DesignCon 2004

Improving Application Performance with Instruction Set Architecture Extensions to Embedded Processors

Jonah Probell, Ultra Data Corp.
jonah@ultradatacorp.com

Abstract

The addition of key new instructions and architectural state to a standard embedded processor can increase its performance by several times for target applications. Examples of processors and applications for which they are accelerated by instruction set architecture (ISA) extensions are reviewed in detail by this paper. In one case, the performance of a MIPS processor core is improved by 3x for voice-over-packet applications. In another case, the performance of an Altera Nios processor core is improved for digital video applications.

Author Biography

Jonah is the founder and principal design engineer of Ultra Data Corporation. His interest is in the architecture of high-performance microprocessors for digital signal processing and communications systems. Previously, Jonah was a processor designer for Lexra, Inc. Jonah holds a B.S. in electrical engineering from Cornell University.

Background

Since their invention the performance per cost of modern microprocessors has grown exponentially according to Moore's law [1]. The number of applications of microprocessors has also grown, though not as dramatically as their performance.

The instruction set architecture (ISA) of a microprocessor consists of the set of instructions that it can execute and its internal software accessible state elements. The instructions are the verbs and the state elements are the nouns of the machine's language.

In the early 1980s, microprocessor developers realized that by simplifying the ISA deeper pipelining could be achieved within a transistor budget feasible at the time [2]. This yielded higher clock frequencies with enough improvement to yield a net benefit despite the penalty of having to build complex operations out of multiple simple instructions. This led to the reduced instruction set computer (RISC) revolution. The use of RISC architectures involved a partitioning of responsibility between hardware and software to achieve a high performance system. Because of their simplicity and high speed, RISCs performed well on a wide range of applications and made great general-purpose processors.

Intel never modified the complex instruction set computer (CISC) ISA of its commercially successful x86 processor line to join the RISC movement. Doing so would have broken backward compatibility for a large installed base of software. Advances in process technologies gradually allowed the number of transistors per chip to grow. By the late 1990s the transistor cost of a CISC ISA, as compared to a RISC ISA was negligible for PC, workstation, and server computer processors (with large on-chip caches). With the advantage of richer instructions built into the ISA, as well as Intel's top-notch chip manufacturing capabilities, highly pipelined x86 based processors emerged as the performance leaders. The performance of x86-based processors on some applications has been further improved with the single instruction multiple data (SIMD) processing multimedia extension (MMX®), 3DNow®, streaming SIMD extension (SSE), and SSE2 instruction extensions to the already rich ISA.

As embedded computer processors have become less expensive and ever more ubiquitous in devices such as entertainment systems, networking equipment, and portable electronics RISCs have again proven themselves the architectures of choice. Embedded systems for office and consumer markets are highly cost sensitive. The simplicity of RISCs means that they can be implemented in small chips, which means high yields and low costs. Furthermore, the use of standard ISAs allows system developers to utilize readily available complete software development environments.

A general-purpose processor ISA does not offer enough performance to meet the requirements of some applications. This is often the case for multimedia, cryptography, and network communications applications. The requirements can often be met by standard RISCs if they are customized with ISA extensions targeted at the processing needs unique to the application. Many of today's embedded RISC processor vendors, such as ARM®, IBM®/PowerPC®, MIPS Technologies®, Tensilica®, ARC International®, and Altera®/Nios®, provide interfaces on their processors to enable the integration of such ISA extensions. Furthermore, such ISA extensions are strictly additive and implemented in such a way that the new hardware is compatible with existing software written or compiled for the standard ISA.

As the cost of transistors decreases and their numbers on chips grow, the standard processor consumes an ever-smaller portion of the chip. The cost in chip size of adding ISA extensions is similarly

decreasing. Due to the reduced cost for the competitive advantage of a standard ISA processor with improved performance from ISA extensions, it will be increasingly common in the future to customize standard ISAs with extensions to enhance performance for target applications.

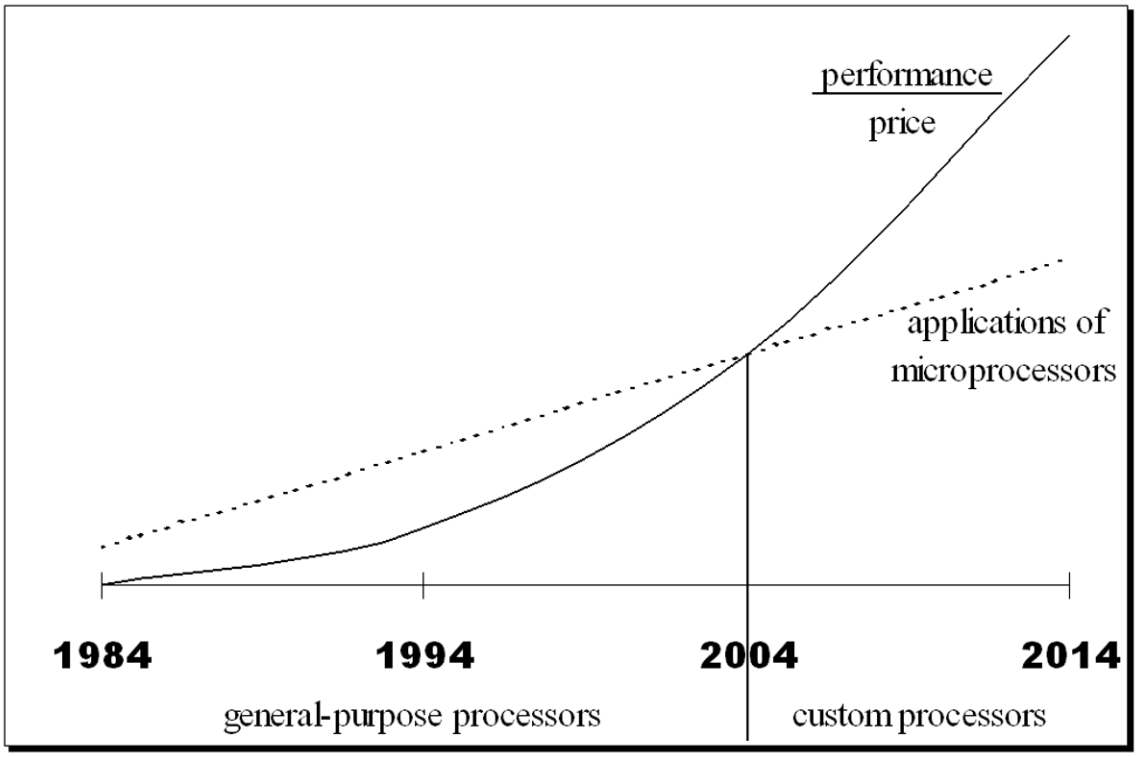

Figure 1: A qualitative representation of the assertion that future costs will make it worthwhile to customize processors for their target applications.

All microprocessors have a system bus interface, which provides access to memory and memory-mapped devices. The capabilities of a system can be extended by the addition of system bus devices. Such devices fall outside of the scope of the ISA. The focus of this paper is strictly on ISA extensions.

Creating an ISA Extension

The process of implementing and using an ISA extension to maximal benefit requires the collaboration of hardware and software design. Software design identifies segments of code that could be replaced or loop iterations that could be reduced if recoded utilizing architectural extensions. Hardware design models the new extensions and determines their cost to die size and their cost, if any, to overall processor clock speed. Together system designers trade off die size and clock speed cost with architectural performance benefits. The new instructions and the software that uses them are revised iteratively.

Amdahl's law states that the amount of performance enhancement to a complete application program equals the amount of performance enhancement to a section of the program times the fraction of the program execution time that that section represents [3]. To gain the greatest possible application performance enhancement within an allowable design schedule, the most executed code segments should be inspected first for performance enhancements. A code-profiling tool can be used to guide

software design to the most executed code segment. These are, typically, the innermost loops within the nested loop structure of application software.

For high performance or low power applications software design often writes optimized critical inner loop code in assembly language or in a high-level language (HLL) with inline assembly or compiler intrinsics. This is critical for achieving the benefits of ISA extensions because HLL compilers are not capable of utilizing the new instructions and processor state. Furthermore, it is difficult to add the ability for a compiler to identify constructs written in an HLL that could be mapped to assembly code using ISA extensions. Since ISA extensions are designed to target specific pieces of critical code they would rarely apply to other compiled code, and so compiler support for ISA extensions is not worthwhile.

After identifying a section of critical assembly code several techniques are applied to identify how best to employ ISA extensions. The code segment's constituent data processing, data movement, and program flow control instructions are discriminated. Code is then rearranged in order to group the data processing instructions, even when this means placing no-ops in delay slots. Operations on unpacked data are identified. That includes data stored in unpacked structures in memory as well as data that does not utilize all of the bits within the processor's registers. Operations that can incur overflow or special conditions are identified. These might include but are not limited to conditions that are signaled by flags or cause exceptions in some ISAs. Finally, operations that incur stall cycles because they are waiting for the availability or the result of a processing unit are identified. A good instruction set simulator (ISS) for the processor can be helpful in finding such conditions. After identifying the performance limitations of the software, ISA extensions can be identified that will overcome them.

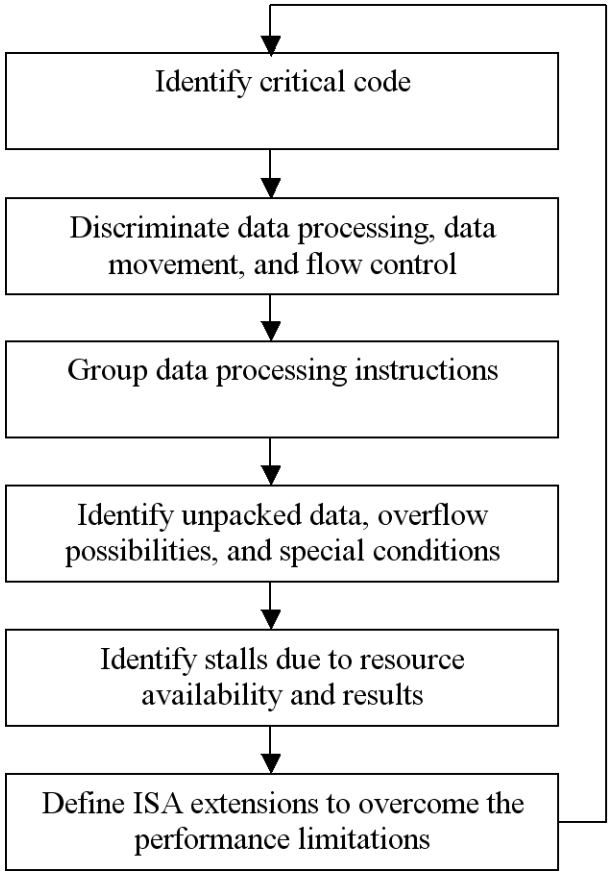

Figure 2: Steps for defining an appropriate ISA extension.

Methods of Acceleration

Some techniques of ISA extension with high applicability across a range of applications include SIMD, hardware saturation, guard bits, multiple accumulators (for loop unrolling), and operations, such as bit-field manipulation, on packed data.

SIMD is a powerful performance-enhancement because it is a form of processing-element parallelism, which allows multiple data samples to be processed simultaneously. It improves performance most when applied to applications that operate on data with a resolution that is an integer fraction of the processor's natural data path width. For example, SIMD on 32-bit processors accelerates multimedia-processing applications such as video, which typically operates on 8-bit input data, and audio, which typically operates on 16-bit input data. As a result, the industry standard instruction sets for consumer devices have each had SIMD extensions added. MMX, 3DNow, SSE, and SSE2 extend the x86 ISA, AltiVec® extends PowerPC, the ARM SIMD media extensions extend ARM, and MDMX extends the MIPS ISA.

A 255-point FIR filter of 16-bit data, such as the one written in C in figure 3, implemented in an industry standard 32-bit processor is accelerated by a SIMD ISA extension.

```
short samp[255];
short coef[255];
long  res;
for(i = 0; i < 255; i++) {
   res += coef[i] * samp[i];
}
```

Figure 3: C code for a typical 255-point FIR filter.

The filter algorithm, compiled or written in assembly language for a standard RISC, looks like the listing in figure 4. This implementation requires 255 loop iterations and only uses 16 of 32 bits in the load/store, general-purpose register (GPR) and multiplier data paths.

```
li   r3, 255        /* r3 is the loop counter */
loop:
   lh   r4, 0(r1)   /* r1 is samp ptr */
   lh   r5, 0(r2)   /* r2 is coef ptr */
   subi r3, r3, 1
   mad      r4, r5  /* multiply-add to accumulator */
   addi r1, r1, 2
   bnez r3, loop
   addi r2, r2, 2
mflo r3             /* r3 gets accumulated filter result */
```

Figure 4: MIPS32® compiled assembly code for a 255-point FIR filter.

A SIMD multiply-add instruction is designed. It multiplies the upper 16 bits of each 32-bit register operand, multiplies the lower 16 bits of each 32-bit register operand, and adds those two products to an accumulator register. A supporting instruction, mfa, is needed to move the data from the accumulator to a GPR. The resulting filter loop is shown in figure 5. Full words, each containing two half-words, are loaded from memory. The filter loop requires the same number of instructions, but completes in 127 rather than 255 iterations.

```
li   r3, 127          /* r3 is the loop counter */
loop:
   lw   r4, 0(r1)     /* r1 is samp ptr */
   lw   r5, 0(r2)     /* r2 is coef ptr */
   subi r3, r3, 1
   simd mad r4, r5    /* multiply-add to accumulator */
   addi r1, r1, 4
   bnez r3, loop
   addi r2, r2, 4
mfa  r3               /* r3 gets accumulated filter result */
lh   r4, 0(r1)
lh   r5, 0(r2)
mult r4, r5           /* multiply last odd filter terms */
mflo r2
add  r3, r3, r2
```

Figure 5: 255-point FIR filter in MIPS32 assembly code, modified to utilize a SIMD multiply-add instruction.

Notice the extra instructions needed at the end of the loop. These are needed to complete the last multiply-add of this filter with an odd number of terms. Because of such overhead, SIMD is economical only for loops with a relatively large number of iterations. Furthermore, this example assumes that the filter data and coefficients both start at a word aligned address. If that condition were not guaranteed, even more code overhead would be necessary.

Other ISA extensions such as hardware post-modified pointers and hardware loop counters significantly improve the critical loop performance of algorithms such as FIR filters. The addition of such extensions is not supported by the ISA extension interfaces of major RISC processor cores.

Some standard RISC cores include a 3-stage pipelined multiplier in order to achieve the clock frequency goal. The multiplier stores its result in a special accumulator register. RISC architectures often include multiply-accumulate instructions that mask the multiplier latency by allowing multiply-accumulate operations to be issued to the result register in series without stalls. For algorithms that require multiply operations without the need to accumulate results, the multiplier latency must be masked with other useful instructions. If this is not possible, for example because a series of multiplies is required, software will incur stalls in a standard RISC implementation as shown in figure 6. This example requires 12 cycles to perform 3 multiplies.

```
mult  r1, r2
mfa   r3     /* 2-cycle stall */
mult  r4, r5
mfa   r6     /* 2-cycle stall */
mult  r7, r8
mfa   r9     /* 2-cycle stall */
```

Figure 6: A series of multiplies with 32-bit results in generic assembly code using a standard 3-stage pipelined multiplier.

An ISA extension for such software would include as many result registers as multiplier pipeline stages. Multiplies can be performed in series to different target registers. This fills the delay between a multiply and the reading of its result with other useful multiplies and result register reads. A series of multiplications, coded as shown in figure 7, can be performed in 6 cycles without any stalls. Processing time for such an algorithm is halved by the use of multiple accumulators.

```
mult0 r1, r2
mult1 r4, r5
mult2 r7, r8
mfa0  r3
mfa1  r6
mfa2  r9
```

Figure 7: A series of multiplies with 32-bit results in generic assembly code using an ISA extension with multiple result registers.

Some applications represent data as fractional signed fixed-point numbers in which the most significant bit is a –1 bit, and the remaining bits, to the right of the radix (binary) point represent diminishing positive fractional powers of 2. Fractional representation has the benefit of (almost) completely eliminating the chance for overflow in multiplication. The product of two signed fractional numbers has two copies of the sign in the two most significant bits, as shown in figure 8.

```
signed       signed
decimal      fractional binary

 -0.5         1.100
× 0.75       ×0.110
 -0.375      11.101000
```

Figure 8: A fractional signed fixed-point binary multiplication.

The extraneous sign bits must be eliminated with a left shift by 1 bit position. The shift operation, needed to accommodate signed fractional multiplication, is not present in the fixed-point multiplication unit of standard RISC processors. Instead, each product read from the multiplier result register would require a left shift instruction. If data is to be used from both the high and low halves of the product, then the bit shifted left from the MSB of the low half must be shifted into the LSB of the high half of the product. A multiplier in an ISA extension that operates in fractional arithmetic mode can eliminate the instructions needed for the shift operations, enhancing software performance.

A series of multiply-add operations, storing results in an accumulator register, is common to many applications. Sometimes the series of multiply-adds can generate results that will overflow or underflow the accumulator register under the condition of worst-case data values. Standard RISC processors do not flag or trap accumulator overflow or underflow. To check for and handle this condition in a standard RISC core requires many conditional branches. Depending on the data and what processing is done on it, an overflow or underflow condition may need to be checked as often as once for each multiply-add instruction. An ISA extension that appends software-visible guard bits to the most significant end of the accumulator to store the most-significant bits of overflow or underflow can maintain the fidelity of calculations without the heavy cost of checking for overflow or underflow in software for each multiply-add operation. Naturally, this requires the programmer to have an understanding of the filter size and the coefficients and data.

Other lower-fidelity applications can accept saturation or clipping of the accumulator value. This requires storing the highest value that can be represented in the case of overflow or the lowest value that can be represented in the case of underflow. Wrap-around to a low number in the case of overflow or a high number in the case of underflow is unacceptable. To perform saturation in a standard RISC requires the software complexity and performance penalty of overflow or underflow detection. Adding hardware support for saturation in the data path of an ISA extension is straightforward, and completely eliminates the need to saturate in software. Software saturation for a MIPS™ processor is shown in figure 9. The MIPS architecture does not have an overflow flag. Signed add and subtract operations can trap on overflow, but an exception is very costly.

```
    addu t0, t1, t2 /* compute sum */
    xor  t3, t1, t2 /* if operands have opposite signs */
    bltz t3, no_ovf /* then overflow not possible */
    /* operands have same sign */
    xor  t3, t0, t1 /* if sum does not have same sign */
    bltz t3, do_sat /* then add overflowed */
    nop             /* nop or some other instruction here */
no_ovf:
    ret

do_sat:
    /* we know overflow has occurred, and we know both operands
       have same sign, so we look at the sign of an operand, if
       its positive we limit (saturate) to positive full scale,
       if negative we saturate negative full-scale */
    li   v0, 0x7fffffff /* positive full scale */
    li   t8, 0x80000000 /* negative full scale */
    slt  $1, $0, t1     /* check sign of operand */
    movn v0, t8, $1     /* saturate */
    ret
```

Figure 9: MIPS assembly code to perform saturation in software

SIMD operations, multiple accumulators, fractional multiplication, guard bits, and hardware saturation are ISA extension techniques that apply to a wide range of applications. Each software application performs unique operations that can be accelerated with unique ISA extension hardware. MIPS Technologies has demonstrated a critical AES encryption/decryption table lookup and rotate routine [4]. With minimal additional hardware the number of cycles for the routine is reduced from 24 to 6 with the addition of a single instruction. MIPS Technologies has also demonstrated a 3x performance increase for an entire voice over IP application with the addition of 19 instructions and 259 architectural state bits [5].

In digital video processing, pixel data for 4x4 blocks can be stored in a packed format in memory with a resolution of 1 byte per pixel. Standard video coding formats employ motion estimation, which is one of the most processor-time consuming procedures in the video encoding process. Motion estimation requires a search of neighboring blocks to find the closest match. The procedure for such a search requires computing a sum of absolute differences between the block being coded and its neighbors. Truncated code to compute a 4x4 sum of absolute differences in a standard RISC CPU is shown in figure 10.

```
/* compute abs diff between pixel [0,0] of each block */
lb
lb
lb
sub

/* compute abs diff between pixel [0,1] of each block */
/* add to running sum */
lb
lb
sub
abs
add

/* compute abs diff between pixel [0,2] of each block */
/* add to running sum */
lb
lb
sub
abs
add

:

/* compute abs diff between pixel [3,3] of each block */
/* add to running sum */
lb
lb
sub
abs
add
```

Figure 10: Assembly code operations, truncated for brevity, to compute the sum of absolute differences for a 4x4 block of video data.

The 4x4 block SAD requires 79 instructions. To improve this, we design an instruction, SAD, which stores in a 24-bit accumulator register the sum of absolute differences between two 32-bit registers of 4 packed 8-bit values in parallel, and a similar instruction, SADA, that adds to the accumulator. The full 4x4 block SAD procedure, using the 2 additional instructions and 24 additional state bits is shown in figure 11.

```
lw
lw
sad

lw
lw
sada

lw
lw
sada

lw
lw
sada
```

Figure 11: Sum of absolute differences assembly operations using SAD and SADA ISA extensions.

This procedure requires only 12 instructions, which is more than a 6x speedup on SAD calculations for motion estimation in a video encoder.

Another simple extension to many standard RISC ISAs is the creation of “insert” and “extract” bit-field instructions. These can effectively save many constituent shifts, ands, and ors in order to read from or build data elements with unaligned bit-fields. Figure 12 is a table summarizing common ISA extensions and their benefits.

ISA Extension	**Benefit**
SIMD	Reduce critical loop iterations
Multiple accumulators	Eliminates multiplier result stalls in series of multiplies
Fractional multiplication	Eliminates accumulator read, shift, write operations
Guard bits	Avoids costly overflow and underflow detection
Hardware saturation	Eliminates costly overflow and underflow detection
Bit-field extract/insert	Shortens series of shift and logical instructions
Special operations (e.g. SAD)	Eliminates series of data processing instructions

Figure 12: Summary of ISA extensions and their benefits.

CPU Interfaces for ISA Extensions

ISA extensions come in many flavors, distinguished by ill-defined terminology such as “coprocessor”, “instruction extension”, “custom instruction”, and “user defined instruction”. The simplest and longest-used type of ISA extension is a generic coprocessor interface. Coprocessor interfaces are nearly universal in commercial general-purpose processors, and are defined within the x86, PowerPC, MIPS, and ARM architectures among others. Generic coprocessors are most effective for long operations because the overhead of the data movement is amortized over more cycles. Generic coprocessors are also most effective on small pieces of data because the interface is limited to a single data word read or write in a cycle. The most widely known uses of coprocessors are for implementing floating-point units and graphics processing engines. The interface signals of a typical generic coprocessor interface are shown in figure 13.

Initiate			
Input	RD_GEN	1	Read a general register
Input	RD_CTL	1	Read a control register
Input	WR_GEN	1	Write a general register
Input	WR_CTL	1	Write a control register
Data in			
Input	WR_ADDR	5	Register write address
Input	WR_DATA	32	Register write data from CPU to coprocessor
Data out			
Input	RD_ADDR	5	Register read address
Output	RD_DATA	32	Register read data from coprocessor to CPU

Figure 13: Generic coprocessor interface.

A generic coprocessor ISA extension only has the ability to read or write a single word of data in a cycle. This sets a relatively low limit on bandwidth in and out of the ISA extension. The generic coprocessor can only be accessed by "move to" and "move from" instructions defined as part of the RISC ISA. Different functions can be invoked in the generic ISA by using different register write addresses, but a generic coprocessor is less flexible than other types of ISA extensions. Altera has addressed these limitations in its Nios RISC soft-core [6]. The Nios custom instruction interface is shown in figure 14.

Initiate			
Input	START	1	Begin an operation
Input	PREFIX	11	The Nios K register
Data in			
Input	DATAA	32	Data input A
Input	DATAB	32	Data input B
Data out			
Output	RESULT	32	Return data

Figure 14: Altera Nios custom instruction interface.

Nios custom instructions accept data inputs from two source GPRs and can simultaneous return data to a destination GPR, just like the ALU in general-purpose RISC processor data paths. This gives Nios custom instructions greater processing bandwidth and lower latency than a generic coprocessor. Custom instructions can be performed each cycle, just as if they were part of the ALU.

There are 5 Nios custom instruction ports, but each custom instruction can perform numerous functions as selected by the processor's K register. The ability to perform extra functions comes at the cost of an extra prefix cycle to set the K register value.

Nios also supports multi-cycle instructions. Once a multi-cycle instruction is executed the compiler can be instructed to not call that custom instruction again until the multi-cycle completion time has passed. Assembly code must also respect the completion time restriction. It is nearly impossible to design a multi-cycle Nios custom instruction with data dependent variable processing time because if an instruction completes before the maximum time elapses the CPU has no way to tell when the completion has occurred. Similarly, it is also difficult to use pipelining in custom instruction logic, such as in a pipelined multiply-accumulate unit, because the instruction to move data from the accumulator to a GPR will not automatically stall the processor if called while the last multiply is still in progress. This violates the strict ordering of machine instructions and relies on the assembly code writer to invoke the move instruction with an awareness of the pipeline timing.

The coprocessor interface of the ARM 9 processor, shown in figure 15, addresses this limitation with handshaking signals [7]. The ARM 9 coprocessor interface is more advanced than a typical generic coprocessor interface. ARM 9 coprocessors have the IKILL, CHSD, and CHSE handshaking signals, which tie the coprocessor more tightly to the CPU pipeline. The CHSD and CHSE signals can stall the processor if necessary, such as when executing an accumulator move instruction while a multiply-accumulate is still moving through the multiplier pipeline. The IKILL signal allows the processor to cause the coprocessor to kill an instruction in progress. This can improve exception response time. In the Nios custom instruction interface the timing of each signal is favorable for the ISA extension designer. In an ISA extension interface with handshaking, the ISA extension designer must design with a keen awareness of the cycle timing budget in order to avoid significantly degrading the processor's maximum clock frequency.

Initiate			
Input	PADV	1	Instruction fetch has been requested
Input	INSTR	32	Instruction word
Data in			
Input	CPDIN	32	Coprocessor data in
Handshaking			
Input	IKILL	1	Disregard instruction requested in previous cycle
Output	CHSD	2	Handshake used when new instruction entering execute stage
Output	CHSE	2	Handshake used when coprocessor instruction requires another execute cycle
Data out			
Output	CPDOUT	32	Coprocessor output data

Figure 15: ARM coprocessor interface.

Like generic coprocessor interfaces, ARM coprocessors are limited to a single input data word. This gives ARM coprocessors a lower effective processing bandwidth than other ISA extension interfaces. The ARM 9 coprocessors interface gives designers the benefit of the ability to decode instructions. As a result, ARM 9 coprocessors can perform a wide range of operations and variants without a multi-cycle procedure to call the variant.

The MIPS ISA defines multiple coprocessor interfaces [8]. They allow a single unidirectional word of data movement each cycle. The MIPS ISA's coprocessor interface allows instruction decoding and has handshaking signals. This is also true for the MIPS Technologies user defined instruction (UDI) interface [4]. MIPS Technologies UDI interface signals are shown in figure 16.

Initiate			
Input	IR_E	32	Instruction word
Input	IRVALID_E	1	Valid indication for IR_E
Input	START_E	1	Pipeline control logic run signal
Output	RI_E	1	Reserved (unrecognized) instruction
Data in			
Input	RS_E	32	RS register source operand
Input	RT_E	32	RT register source operand
Handshaking			
Output	STALL_M	1	Multi-cycle stall indication to the CPU
Input	KILL_M	1	Kill instruction in progress (i.e. due to an exception)
Input	RUN_M	1	Run signal used to qualify KILL_M
Data out			
Output	WRREG_E	5	Destination GPR address
Output	RD_M	32	32-bit result data

Figure 16: MIPS Technologies user defined instruction interface.

The MIPS Technologies UDI interface allows 2 GPR reads and 1 write in each cycle. It also allows ISA extension decoding of the instruction. The interface has handshaking in both directions between the CPU and the UDI. The MIPS Technologies UDI interface is unique in giving the UDI the ability to specify its destination GPR with the WRREG_E signal.

A tight coupling of an ISA extension to the processor pipeline maximizes the ability for the cycle-time budget aware designer to enhance processor performance. The MIPS Technologies UDI interface is more closely tied to the CPU pipeline than any other major commercial ISA extension interface, with the possible exception of the ARC and Tensilica instruction extension interfaces. Tensilica instruction extensions are specified in a proprietary language and compiled into the processor with Tensilica's processor configuration tool. Neither ARC's nor Tensilica's instruction extension interface signals are publicly known.

The Value of ISA Extensions

For audio-visual and scientific signal processing, digital signal processors (DSPs) are popular. DSPs are microprocessors tailored for a wide range of real-time signal processing applications. Most modern DSPs employ the techniques of SIMD operations, multiple accumulators, fractional multiplication, guard bits, and hardware saturation, among others to achieve remarkable performance on their target applications.

The use of rich tools eases development and improves software quality. General-purpose processors have the advantage of support from industry-standard third-party software tools including operating systems, application software, code libraries, optimizing compilers, debuggers, and other software development tools. For this reason, many chip designs require a general-purpose processor. Some system designers choose to include both a DSP and a RISC microprocessor. This incurs software design complexity in the need to handle interaction between the two processors and the need to use two different software development environments. Using separate cores adds hardware cost due to increased die area and power consumption. Even if the application fully utilizes both processor cores, there is an inefficiency of transistors in the duplication of caches and system bus interfaces. Applying ISA extensions to a general-purpose processor is more efficient because it allows the designer to trade area for performance as needed, and do so within a single software development environment.

For applications with a fixed real-time performance requirement, the performance benefit of ISA extensions can also be used to reduce energy consumption. The increased performance realized by an ISA extension allows the microprocessor to be run at a lower clock speed, which saves energy. Decreasing the clock speed requirement also allows the processor logic design to be synthesized with a more relaxed timing constraint enabling the EDA tools to reduce the gate count, which often decreases power consumption. Also the chip could be manufactured in a low power process technology where supply and threshold voltages are tuned to minimize switching and leakage power consumption.

The performance benefits of ISA extensions can also permit lower manufacturing cost. With the ISA improvement, performance requirements can be met in a larger geometry process technology with lower mask set and wafer production costs. ISA extensions also have the benefit of improving efficiency as measured in work per executed instruction. This means that fewer instructions are required for the critical code and therefore a smaller instruction cache is required. Many ISA extensions, such as those that operate on packed SIMD data, also improve data storage efficiency maximizing the amount of data locality (in the register file and data cache) and decreasing the data cache size requirement. Since on-chip memories are the largest subsystem consumers of die area in typical systems-on-chip, the addition of ISA extension logic to a chip can reduce the total die area. Furthermore, with a reduced clock speed requirement, EDA tools can make compromises to reduce the synthesized gate count. These die size decreases improve yield and allow more dice to be manufactured on a wafer, both of which reduce manufacturing costs and improve profit margins.

Citations

[1] Gordon E. Moore, “Cramming more components onto integrated circuits”, Electronics, Volume 38, Number 8, April 19, 1965
[2] David A. Patterson, “Reduced Instruction Set Computers”, Communications of the ACM, Volume 28, Number 1, January, 1985
[3] John L Hennessy & David A Patterson, “Computer Architecture A Quantitative Approach”, Second Edition, 1996, page 29
[4] Mike Thompson, “Accelerating Application Performance with MIPS Technologies’ Pro Series ™ Cores”, http://seminar2.techonline.com/~mips22/may1303/, May 13, 2003
[5] MIPS Technologies, Inc., “Accelerating VoIP with CorExtend™ in MIPS32™ Pro Series Cores”, document number MD00302, revision 1.0, January 24, 2003
[6] Altera Corporation, “Application Note 188: Custom Instructions for the Nios Embedded Processor”, version 1.2, September, 2002
[7] ARM Holdings PLC, “ARM9EJ-S Technical Reference Manual”, revision r1p2, September 30, 2002
[8] MIPS Technologies, “Core Coprocessor Interface Specification”, revision 1.14, March 22, 2002