 instruction separately, executing each instruction in its own pipeline. Only the results of actual x86 instructions are distributed between the integer units. Therefore, for this example, the RISC-like design requires 150% more communication between function units to achieve the same performance. This added communication is endemic to the RISC-like design.

RISC-like Design Increases Complexity

Figure 2 shows the block diagrams of two comparable two-way superscalar x86 microprocessor implementations, one native and one RISC-like. Each can decode and issue up to two x86 instructions per cycle. Aside from this similarity, the two implementations differ greatly.

In the figure, a box labeled Xbar (crossbar) has been placed at each point in the designs where significant communication is required between blocks. Each connection between an input and an output of each crossbar

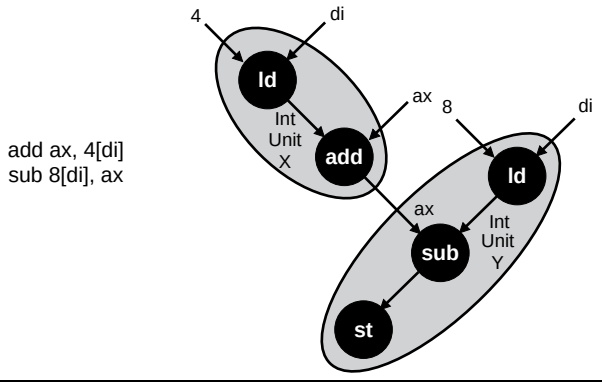

Figure 1. In a native x86 design, all operations required to complete each instruction are performed in tightly coupled "macro" function units.

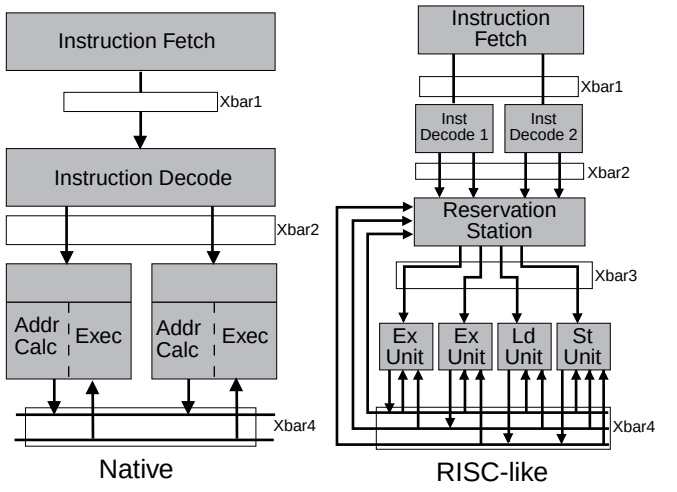

Figure 2. A native x86 processor design requires much less interconnect than a RISC-like version.

represents an average of 40 wires (sometimes control, sometimes data), so the number of unique wires explodes quickly. (Forty wires per crossbar is a conservative estimate derived from the actual number of wires used in existing 32-bit processor implementations.) Wires are the enemy when one is attempting to reduce design time, increase frequency, and lower cost.

Early RISC proponents singled out transistor count as the main culprit of the high cost and complexity of CISC microprocessors; this is no longer the issue today. The inexorable march of semiconductor technology has made it possible to place a greater number of transistors on a die, leaving only the problem of connecting them as the stumbling block.

In both designs, the size of Xbar1 is dependent only on the number of x86 instructions to be decoded in a single cycle, so Xbar1 is of degree 2 in both the native and RISC-like implementations. (A crossbar of degree 2 has two inputs and two outputs, with a total of four connections). The boxes labeled Xbar2 in each design represent the communication required to issue two x86 instructions worth of work in each clock cycle. Since the native design issues each x86 instruction as a unit, Xbar2 in the native case is the same size as Xbar1: degree 2. The RISC-like design allows for a 2:1 expansion of ROPs from x86 instructions, so Xbar2 in the RISC-like case is of degree 4.

Even with this dramatic increase in communication (a degree 4 crossbar is four times as complex as a degree 2 crossbar), this particular RISC-like design could not issue the two instructions from the preceding example, the ADD and the SUB, in parallel. These two instructions require an Xbar2 of degree 5 for the RISC-like design to equal the performance of the native design.

Because of the inherent symmetry in a native design (i.e., the number of fully functional integer units is equal to the maximum number of x86 instructions issued per clock), Xbar2 is not required in the native implementation; Xbar2 provides only load balancing between the integer units. In the RISC-like design, however, Xbar2 is required due to its single centralized reservation station. In contrast, distributed reservation stations can be added transparently to the native design by placing them inside each integer unit. Therefore, in the native implementation, no extra communication is required to allow full out-of-order, dataflow instruction execution.

The native design requires one type of communication that does not arise in the RISC-like case: each integer unit has at least one link between the address calculation (AC) unit and the execution (EX) unit inside it. Therefore, the native design has two connections absent in the RISC-like design; however, each of these connections is local to the integer unit and is less costly than global communication between function units.

Xbar3 exists only in the RISC-like design. This crossbar is the mirror image of Xbar2 in the RISC-like design and is required to distribute ROPs from the central reservation station to each of the function units. Since this example RISC-like design is attempting to maintain a throughput of four ROPs per cycle, Xbar3 is of the same size as Xbar2: degree 4.

Finally, Xbar4 passes results between function units. As stated above, the native design requires communication only for results of x86 instructions, and so it distributes a maximum of two results per cycle. Therefore, Xbar4 is of degree 2 in the native implementation. Because the RISC-like design must distribute the results from each function unit to all others, Xbar4 is of degree 4 in this case. This difference in size for Xbar4 is consistent with the conclusion drawn earlier from the simple two-instruction example.

Table 1 sums up the differences in communication requirements for these example two-way superscalar designs. As each connection in Table 1 represents an average of 40 wires, the RISC-like design of a two-way superscalar x86-compatible microprocessor requires roughly 1,520 more wires than a native design of comparable performance. Each of these wires is a potential logic error, speed limiter, manufacturing defect, and area penalty.

Future Directions for x86 Implementations

Current RISC-like designs suffer from significant additional communication overhead. But what about future products? To explore this question, we will extend the preceding example of a two-way superscalar x86 microprocessor to an eight-way superscalar processor.

The design of a native eight-way superscalar x86 processor is fairly straightforward. With eight instructions decoded and issued per cycle, eight autonomous integer units are used. Each integer unit has its own AC and EX blocks. The only unexpected difference from the

two-way superscalar design is that Xbar2 is omitted, since its load-balancing characteristics are less important with so many more execution resources available. In the RISC-like implementation, however, it is possible to take advantage of the lack of symmetry in the design by reducing the number of function units. It would probably be possible for a RISC-like design to achieve performance comparable to that of the eight-integer-unit native design if it had only six ACs (four for loads, two for stores) and six EXs, for a total of twelve function units.

As Table 2 shows, even in a very aggressive superscalar design, the RISC-like implementation of an x86-compatible microprocessor requires almost four times as much communication as the comparable native design. This time, the difference between the two designs is almost 15,000 wires! The trade-off in the native design is between these thousands of wires or four extra function units, each of which is an instantiation of a block which has already been designed. The choice to avoid the wires is the right one. Thus, the advantage of the native design method over the RISC-like method becomes even more pronounced as new x86 processors strive for ever-increasing levels of parallelism.

The native design style not only requires less communication than comparable RISC-like implementations, but it also offers other advantages that reduce design time and/or allow for a smaller design team. Because native designs inherently balance resource allocation, there are fewer opportunities for unforeseen bottlenecks. In general, native designs have a one-to-one correspondence between fetch, decode, issue, memory access, and execution resources. In this type of design, if an instruction can be fetched, it can be executed without regard to resource conflicts, subject only to the resolution of any true dependencies that the instruction may have.

In contrast, RISC-like implementations typically have a mismatch between peak instruction issue rate and corresponding execution resources. As a result, there are instances when instruction issue is stalled due to a lack of function units.

The native design style demonstrates yet another advantage when it is applied to superscalar microprocessors. As noted, native designs tend to be inherently balanced in performance. A side effect of this balance is that native designs also tend to be highly symmetric within each stage of the execution pipeline, resulting in reusability of function units. This property of native implementations shows its greatest benefit in the design of future, aggressively parallel x86 microprocessors. In fact, it is conceivable that the number of unique (non-replicated) transistors in future native designs will be reduced, while the total number of raw transistors increases dramatically. This improvement can reduce design times even though the resulting products increase dramatically in complexity.

	Native	RISC-like
Xbar1	2×2 = 4	2×2 = 4
Xbar2	2×2 = 4	4×4 =16
Xbar3	0	4×4 =16
Xbar4	2×2 = 4	4×4 =16
Internal	2	0
Total Connections	14	52

Table 1. A two-way superscalar RISC-like processor has more than three times the number of interconnections as in a similar native design.

VLIW Is Not the Answer

Another approach to increasing x86 performance is to utilize VLIW techniques *(see **080205.PDF**)*. The basic advantage of VLIW-based design is that it exposes more of its core to compiler optimization with the intent of increasing parallelism and hence performance. There are three approaches that can be used to incorporate varying degrees of VLIW in an x86 processor:

1. Run-time recompilation of x86 into VLIW
2. Translation of binaries from x86 to VLIW
3. Source-code recompilation

The first approach maintains compatibility with existing x86 code, but at the expense of performance. With this approach, the code window that can be used to extract parallel instructions is limited, and therefore the performance that can be achieved is also limited. The second and third approaches achieve better performance but do not maintain compatibility with existing code. Additionally, these approaches do not address the issue of memory bandwidth. Providing a steady stream of VLIW instructions to support a highly parallel execution core results in high demands on the memory subsystem.

To illustrate how instruction memory bandwidth affects performance, we define a simple model for processor performance:

$$\text{IPC} = \frac{1}{\frac{1}{\text{IR}} + \text{BP} + \text{IC}}$$

where IPC is the number of instructions executed per core clock cycle, IR is the sustained instruction issue rate, BP is the average cycle penalty per instruction for branch mispredictions, and IC is the average cycle

	Native	RISC-like
Xbar1	8×8 = 64	8×8 = 64
Xbar2	0	12×12 =144
Xbar3	0	12×12 =144
Xbar4	8×8 = 64	12×12 =144
Internal	8	0
Total Connections	136	496

Table 2. For future eight-way superscalar processors, the RISC-like method still requires more than three times the interconnect of the native design.

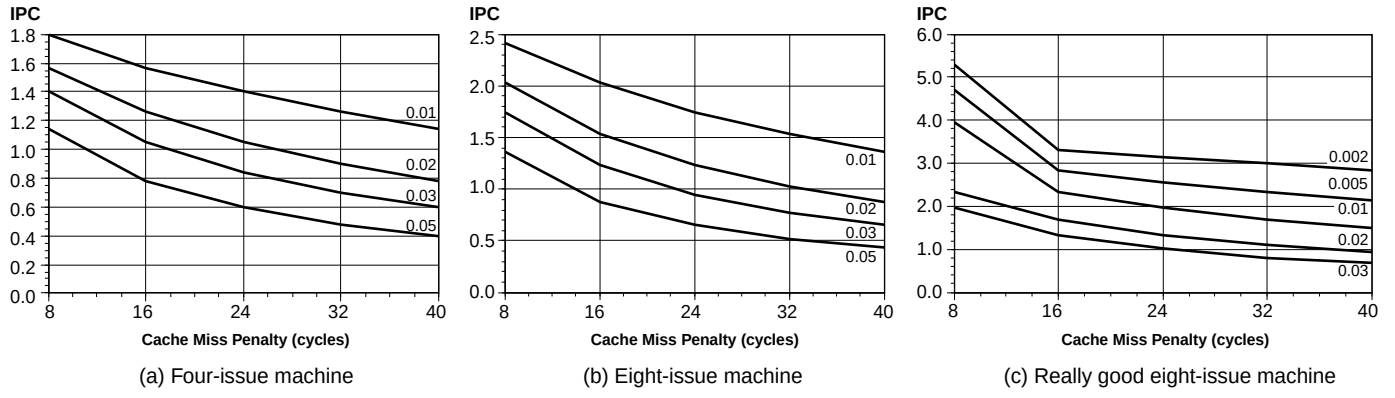

Figure 3. Estimated performance, in instructions per cycle, for hypothetical processors as a function of the the instruction cache miss penalty and various instruction cache miss rates (shown as labels on lines). (a) A four-way superscalar processor with a BTB hit rate of 95% and branch prediction accuracy of 93%. (b) An eight-way superscalar processor with similar parameters. (c) An eight-way superscalar processor with a BTB hit rate of 99% and branch prediction accuracy of 97%. (Source: Cyrix)

penalty per instruction for instruction cache misses. This equation assumes a perfect instruction execution engine. In other words, if an instruction can be fetched and issued, it will be executed without incurring any additional penalty.

BP, which represents the performance lost due to branch-prediction errors, is a function of the branch target buffer (BTB) hit rate and conditional branch prediction accuracy (BPA). IC, which represents performance lost due to instruction cache misses, is a function of the instruction cache miss rate and the latency of main memory. Figure 3 shows the performance of three different designs as a function of various instruction cache miss rates and instruction cache miss penalties.

Figure 3(a) shows the performance of a four-way superscalar processor with a BTB hit rate of 95% and a branch prediction accuracy of 93%. The penalty for a BTB miss is eight cycles, and the penalty for an incorrectly predicted conditional branch is 12 cycles. The sustained issue rate is set to 3.5, assuming that it is impossible to ever achieve a perfect instruction decoder. This design is aggressive but probably buildable in the near future. In the best case, with a very low instruction cache miss rate and a fast memory system, this design achieves an IPC of only 1.8. Therefore, the delivered performance is only about 45% of the sustained issue rate. Performance drops rapidly if either the instruction cache miss rate or the miss penalty increases significantly.

Figure 3(b) shows an eight-way superscalar design with similar parameters. The sustained issue rate is set to 7.0, optimistically assuming that an eight-instruction decoder could be as efficient as a four-instruction decoder. Even though the resources in the design have doubled, peak performance has improved by only 1.5×. This design is only about 30% efficient.

Figure 3(c) shows what would happen if it were possible to build an almost perfect microprocessor. In this case, the BTB hit rate is increased to 99%, with a miss penalty of only three cycles. The branch prediction accuracy is increased to 97%, with a penalty of only six cycles for being incorrect. The highest performance line on this graph represents an instruction cache with a miss rate of only 0.2%. At last, we have an architecture that can make efficient use of its resources: this hypothetical design achieves an IPC of more than 5.0 if its memory system is very fast. But the efficiency of the machine drops from about 65% to just 40% if the instruction cache miss penalty changes from 8 clocks to 16.

With x86 processors expected to operate at 200–300 MHz in the near future, an access to DRAM could take more than 30 processor core cycles to complete. At this point, any practical machine would be idling for long periods, regardless of the performance of its CPU core. This analysis points to the most significant problem with VLIW microprocessor design: to achieve performance improvements, the instruction fetch rate must be increased, resulting in higher demands on memory subsystem bandwidth.

Native Approach Has Good Future

Native implementations offer the best hope for turning any increases in memory bandwidth directly into increases in processor performance that are visible to a user. Native designs take advantage of x86 code density, allowing them to make efficient use of limited memory bandwidth. The symmetrical execution resources in the native approach create a well-balanced design that can consume x86 instructions as fast as they are supplied. Additionally, native designs reduce on-chip communication compared with RISC-like designs of similar performance. The reduced communication complexity and inherent symmetry can reduce the design time of next-generation products while continuing to enhance delivered performance. ♦

Mark Bluhm and Ty Garibay are chief architects of Cyrix's x86 processors, including the M1.

