Interesting Characteristics of Barcelona Floating Point Execution *

Ben Bales and Richard Barrett
Oak Ridge National Laboratory
Oak Ridge, TN 37931

Presented at Cray User Group, Atlanta, Georgia, USA on May 7, 2009

In almost all modern scientific applications, developers achieve the greatest performance gains by tuning algorithms, communication systems, and memory access patterns, while leaving low level instruction optimizations to the compiler. Given the increasingly varied and complicated x86 architectures, the value of these optimizations is unclear, and, due to time and complexity constraints, it is difficult for many programmers to experiment with them. In this report we explore the potential gains of these "last mile" optimization efforts on an AMD Barcelona processor, providing readers with relevant information so that they can decide whether investment in the presented optimizations is worthwhile.

1 Introduction

Commodity x86 processors have made huge inroads into the supercomputing market. As many programmers look for easy to program, general purpose processors, the x86 lineup has become increasingly attractive. While programming on alternate architectures such as GPUs and in-order vector processors, knowledge of execution mechanisms and instruction timing is very important. However, this same information is often glossed over or ignored in x86 architectures. In this paper, we attempt to fill in the cracks to give programmers some idea of the characteristics of floating point execution in an AMD Barcelona core.

For good reasons, developers spend much of their time tuning algorithms, communication systems, and memory access patterns. Tuning algorithms is a very natural thing, because if there is less to be done for a program, that program can generally be executed in less time as well. Adjustment of communication systems and memory access patterns are the first step in fitting purely theoretical execution models to the real hardware platforms that will actually execute the code. However, beyond this point, most projects never venture. Low level assembly optimizations have always been leveraged in BLAS libraries, but they are not used much past that. In this light, we try to answer whether or not these optimizations will benefit regular, less generic code.

*This research used resources of the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

The paper is organized as a presentation of the relevant hardware features of the Barcelona core, followed by a number of small benchmarks meant to test these features, and concluded with two pseudo real-world benchmarks meant to test these features in a more practical environment.

2 The Barcelona FPU

The AMD Barcelona is an out of order, superscalar processor complete with SIMD instruction extensions and a three level cache hierarchy. Generally speaking, this is as far into the architectural description that anyone needs to go to write decent code. However, a slightly more detailed model will be useful in explaining and examining the results of the following tests.

The basic execution model of the FPU in the Barcelona looks something like Figure 1.

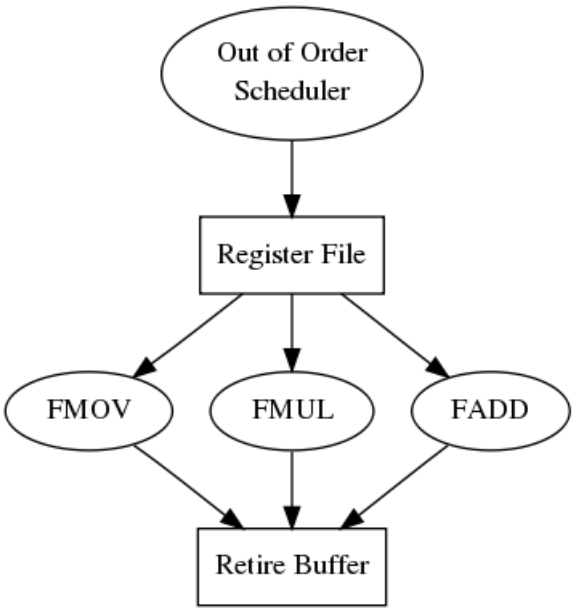


Figure 1: Overview of the Barcelona FPU

The renamed register file is the set of registers from which the processing pipelines operate. As instructions are read, the software accessible registers referenced by the instructions are remapped in different locations to allow a pipeline free access to the original register contents. The software visible registers xmm0-xmm15 are not real, but are allocated within the register file which the internal processing mechanisms operates from. On each clock cycle, instructions in the scheduler check to see if their dependencies have been filled and, if they have, jump into one of the FMOV, FADD, or FMUL processing pipelines. Only one instruction can jump into each pipeline at a time, but each pipeline can accept one instruction per cycle. This constitutes the superscalar nature of the Barcelona processor: more than one instruction can execute per cycle. Naturally (as the names indicate), the

FMOV pipeline takes move instructions, the FADD pipeline takes add instructions and the FMUL pipeline takes multiplication instructions. In all actuality, most moves and logical instructions can be performed in more than one pipeline, while pure adds and multiplies are restricted to their special pipelines. The SIMD nature of the processor lies one level deeper, as each operation in the FMUL or FADD pipeline can account for up to two double or four single precision operations. Most of the commonly used instructions have four cycles latencies, while simple register to register moves have two cycle latencies. Vector operations that create dependencies between data in the same register are emulated through shuffles and SIMD instructions.

After instructions finish running through the execution pipelines, they land in the retirement buffer (which is actually part of the renamed register file). Though values headed for the retirement buffer can be fed back into dependent instructions, until they are officially retired out of this buffer and back into the regular register file, they are not guaranteed to have an effect on software execution. A branch mis-prediction, for instance, invalidates the speculative values in this buffer. Instructions retire out of this buffer in-order, and there are no superscalar limitations implied at retirement (three multiplies or three adds may retire together).

3 Instruction Ordering

We examine out-of-order scheduling and vectorization strategies here.

3.1 Out-of-Order Limits

As previsouly discussed, achieving peak performance on a Barcelona core requires both a vector add and multiply execute in each cycle. Traditional in-order processors require instructions be shuffled in this manner, but in theory, the out-of-order scheduler on the Barcelona should work just as fast with code organized in blocks of adds followed by blocks of multiplies. We constructed a computation consisting of SSE multiplications followed by the same number of SSE additions (blocked), and compared it to the same computation with the additions interlaced one to one into the multiplications (shuffled). Varying the sizes of the add/multiply blocks exposes the processor's ability to schedule look ahead so that both FMUL and FADD pipelines are full, as illustrated in Figure 2.

The AMD manual claims the scheduler can look up to 72 macro-ops (a type of internal instruction) into the future. In these tests, performance died off around blocks of fifty instructions, indicating that the scheduler was no longer able to see far enough into the future to effectively fill the SSE pipelines.

These tests required no data movement, a situation rarely encountered in practice. For a more practical comparison, we measured the performance of two parallel dot product kernels, one with blocked instructions and the other with shuffled ones (which both make quite heavy use of the memory subsystem).

These tests revealed a performance difference between the two approaches. The shuffled code achieved 16.0 GFLOPS while the blocked code performed at around 15.1 GFLOPS, a difference of about five percent. Because data movement is present, it is difficult to

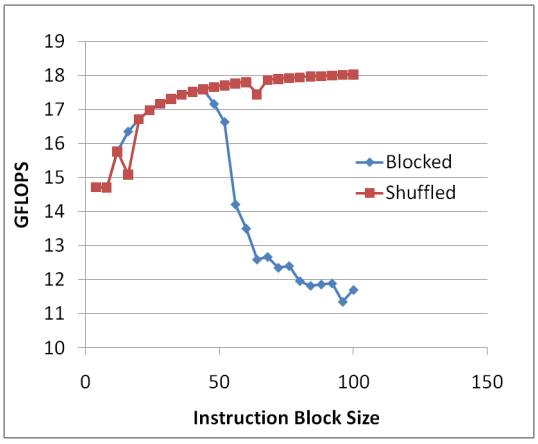


Figure 2: Ordered vs. shuffled code execution

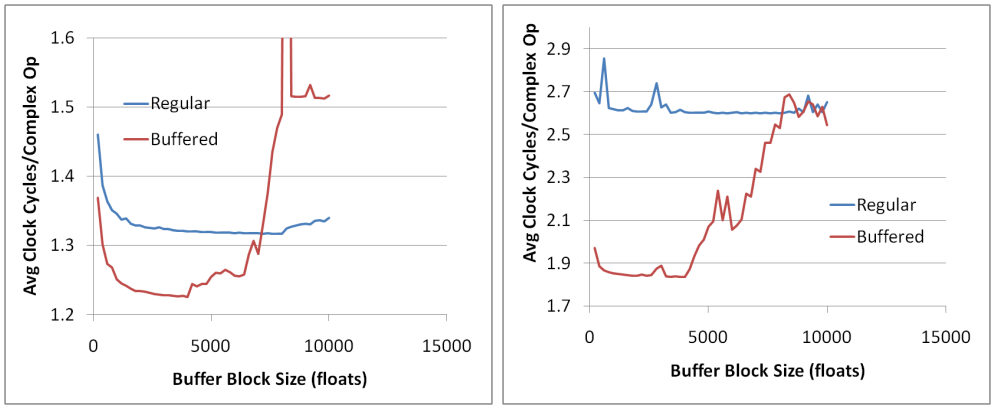
attribute this difference purely to the scheduler. However, as real code has memory moves, this difference is probably more representative than that of the benchmarks in Figure 2.

3.2 Vectors

The SSE instruction set is built in the spirit of Single Instruction Multiple Data (SIMD) operation. Multiple values can be held in one register, but they are only together so they may be referenced at one time, not so that they assist each other in any way. Prior to the specification of SSE3, the instruction set did not link elements in the same register by anything other than move and shuffle operations. However, many simple computations such as dot products and complex multiplication create dependencies between neighboring elements in a register. The associated realtime re-ordering and shuffling (which can become expensive) has made buffering of a set of this shuffled data quite popular. Unfortunately, it is very difficult for compilers to recognize and effectively implement these temporary buffers, and manual implementation can be very costly both in terms of code quality and maintainability.

We configured two tests to examine the effectiveness of these extraneous shuffle and move instructions by comparing a simple complex dot product code with one that takes advantage of memory reordering and buffering. The first uses an SSE2 single precision complex dot product found in the AMD 10h optimization guide[1]. The second uses the same basic code structure, but incorporates buffers to store reordered copies of the original data to simplify the on-the-fly shuffling. The difference amounts to trading a shuffle and four register moves for two cache loads. We executed the tests for different reorder buffer sizes.

As seen in Figure 3(a), for small buffer sizes the code gets up to a seven percent performance boost. However, as the buffer sizes grow and more information gets reordered in temporary buffers, the on-the-fly code works better. The broken L1 cache causes the buffered code to run significantly slower than the original. (This is the point where the shuffle instruction and four register moves are cheaper than the two cache loads.)



(a) Barcelona (b) Athlon

Figure 3: Buffered vs. on-the-fly complex multiplication

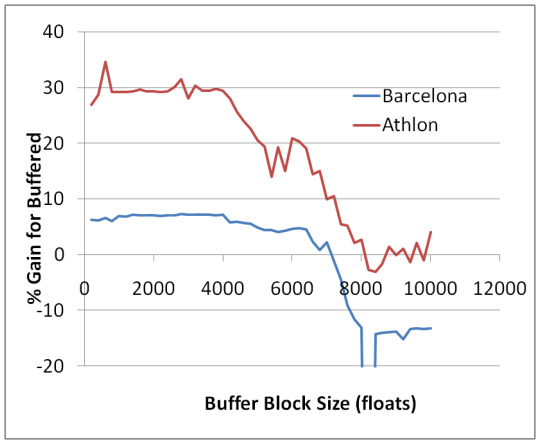


Figure 4: Athlon and Barcelona: buffering

As illustrated in Figures 3(b) and 4, this optimization appears to have been more relevant on the older Athlon 64 style processors. While this buffering would almost guarantee a speedup on the Athlon, improved shuffle instruction throughput on the Barcelona makes consideration of these instruction-trade optimizations something only very high end code would need to worry about.

3.3 Ordering Conclusions

We see two conclusions from these tests. First, manual instruction ordering (at least of adds and multiplies) will probably not result in imrproved performance. Strong performance requires balancing multiplies and adds, unrolling enough to allow the pipelines to stay busy, and keeping instruction blocks within about a 50 instruction look ahead range (this really depends on the number of macro-ops in an instruction).

Second, re-ordering and buffering data can provide some performance gain, though it is hard to quantify this gain in any general sense. In the case of the complex multiplication on a Barcelona processor, the seven percent speedup is probably not sufficient motivation to implement these optimizations. However, the changes would have been worth considering on the older Athlon 64 style Opterons. For the most part, the extra shuffle and move instructions associated with non-reordered code will not be the difference between success and failure in tuning an application, but the difference does exist. Unfortunately, these code changes may be awkward to implement and can easily lead to additional problems in the cache heirarchy.

Overall, the Barcelona provides a nice out-of-order execution environment. If a program feeds the processor within the limits of the look-ahead buffer, the processor will probably run it efficiently.

4 Branch Predictor

Advanced branch prediction is one of the strong selling points of new x86 processors, allowing a core to guess how to branch while only requiring corrective action when the respective conditional is fully evaluated. Most code uses only the simplest features of the predictor, or is otherwise memory bound such that the cores have an excess of time to work on conditionals. However, understanding the branch predictor may improve performance for compute-bound code with sufficiently patterned conditionals. The following is a description of the Barcelona branch predictor largely sourced from information compiled on the Athlon 64 branch predictor by Hans de Vries[2].

Static branches are not a real issue on the Barcelona processor, and they are detected and predicted separately from the dynamic branches. To handle dynamic branches, each core maintains a table of Global History Bimodal Counters (GHBC) holding two-bit unsigned integers which are associated with dynamic branches in code. When the core encounters a branch and the respective counter is either zero or one, it does not take the branch, and when the counter is two or three it does[1]. When the conditional is finally evaluated and the core knows which way the conditional went, it either increments the counter (branch) or decrements it (no branch). The counter does not overflow from three to zero or vice versa. However, were the branch predictor heuristic this simple, even alternating between branching and not branching on a single conditional would baffle the core. To account for patterns, the last few bits of the global branch history are used in combination with bits from the instruction address as the address then used to select the appropriate GHBC entry (hence "Global History" in the name). This means if a core encounters a conditional after branching, it will use a different GHBC entry than if it had not previously branched but still reached the same conditional. Each dynamic branch occupies multiple entries in the GHBC. This gives the branch predictor state, and by knowing where it has been, it can use the GHBC to guess more intelligently where to go.

Running a simple simulation of the GHBC on random data and then sending the same data to the processor and timing execution shows that the Barcelona probably feeds back twelve or thirteen bits into its GHBC (shown in Figure 5). This means the processor

[1] We are unsure exactly which numbers cause what behavior, but the basic idea is correct.

can remember relatively large conditional patterns (that may even be longer than twelve conditionals).

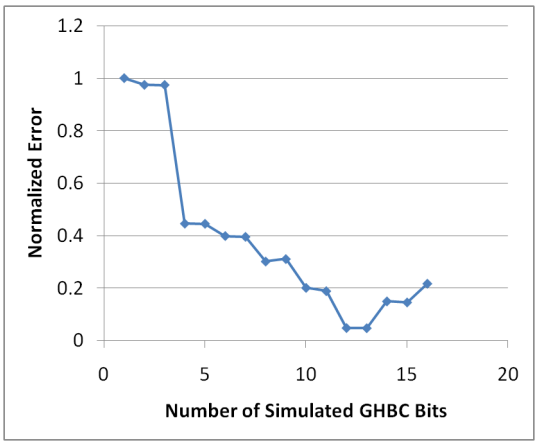


Figure 5: Branch mispredictions
Error between measured and simulated GHBC.

The weakness of the branch predictor lies in the fact that the branch history is global. For branches inside long loops (which are quite often the most costly branches), the loop conditional will hold half of the global branch history as either constant zeros or ones (because loops are based on very constant branches, with only the last break conditional being unique). This reduces our twelve bit history to a six bit history! Fortunately, loop unrolling make the loop branch history less relevant in comparison to the full global branch history. Also, if unrolling allows conditionals to be separated a significant amount, prediction rates should also go up based on the fact the GHBCs of multiple branches are being trained rather than those of just one.

It is important to put Figure 5 in perspective. We purposefully generated worst case input for the branch predictors. Real pseudo-random branching will probably not exhibit such sharp performance differences between each global history length. If a code follows some pattern for some short amount of time, the branch predictor will probably pick up on it. Also, a branch misprediction will not cost code of any reasonable size that much time. The AMD 10h Optimization guides says a branch misprediction will cost at most ten cycles. As our test code, we used a simple conditional addition statement. Even with all the misses, execution time only doubled in the worst case scenario. For code containing a serious number of addition or multiplication operations, branch misprediction problems would probably disappear.

4.1 Branch Types

Aside from the regular floating point comparison hardware, there are two other ways to optimize floating point conditionals on the Barcelona. First, and most important, is recognition of conditional moves. Second is use of the integer comparator unit instead of floating point (as described in [1]).

Conditional Move: In some cases, branches are used to simply choose data rather than operate on it. An example is "`use s1 unless c is true, then use s2`". The SSE instruction set was designed to accommodate this structure, and the equivalent SSE conditional move runs much faster than the equivalent code branches due to both SIMD parallelism and pipelinability.

The steps for the Conditional Move Operation for this statement are

1. Initialized data to a default value (`s1`),
2. Evaluate the conditional, using a SIMD comparison, leaving a mask in a target register indicating elements the need to change,
3. `AND` with values of (`s2 - s1`) in an offset register, and
4. Add the offset register to the original (`s1`), leaving the correct value in the final register.

Integer Comparison: Using the integer unit for floating point comparisons as described by AMD[1] is somewhat effective. It does break full IEEE-754 compatibility, but the performance gains are meaningful and it is easy to implement these changes in real code.

In C, the conditionals look something like

```
if (*( (INTTYPE *) &floatingdata ) < CONSTANT )
```

where `INTTYPE`, `CONSTANT`, and the direction of the comparison depend on the desired operation. The appropriate values are given the the 10h Optimization Guide.

As a simple test, we fed linear arrays of single precision floats through a conventional floating point code conditional, an integer-based float point code conditional, a single element SSE conditional move, and a four element SIMD SSE conditional move. In order to take greater advantage of the GHBC, we also included versions of both of the code branchers unrolled four times.

For one test we used random values for the input float array, and for the other we used conditionals representative of very looplike code (very infrequent changes). Results of these experiments are shown in Figure 6. The "blocked" tests reference the full SIMD and unrolled conditional functions.

From this we observe that

- the SIMD conditional moves scaled naturally (by a factor of 4) with multiple data inputs, and performed virtually the same on random and non-random data;
- single element conditional move beat the code conditionals on random data but lost on non-random data;
- for non-random data, all code conditionals performed the same (neither unrolling or feeding data through the integer comparators helped);

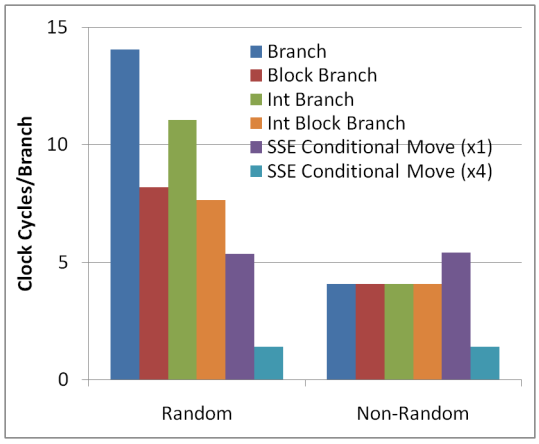


Figure 6: Branch speed comparison, various conditional types

- for random data, the integer comparators performed better than their respective floating point comparators; and
- for random data, the unrolled comparators performed much better than their rolled counterparts (indicating the data was not really random, and that the branch prediction unit was able to find patterns).

Of note, if these conditionals were not independent and could not be pipelined, then the performance gains of using the SSE conditional moves would drop off severely because there would be no parallelization to exploit, and the regular conditionals may become faster in all cases.

5 Two More Tests

We examine two more computations often found in scientific codes, each of which presents performance challenges to x86 architectures.

5.1 Computing the sine

Virtually all x86 processors have hardware versions of the sine function to retain compliance with the old 387 floating point unit. This instruction is chosen for simple operations with no performance constraints because it seamlessly supports full 80-bit floating point numbers. However, the 387 sine is not pipelined, and sequential FPU sine calculations can occur only when previous ones have finished.

Because the computation of the sine is based on a simple Taylor expansion, it fits well to the SSE instruction set and can be separated into two independent stages: take modular input, then apply Taylor expansion. We organized tests using combinations of several approaches, listed in Table 1, the performance of which is shown in Figure 7.

Blocked	Instructions were arranged with blocks of `FADD` instructions separate from blocks of `FMUL` instructions.
Shuffled	`FADD` instructions interlaced evenly when possible through `FMUL` instructions.
Staged Rounding	Input values were all rounded before any were sent through the Taylor expansion.
Non-staged Rounding	Input values were rounded as they were requested by the Taylor expansion.
Pipelined	Instructions were arranged to virtually guarantee availability of work for the out-of-order scheduler (mainly through calculation of different sine values in parallel).
Non-Pipelined	Out-of-order scheduler left to recognize independence of operations itself.

Table 1: Sine function code organization strategies.

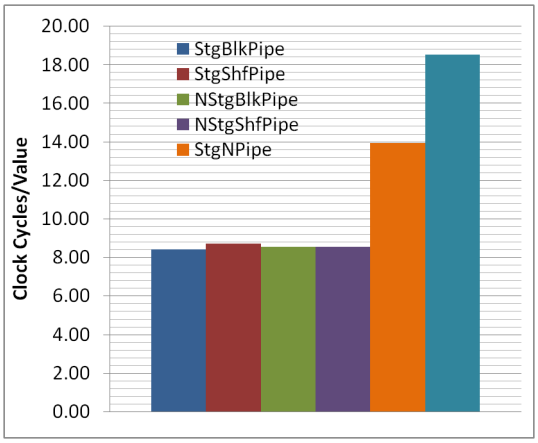


Figure 7: Sine implementations
Legend: `Stg`: Staged, `Blk`: Blocked, `Shf`: Shuffled; `Pipe`: Pipelined. Prepending with an "`N`" means that configuration is *not* implemented.

From this we observe:

- If the pipeline has something to do, code will run relatively efficiently. All the Pipelined implementations worked about the same.
- Breaking up the dependent stages into different loops (as seen in Staged Rounding, Non-Pipelined vs. Non-staged Rounding, Non-Pipelined) helps the scheduler work much more efficiently when pipelining is not already manually implemented.
- The difference in blocking and shuffling instructions was largely irrelevant.

Of practical note, we did not use a sustainable method of modular arithmetic to fit the Taylor expansion. As inputs increased, the modular operation started destroying a

```
y = input_y[j];
x = input_x[j];
z = 0.0;
for( i = 0; i < ITERS; i++ ) {
   if ( y < 0.0f ) {
      s = 1.0f
   } else {
      s = -1.0f;
   }

   n_x = x - s * y * pow2i_ss[i];
   n_y = y + s * x * pow2i_ss[i];
   n_z = z - s * atan_pow2i_ss[i];

   x = n_x;
   y = n_y;
   z = n_z;
}
out_atan[j] = z;
```

Figure 8: CORDIC computation

significant amount of input precision. Also, convergence of the Taylor expansion is always an issue, and we spent virtually no time determining if the method or expansion (to the 19^{th} power) was really appropriate. Of note, even values as low as π showed error in the lowest precision bits.

5.2 CORDIC Arctangent

CORDIC (coordinate rotation digital computer) algorithms are a set of a simple iterative formulas popular in FPGAs for compactly implementing a wide variety of complex operations (sine, hyperbolic sine, square root, division, etc.). For a regular processor, it has limited use. However, it provides interesting tests for the Barcelona because, while lending itself to the SSE instruction set, a conditional in sequential iteration proves to be a tricky bottleneck. A C implementation is shown in Figure 8.

Evaluation of `pow2i_ss` and `atan_pow2i_ss` represent lookup tables for `powf(2.0f,-(float)i)` and `atanf(powf(2.0f,-(float)i))`. Scalars `x` and `y` are initialized to their respective values for the operation `atan2f(y,x)`, and `z` is initialized to zero. After `N` iterations, `z` represents an approximation to `atan2f(y,x)`.

There are two ways to utilize the SSE instruction set for this code. First and easiest is to use three separate registers, each containing a set of independent `x`, `y`, and `z` values for computing arctangents in parallel. This will not offer any speedup for cases where only one arctangent needs calculated, but it should give significant gains when four or more values need computed. The second method is a bit tricker, but overall requires fewer instructions

for cases where only a single arctangent needs calculated. In this method, dependent values of `x`, `y`, and `z` would be stored in adjacent elements of one vector register. By absorbing subtraction into another special lookup table, the processor could use three out of four possible operations per SSE instruction in solving for a single arctangent. Naturally though, the vector nature of this second algorithm would require some extra shuffling.

For the parallel case, everything but the `if` statement transforms directly into SSE SIMD code. Because adjacent elements are not dependent on each other, the multiplies, adds, and lookup tables are all very simple. In most cases the existence of `if` statements would make it impossible to efficiently use SIMD instructions; however, this if statement can be implemented as an SSE conditional move discussed earlier.

Again, for the vector case, everything maps well to the SSE instruction set (given a new lookup table). Conditionals can be implemented as either code branches or conditional moves.

For some reference, Figure 9 represents the log base two of the average difference be-

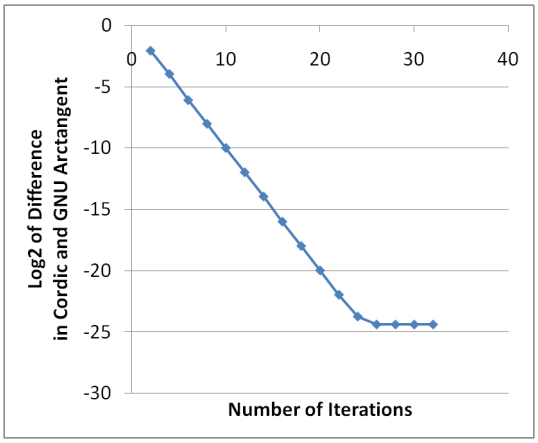


Figure 9: CORDIC vs. GCC arctangent computation comparison

tween the CORDIC and the GCC arctangents as a function of the number of iterations in the CORDIC implementation. The more negative numbers indicate less error, and the magnitude of the y value should roughly represent the number of effective bits of precision in the floating point number.

These tests are intended to illustrate the performance of different conditional types in a few different code arrangements:

Single Element, Parallel	One arctangent is calculated at a time using registers as in the parallel mode described above.
Single Element, Vector	One arctangent is calculated at a time using registers as in the vector mode described above.
Pipelined, Vector	This represents the Single Element, Vector code unrolled four times to improve pipelining of instructions

Results are shown in Figure 10. The horizontal axis represents the manner in which conditionals were computed. The vertical axis represents the average number of clock cycles

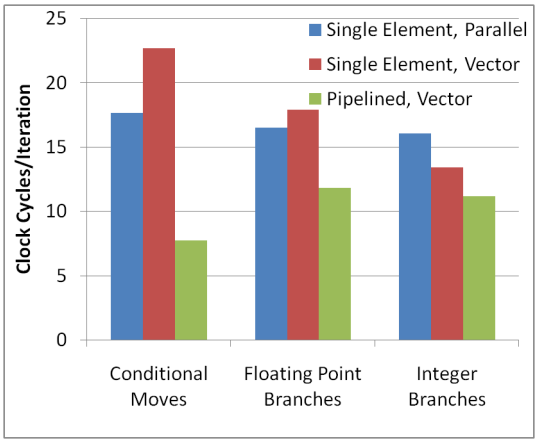


Figure 10: Effects of different conditional types on CORDIC calculations

each iteration took. While not as absolute or clear as previous ones, they still show a number of important facts:

- The integer unit is faster than the floating point unit in practice.
- Except where code can be pipelined, regular code conditionals beat single element conditional moves.
- The performance benefits of increasing operational density through the use of carefully packed vector data are unclear (since neither single element, non-pipelined configurations decisively beat the other).

While it is evident in the single element case without pipelining that code conditionals beat conditional moves, this changes entirely when either the full SIMD instructions are used or code is pipelined. The pipelined single element vector code proved to be the fastest of the last set of tests, but it is quite easy to do even better with different code types:

Pipelined, Vector	Same as Pipelined, Vector above.
Full SIMD, Parallel	Full SIMD operations on registers as in the parallel mode
Full SIMD, Pipelined, Parallel	Full SIMD operations on registers as in the parallel mode while calculating two sets of arctangent values in a pipelined manner (for a total of eight arctangent values at a time)

Results for these configurations are shown in Figure 11. Again, the conditional moves in the full SSE SIMD pipelined code easily surpasses the performance of all the branch based codes above.

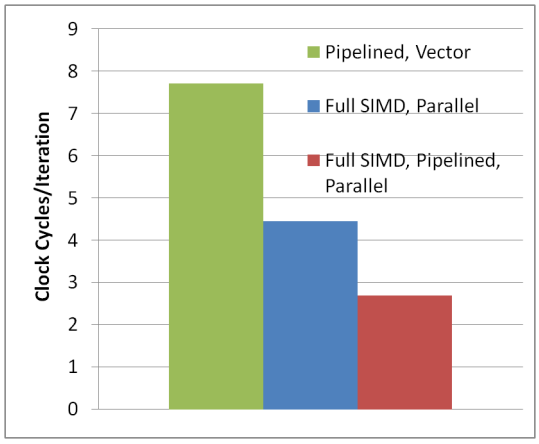


Figure 11: Comparison of various high end CORDIC implementations

5.3 Discussion

The difference in carefully shuffled instructions versus those carelessly blocked together are negligible. Also, instructional density differences will probably not be a problem for most codes, and, especially given the dual load capability of the AMD Barcelona, L1 cache is starting to be very close to free for all but the highest end applications. In the sine tests, buffering to level one caches proved to be almost invisible.

Further, the integer comparison unit performs better than the floating point comparator in practice. This will probably only be true for cases where the conditional is a major part of the computation (as in the CORDIC case).

Finally, if operations can be unrolled, broken at stages, or otherwise made such that the code is pipelined and the out-of-order scheduler has more than one series of operations to work on, then performance will be relatively good.

6 Conclusion

The information in this paper will probably not be useful in immediately improving performance in any stable application, as most system libraries probably offer better performing and more complete sine and arctangent implementations, and most developers have no need to work at the assembly level. Rather, this paper is meant to highlight how well the Barcelona processor core is able to use its advanced execution features to make code run fast and to what extent developers need to be aware of its low level intricacies.

From that perspective, the Barcelona processor is adept at executing balanced mult-add code at close to peak efficiency, even where the multiplies and adds are not arranged in a pattern conducive for superscalar execution. In this regard, the processor performs well as long as application code can feed it a balanced multiply/add stream. When the stream is either unbalanced, or part of it gets out of the look ahead window (which functioned up to around fifty instructions into the future), then performance will drop accordingly.

Tricky-to-implement, on-the-fly reordering and buffering accounted for some measurable

speedups of about seven percent for complex multiplication. However, this can introduce an extra load on the cache heirarchy, which may prove more detrimental than any shuffling gain. While these obsessive instruction-saving reorderings may have been relevant on older architectures, the Barcelona handles shuffling and vector operations in general much more smoothly.

SSE conditional moves proved to be dramatically more predictable than the branch predictor, but they did not always perform better than the branch predictor when code was not pipelined. If code was pipelinable and SIMD friendly, then the various conditional moves proved to be much faster. However, when pipelining is not easy (and code conditionals are the better choice), integer based conditionals operate at least as fast, if not faster, than floating point ones. As the processor cannot translate code branches to conditional moves on the fly (or floating point branches to integer branches), these modifications must be explicity configured in an application (perhaps through use of Intel Intrinsics[3] or other code constructs).

For the most part, all of this means that relatively efficient C code will translate to efficient execution. The Barcelona core hides its intricacies well, so as long as no unreasonable expectations are placed on the branch predictor, it is safe to program the Barcelona core as a simple, cached, vector processor.

Acknowlwedgment

This work was significantly enhanced by discussions with Brian Waldecker of AMD.

References

[1] AMD. Software Optimization Guide for AMD Family 10h Processors. www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/40546.pdf.

[2] Hans de Vries. Understanding the detailed Architecture of AMD's 64 bit Core. www.chip-architect.com/news/2003_09_21_Detailed_Architecture_of_AMDs_64bit_Core.html, September 2003.

[3] Intel. Intel 64 and IA-32 Architectures Software Developer's Manuals. http://www.intel.com/products/processor/manuals/.