A Survey of Microprocessor Architectures

Jeremy Langston
Electrical and Computer Engineering
Tennessee Technological University
Cookeville, TN 38505, USA
Email: jwlangston21@tntech.edu

Abstract

In this paper, an introductory description of modern microprocessors is given. The goal is to cover the many different aspects of computer architecture without getting weighted down with details. Many of the current trends in design are discussed, along with the differences between several Intel, AMD, and Sun microprocessors. A brief introduction of many broad ideas in architecture accompany to bring the readers to the same understanding, as well as defining certain concepts. The broad ideas are then taken into deeper detail, explaining certain processor implementations.

I. Introduction

The microprocessor has come a long way since the first Intel 4004 chip was released in 1971 [1]. At the time, large room-sized computers were typical and no small-scale solutions were available. Due to a recent advance in transitor technology, small processors could be realized without the use of bulky vacuum tubes. At a mere 2,300 transitors, the 4004 rivaled computers currently in use. Needs began to grow at a staggering rate necessitating advances in processor technology. Costs were also a major factor in the progression of microprocessors. In Moore's famous paper [2], the underlying tone of integrated circuit (IC) growth was mandated by costs. His short report looked at the future of the electronics industry and proposed that "the complexity for minimum component costs has increased at a rate of roughly a factor of two per year". This is commonly referred to as Moore's Law, but often is used in an incorrect manner.

Jumping thirty years later, microprocessors have surpassed their predecessors unlike any other technology has done. For instance, the Intel Itanium 2 with 9 MB of cache has 592 million transitors and can be purchased by an average consumer. Compare this to the ENIAC with 18,000 vacuum tubes [1] at a price that can only be amortized by the largest firms. Processing speed has also increased dramatically from under 1 MHz to over 3 GHz.

There is a lot more than these mere numbers that make current microprocessors what they are today. Differences in technologies and implementations allow for a wide array of uses and markets. Applications at a high scale may include servers, workstations, and laptop computers, which are the primary focus of this paper. However, microprocessors are not limited to these machine types. They can be found in embedded designs requiring high mobility, entertainment systems, automobile control, and so on. The key differences in microprocessor implementations are resources required, power consumption, and technologies required. For instance, a mobile application typically relies on batteries for a power source. Reducing power consumption enables the batteries to last longer and thus allow longer usage times. A back-end data server concerns itself more with high bandwidth and low latency.

In this paper, several mainstream microprocessors and their architectures are presented, as summarized in Table I. These range from high-end, scalable microprocessors like the AMD

Opteron to the energy-aware Intel Core Duo. As expected, these microprocessors can vary quite extensively in implementation, but often maintain similarities. All modern day microarchitectures implement caching of some form and varying levels, which is a way of decreasing the time delay for memory accesses as explained in the next section. Differences can occur in cache organization, number of levels, caching schemes, eviction, and so on. Each cache is tailored to its end target primarily based on necessity and cost. Other technologies, such as the Itanium 2's register stack engine, are manufacturer-specific and cannot be found elsewhere. This paper looks at many of the distinguishing features of the individual microprocessors, noting what they do and why they are needed, as well as the similarities to other microprocessors.

In the remainder of this paper, a brief background in Section 2 brings the reader up to speed. Section 3 delves into the particular implementations of a microprocessor's architecture, excluding the ISA. A short comparison shows the differences in ISAs in Section 4, and Section 5 concludes this paper.

II. Background

In this section, many of the broad techniques and technologies implemented in modern microprocessors are explained. This gives a baseline against which different implementations can be distinguished. This section may be skipped if the reader is comfortable with such architectural concepts.

A. 32-bit vs 64-bit

A recent change in microprocessors is the natural word size. Since the push to 32-bit chips in 1985 with the Intel 80386 [3], most all modern processors use 32-bits as their word sizes. Not only do register widths change to 32-bits, but data bus widths change. Without this change, it take multiple operations for the upper 16-bits and the lower 16-bits. Using a 16-bit bus in a 32-bit system would be detrimental to the possible bandwidth. Increases in word sizes also allow programs to access more memory locations. For example, a 32-bit microprocessor can address $(2^{32} = 4GB)$ different memory locations - whether it is allowed to is another story.

Today, the next shift underway in going to 64-bit microprocessors. These are currently employed in server environments where large numbers are used in mathematical calculations and large amounts of memory need to be accessed. While 32-bit microprocessors were able to process larger numbers than allowed by hardware, the process was much slower as extra instructions were required to hack the overall instruction sequence together. Equally important is the dramatic increase is accessible memory: up to $(2^{64} = 18,000PB)$. Even the largest supercomputers of today are far from this scale. However, when designing a microprocessor, the needs of the future must be taken into account, without going overboard. It should be noted that 64-bit microprocessors are nothing new. Not until recently has the need risen high in the enterprise environments.

A major problem with making switches in word sizes is that software written using smaller word sizes can become unusable. Without some way of achieving backwards compatibility, all software will need to be altered before running it on the larger word sized microprocessor. This has been taken into account, however. In the Intel Itanium 2, compatibility for the IA-32 architecture is provided at the beginning of the pipeline in the IA-32 Engine, allowing the 32-bit applications to take advantage of the other technologies implemented by the Itanium 2 [4].

B. Multi-core, Multithreading, and Multiprocessors

Manufacturers began noticing a trend while increasing the clock frequency in their microprocessors. The feasibility of manufacture lessened due to heat generation and power consumption [5]. Coupled with the way computers multitask, a paradigm shift moved the industry into using multiple units or cores. Apart from

Processor	Release Year	CPU Speed	Word Size	# of Cores	Cache L1I,L1D	L2	L3
Intel 4004	1971	740kHz	4	1	-	-	-
Intel Pentium 4	2000	1.4-3.6GHz	32	1	32kB, 32kB	256kB-1MB	-
AMD Opteron	2002	1.4-3GHz	64	1	128kB, 128kB	16MB	
Intel Itanium 2	2002	0.9-1.6GHz	64	1	16kB, 16kB	256kB	3MB
Sun UltraSPARC IV	2003	1.05-1.35GHz	64	1	32kB, 32kB	16MB	
Intel Core Duo	2006	1.06-2.33GHz	32	2	32kB, 32kB	2MB	
Intel Core 2 Duo	2006	1.06-3GHz	64	2	32kB, 32kB	4MB	-

TABLE I
FEATURES OF SOME MICROPROCESSORS AND THEIR CACHES. DATA COLLECTED FROM INTEL, AMD, AND SUN DATASHEETS.

adding individual execution units, which are explained in the next section, multithreading and multi-core microprocessors allow multiple threads or processes to be executed simultaneously without need to perform a context switch. Multithreading microprocessors use a single core to execute two threads, possibly unrelated, at the same time. The operating system sees such a microprocessor as two logical processors. The Intel Pentium 4's implementation is referred to as Hyper-threading [6].

Some of the downfalls of using multithreading is that only thread execution is simultaneous, not for processes, and programmer awareness. A thread is basically a light-weight process that shares the parent's memory. A program using multiple processes instead of threads does not benefit from multithreading. Also, in order for an application to take advantage of multithreading, the programmer must be aware of the problems associated with sharing memory between threads while writing the application. Using multiple processes evades the problem of shared memory, enabling the programmer to focus more on the end result.

Multi-core architectures essentially have two or more single-core processing units on a single die. Depending on the implementation, each core may have its own levels of cache and other architectural components. Applications are not tied down to relying on threads for increased throughput, but can also use processes. Many of the benefits to using multi-core microprocessors can be found in [5]. A full detailed explanation of multi-core architectures is out of the scope of this paper.

As noted in [7], a middle ground can be found in implementing multithreading practices on a multi-core microprocessor. This allows for threads to still be utilized differently than processes, preserving the light-weight attribute. However, multiple processes are still optimized. This could lead to an application utilizing four logical processing cores from one dual-core microprocessor.

Multiprocessor arrangements are simply systems with more than one physical microprocessor. While not as popular in consumer systems, enterprise servers have used multiprocessor arrangements for many years. This allows for highly scalable expansions in processing throughput. There are downsides to multiprocessor systems compared to multi-core systems. The distance between processing cores is now much farther away. Signal propagation must be taken into account with added delays for interprocessor communications. Caching is also hindered. Any data that is needed by processes/threads running on separate microprocessors must be locked and updated so that old data is not used.

C. Memory Access

1) Cache: Cache deals with the singlemost important barrier in overall system utilization: memory access. Cache is a way to gradually decrease the access time for a slow component with a fast component, that is of a smaller

capacity, using the concept of locality. Simply using all fast components such as SRAM is not feasible due to high costs and other reasons. Consider a simplistic computer with a tape backup, a hard disk, a DDR memory module, and a microprocessor with registers. Here, the hard disk is a cache for the tape backup and the DDR memory modules is a cache for the hard disk. Cache as it pertains to this paper is that which is on the microprocessor or die itself. In the same way that there are multiple levels of cache outside of the microprocessor, there are levels within. Typically there are 2 or 3 levels inside. Each is a little faster and smaller as the level approaches direct contact with the execution core. These levels are labeled as L1 beginning closest to the execution core, L2, and so on.

The cache not only contains data such as variables, but also the instructions that make up programs. One method of optimizing the cache is using separate caches for each the instruction and data memory, as they both exhibit different access patterns. This is typically always done at the L1 cache subarchitecture, and, depending on the architecture, lower levels are typically unified. The separate instruction and data level one caches are labeled L1I and L1D, respectively.

Many caching techniques are in circulation and are out of the scope of this paper, such as identifying temporal/spatial locality and replacement alogorithms. However, for clarity, this paper overviews the main three types of cache organization: direct, fully associative, and set associative [8]. In brief, each memory block can be loaded into cache at certain places in a cache, or cachelines. For direct-mapped cache, each memory block has one and only one place in cache it can be stored. This makes finding the memory very fast, but poses a thrashing problem if the block already resident in that location is frequently used. Fully associative cache allows any block to be placed anywhere in the cache. Eviction is less of a problem, but finding the block is much slower. Set associative is the compromise of the two. Two, four, or more cache block locations comprise a set. For each memory block, a certain set is chosen using mod arithmetic. Within that set, the memory block can be stored at any location - the memory block has multiple locations, or ways. The standard naming convention for such a cache is n-way set associative, where n is the number of ways in a set.

2) Virtual Memory: The concept of virtual memory comes as a solution to two problems: memory protection and large programs. The latter has become less of a concern, as few programs consume more main memory than a system has. Memory protection, on the other hand, is always a relevant problem. For security and reliability, processes should only be able to access the memory allocated to it. Virtual memory works by dividing memory into pages and allocating them to processes [8]. Virtual memory pages are analogous to cache blocks. Processes use virtual addresses that are unique to them. When a process must get or send data, it uses the virtual address which must be translated into a physical address, referred to as address translation. These translations are stored to memory. Since all of the translations are stored in a relatively slow medium and each memory access would require two accesses (translation and intended access), an optimization is employed. Recently accessed address translations are copied to a construct called a translation lookaside buffer, or TLB. The TLB is like a cache for the translations and is normally associated with cache levels. Each level cache would have a TLB with entries that match to the cache's physical addresses.

D. Pipelining

Instructions have a series of steps involved in their execution. The most common steps are fetching the instruction, decoding the instruction, fetching operands as needed, executing the instruction, and storing results as needed. Different implementations use varying amounts of steps as well as what each step does. At

each step, the other components are left unused. The idea of pipelining deals with overlapping this sequence of steps so that all components (data bus, decode logic, ALU, etc.) are utilized 100% of the time. Pipelining has become very complex due to the amount of components in question, but moreso because of the possibility of hazards [8]. These hazards fall into three categories: data, structural, and branch. Data hazards occur when the result of one instruction is required on a subsequent instruction that is already in the pipeline. Structural hazards happen when subsequent instructions require access to a component that is used elsewhere in the pipeline. Branch hazards occur when the program counter (PC) is changed, but subsequent instructions are already in the pipeline. Many options are available to overcome these hazards, including using branch predictions for branch hazards.

III. Microarchitectures

A microarchitecture is, simply put, the portion of the microprocessor's architecture that is not the instruction set. As such, it covers the implementation of the ISA on the hardware level. This section describes the different microarchitecture implementations of past and present. Two main implementations covered are Intel's NetBurst and EPIC. NetBurst, first appearing on the Pentium 4, is characterized by an out-of-order processing, execution trace cache, and a rapid execution engine [6], which are discussed below.

The EPIC microarchitecture, or explicitly parallel instruction computing which is similar to VLIW, appears as the basis for the Itanium. In NetBurst and other microarchitectures, the execution is parallelized at the hardware level at runtime, and often out-of-order. EPIC offloads the parallelization to the compiler at compile-time. This makes the compiler more advanced, but saves complexity in the microprocessor hardware. In implementation, the compiler identifies instructions that can be executed in parallel and groups them into sets of three [9]. Each set is given a label which shows the instruction categories contained. For example, a memory access, an integer ALU instruction, and a branch would have the label MIB. This lets the hardware dispersal logic know which execution units the contains instructions should go to.

In contrast to explicitly parallel instruction computing in which program execution is sequential, other microarchitectures use out-of-order processing, also referred to as dynamic scheduling. Since the program is not compiled in such a manner to give the best sequence of instruction executions, the hardware determines which instructions to execute. As stated previously, out-of-order computing requires much more advanced execution hardware, but relies less on compiler optimization.

The following subsections detail the different advanced techniques in execution, memory access, and power saving. There are many more concerns in real microarchitecture design, but for brevity, they are omitted from this paper.

A. Execution Techniques

In the early days of microprocessors, microarchitectures were much simpler. A microprocessor would only sequentially execute a single pipeline at a time. The mainstream concept of having multiple execution units (known as being superscalar) caught on with the Intel Pentium I [3]. Superscalar is not the same as having multiple separate processing cores. The execution units are the portions of the ALU that perform memory accesses, integer and floating-point operations, and branches. This can be seen in Figure 1, the pipeline of the Itanium 2. Per the figure, the execution units are seen on the EXE/FP1 stage in the pipeline. There are 2 integer, 3 memory, 3 branch, and 2 floating-point execution units. The NetBurst architecture's Rapid Execution Engine allows for up to six operations each cycle, which include 2 double-speed integer ALUs and a complex integer ALU, load and store address generation units, floating-point ALU, and a floating-point move unit [6].

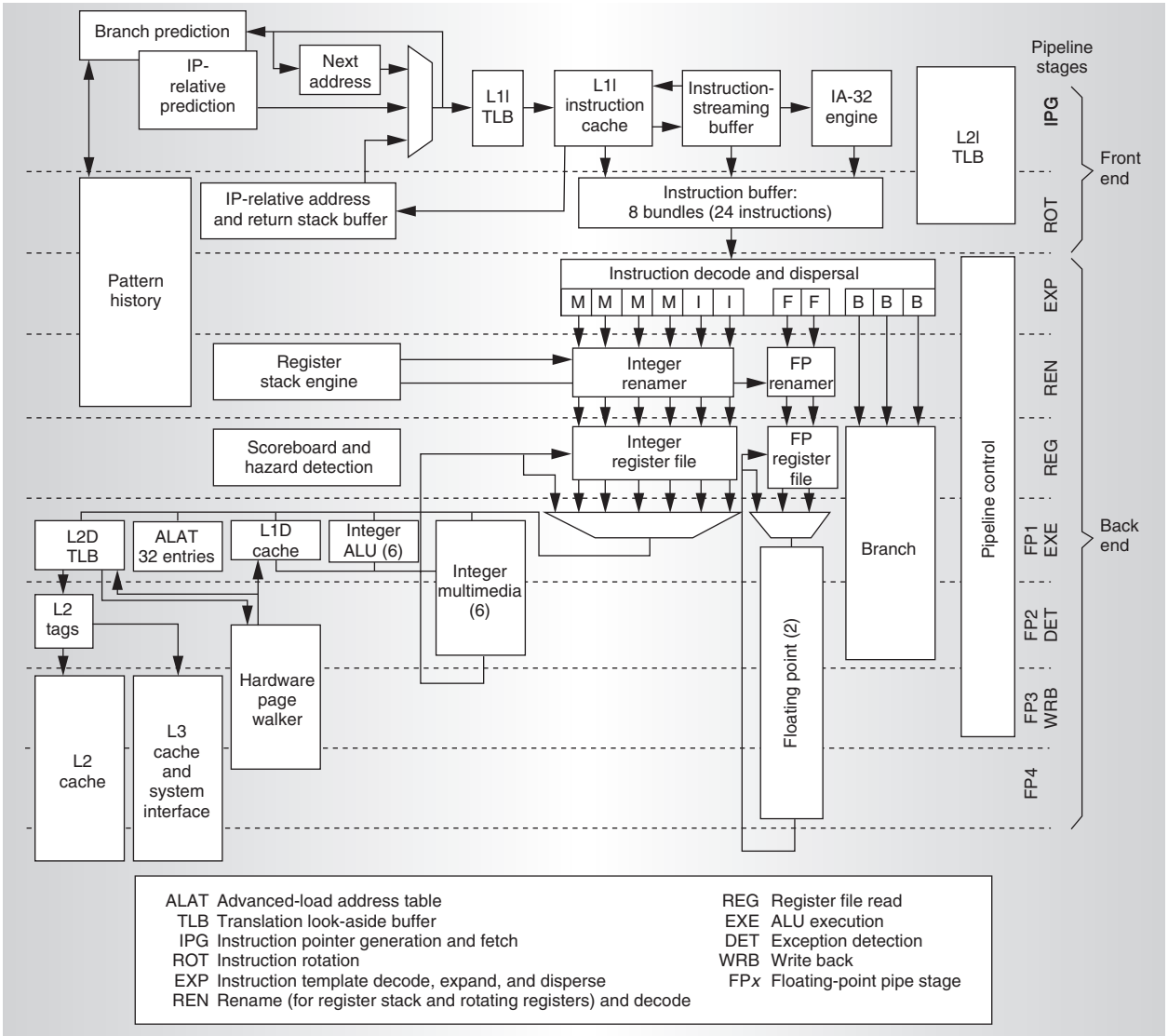

Fig. 1. Intel Itanium 2 pipeline.

Note the stages in the Itanium 2's pipeline [9]. The first stage, IPG, procures the next program counter and instruction. Many factors come into play. Not only is the instruction gathered from L1I cache, but branch prediction occurs, the ISB (instruction streaming buffer) is used as needed, and the IA-32 compatibility engine is envoked when required. Since the Itanium 2 has multiple execution units, multiple instructions are sent to the next stage simultaneously. Branch prediction is a way to increase the efficiency of the pipeline. Consider the case of an if-else branch. If the instruction execution yields true, then go to some program counter; if false, go to another. Branch prediction hypothesizes which direction to go to, in this case using a pattern history. The pattern history is updated as the program executes and notices which direction particular branching statements go. Branch prediction is not the same as speculation. The difference, albeit a subtle one, is that in branch prediction the instruction is only fetched and issued, whereas speculation proceeds to execute the instruction [8]. When a speculated instruction is found to be the

correctly executed instruction, the results are committed to the register file (set of registers) or memory.

Many implementations, including the Itanium 2, use branch predication. Here, each instruction sequence following a branch is given a distinct value by the compiler. Both sequences (to branch or not) proceed through the pipeline until the branch instruction finishes executing. When the correct destination is determined, the instruction sequence with the corresponding predicate bits continues, and the other sequence and its results are dropped.

Due to the nature of the EPIC paradigm, the Itanium 2 has a non-stalling pipeline. This means that instructions proceeding through the pipeline do not need to wait due to hazards. The Sun UltraSPARC IV has a 14 stage non-stalling pipeline, but by a different method [10]. Instead of stalling the entired pipeline waiting for results or a resource, it is dropped and re-executed. The most common cause of waits in a stalling pipeline is memory access - particularly in cases where there is a cache miss. The UltraSPARC also allows for speculative memory loads. Here the memory may or may not have the correct, most up-to-date data, but in the case that the memory will not be changed, the load is performed. If the data was correct, the instruction is commited, otherwise it is re-executed and loaded.

In the NetBurst microarchitecture, complex instructions are decoded into their simpler counterparts, called uops (micro operations). The uops are stored in the Execution Trace Cache. Once decoded, they do not have to be decoded again, so this takes out that step from the main execution loop [6]. This allows for dynamic traces to be formed and executed, lowering hit time (time to find in cache), but drastically increasing design complexity. The AMD Opteron provides space for up to 72 uops to be reordered and fetched to a three-way superscalar core [11]. The Pentium 4's Execution Trace Cache has space for 12,000 uops.

B. Caching and Memory

Caching design often comes to the decision of cost and necessity. As the cache gets closer to the registers and pipeline, the cost increases dramatically due to the design and speed of SRAM. Different versions of a microprocessor are manufactured and sold partially for this reason [12]. Clearly, a larger sized cache level is better suited for holding the memory required of a program, but an increase in size has its drawbacks. In order for the microprocessor to find what is needed in cache, it must perform a search of some sort. The larger the cache, the more complex the searching hardware is. A compromise can be struck between the caching method (e.g. set-associative), size of cache, and the number of virtual address bits used in finding the spot in cache, as well as using fully-associative TLBs. Most all microprocessors use a set-associative cache with sets from two to sixteen ways, between 256 kB and 16 MB of SRAM, and around 20 to 36 address bits for location. Refer back to Table 1 for a brief overview of typical cache sizes and levels.

In the AMD x86-64 architecture, the Opteron supports only 48 of the 64 total virual address bits, and less for physical addresses [8], [11]. The top 36 bits of the virtual address (the page number) are used to get the address translation from the TLB which combines with the remaining bits of the virtual address. This allows the cache subsystem to find the precise place in cache the data or instruction resides. If the data or instruction is misses in the L1 cache, the request is sent to L2. If it has not be cached at all, a complete cache miss occurs and the data or instruction must be pulled from main memory or further.

An optimized TLB and cache structure was used in the Intel Itanium 2 called Prevalidated-Tag Cache [9]. Ordinarily the cache holds a physical address against which the TLB searches. Instead of having the TLB check every physical address in the cache, an association to the TLB entries is stored in the cache. This tightly coupled organization al-

lows for much faster lookups using a one-hot match line implementation. One-hot means that only one bit of cache tag is set to one, while the others are set to zero. During TLB lookup, the normal fully-associative TLB finds the entry that matches the virtual address. Once completed, the entry drives a one with other entries driving a zero. This "match line" is then compared against the cache tags. If a way is found (matching tags), then the data is distributed from the cache. The L2I TLB is 128-entry so, in implementation, the match line will be 128 bits wide.

Apart from finding memory contents in cache and loading them, storing new data, and replacement strategies, which cannot be sufficiently covered here, there is the notion of reliability. Cache, like any storage system, is succeptible to data loss and corruption. Parity bits and error correction codes (ECC) are often the choices, with the faster-to-process parity used at upper cache levels. This can be seen in the Itanium 2 microprocessor [9]. Data reliability is a primary concern for servers in the enterprise setting. The UltraSPARC IV, being built with reliability in mind, takes protection a step further with the external L2 cache bus [10].

C. Power Consumption

The Intel Core Duo focuses on the mobile market, namely laptop computers. One of the biggest concerns in mobile computing is the battery life. The usefulness of a laptop can be drastically reduced if it can only run for a small amount of time while not plugged in. The Core Duo enhances the power consumption rate by including many different sleep states (using ACPI) and disabling one of its two cores [13], [14]. Disabling cores induces problems such as changing the clock for one core, but leaving the other core unchanged. The power states are listed in Table 2. For states C0 through C3, each core has been made independent. The operating system watches activity on the cores and when a core is inactive, the OS can move to the next power state. When none of the cores have been active for some time, the OS can put both cores into states C4 and DC4, which reduces the power consumption by altering the supplied voltages.

State	Description
C0 - Active	CPU is on
C1 - Auto Halt	Core clock is off
C2 - Stop clock	Core and bus clock are off
C3 - Deep sleep	Clock generator is off
C4 - Deeper sleep	Reduced VCC
DC4 - Deeper C4	Further reduced VCC

TABLE II
INTEL CORE DUO SLEEP STATES.

IV. INSTRUCTION SET ARCHITECTURE

The ISA is the complement to the microarchitecture; it defines how the microprocessor is used and with what instructions. The most common instruction sets are the x86 which were brought from the days of the 8086 [15], IA-64, and x86-64. The x86 is a 16 bit ISA that has been modified since 1978 to accommodate 32 bit microprocessors of today, such as the Pentium 4. The x87 instruction subset was introduced in the 1980's to allow for more mathematical instructions. The x86-64 is the further alteration of the x86 ISA to allow for 64 bits as seen in the AMD Opteron. Due to the nature of the Itanium 2, a new ISA was devised to allow for the EPIC processing called the IA-64. Sun's proprietary ISA is known as VIS, which is used on the UltraSPARC series of microprocessors.

In addition to these ISAs, there have been revisions for SIMD, or single-instruction-multiple-data, called SSE (streaming SIMD extension). These additions found in SSE, SSE2, and SSE3 allow for fastor vector processing [8]. With a single instruction, multiple executions can be done on a set of data. Suppose, for example, there is an array that needs have its contents all multiplied by the same value. This can be done using SSE/2/3.

V. CONCLUSION

Modern microprocessors have grown by decisively since their first inception. Not only

have they grown in breadth by the amount of processing that can be achieved, they have grown in depth by the new technologies adopted. This paper has demonstrated the many facets to modern microprocessor architecture, while not becoming weighted down with technical details. It has been seen that the latest trend in computing has been in allowing multiple executions of instructions. This varies in implementation from a completely separate processor, to separate cores, to separate execution units within cores (superscalar), to multithreading. Cache remains a popular research avenue as the biggest limiting factor in microprocessor has become the memory interface. With proper implementation, this limitation can be minimized, but not completely avoided until memory speeds catch up with the speeds of microprocessors.

REFERENCES

[1] Intel. (2006) The intel 4004 microprocessor. [Online]. Available: http://www.intel.com/museum/archives/4004facts.htm

[2] G. E. Moore, "Cramming more components onto integrated circuits," *IEEE Electronics*, vol. 38, no. 8, 1965.

[3] L. Null and J. Lobur, *Computer Organization and Architecture*. Jones and Bartlett Publishers, 2003.

[4] I. Corp., "Intel itanium 2 processor - hardware developer's manual," *Document No. 251109-001*, 2002.

[5] I. Advanced Micro Devices, "Multi-core processors - the next evolution in computing," *Whitepaper*, 2005.

[6] D. Boggs and A. Baktha, "The microarchitecture of the intel pentium 4 processor on 90nm technology," *Intel Technology Journal*, vol. 8, pp. 1–17, 2004.

[7] J. Fruehe, "Planning considerations for multicore processor technology," *Dell Corp. Whitepaper*, 2005.

[8] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 4th ed. Morgan Kaufmann Publishers, 2007.

[9] D. S. C. McNairy, "Itanium 2 processor microarchitecture," *IEEE Micro*, pp. 44–55, 2003.

[10] S. Microsystems, "The ultrasparc iv processor architecture," *Whitepaper*, 2003.

[11] C. Keltcher, K. McGrath, A. Ahmed, and P. Conway, "The amd opteron processor for multiprocessor servers," *IEEE Micro*, pp. 66–76, 2003.

[12] Intel. (Feb. 2006) Intel pentium 4 processor supporting hyper-threading technology specifications. [Online]. Available: http://www.intel.com/products/processor/pentium4/specs.htm

[13] S. Gochman, A. Mendelson, A. Naveh, and E. Rotem, "Introduction to intel core duo processor architecture," *Intel Technology Journal*, vol. 10, pp. 89–96, 2006.

[14] A. N. et al, "Power and thermal management in the intel core duo processor," *Intel Technology Journal*, vol. 10, pp. 109–122, 2006.

[15] Wikipedia. (March 2008) x86 architecture. [Online]. Available: http://en.wikipedia.org/wiki/X86