Franklin W. Olin College of Engineering

Proceedings of

CA 2004

the

Fall 2004 ENGR 3410 Computer Architecture Class

`<ca@lists.olin.edu>`

Needham, Massachusetts, December 2004

Contents

Preface
Mark L. Chang ii
Digital Signal Processors
Grant R. Hutchins, Leighton M. Ige 2
FPGA Architecture, Algorithms, and Applications
Kathleen King, Sarah Zwicker 7
A History of Nintendo Architecture
Jesus Fernandez, Steve Shannon 13
Graphical Processing Units: An Examination of Use and Function
Michael Crayton, Mike Foss, Dan Lindquist, Kevin Tostado . . . 17
Dual Core On-Chip Multiprocessors and the IBM Power5
Thomas C. Cecil 24
Pong on a Xilinx Spartan3 Field Programmable Gate Array
Kate Cummings, Chris Murphy, Joy Poisel 28
The PowerPC Architecture: A Taxonomy, History, and Analysis
Alex Dorsk, Katie Rivard, Nick Zola 32
A Brief Introduction to Quantum Computation
Michael W. Curtis 38
Beauty and the Beast: AltiVec and MMX Vector Processing Units
Drew Harry, Janet Tsai 44
VLIW and Superscalar Processing
Sean Munson, Clara Cho 49

Preface

The members of the Fall 2004 ENGR 3410 Computer Architecture class are pleased to present the Proceedings of CA 2004. This is the first offering of the class, and this document represents only a portion of the work done by the students this semester. Presented here are the final papers describing their end of semester projects.

The students were given a few weeks to research any Computer Architecture topic that interested them. The deliverables included a brief in-class presentation as well as the documents compiled here. The goal of the project was to encourage group learning of advanced Computer Architecture topics not covered in class and give the students a chance to develop independent research skills.

The projects were wonderfully varied, ranging from literature reviews to actual hardware implementations. It is important to note for the reader that several students chose to complete an optional lab as part of the final project and implemented a pipelined MIPS processor in Verilog. This work is not reflected in these proceedings, however the scope of these students' research projects were adjusted accordingly.

As the instructor for CA 2004, I am most proud of the work accomplished by the students this semester. I would like to thank them for their hard work and their creative zeal. For more information regarding Computer Architecture at F. W. Olin College of Engineering, please visit our class web page at *http://ca.ece.olin.edu*.

Mark L. Chang
Assistant Professor
Franklin W. Olin College of Engineering

Digital Signal Processors

Grant R. Hutchins, Leighton M. Ige

***Abstract*— Digital Signal Processors (DSPs) operate on digital signals in realtime applications. They have several benefits over other processor architectures, due to specialized designs and instruction sets. This article offers an overview of general properties of DSPs and comments on their applications in the real world.**

I. Introduction

We live in an analog world composed of continuous functions: the light you see, the sound you hear, and the words you speak. However, for many reasons these signals are converted into digital representations for processing and transmission. It is often cheaper, more accurate, and computationally simpler to manipulate a digital representation of a continuous signal, since there are discrete time slices of data to process instead of a signal with infinite resolution. The power of the digital revolution lies in the ability of digital systems to approximate the analog world. Once in a discrete representation, signals can be digitally processed to have different effects, such as the reduction of background noise in a microphone feed or the encryption and compression of a cell phone call.

Digital signal processors (DSP) are specific microprocessors that specialize in the manipulation of signals in real time. In the specific applications of devices that interface with the world, live inputs must be sampled, processed, decoded, and outputted on the fly. Thus, DSPs must be designed to be able to complete a lot of computations very quickly to be effective. The rate at which these DSPs operate must be at least as fast as the rate of its input, otherwise backlogs will be created. Once the process no longer proceeds in real time you lose the synchronization between inputs and outputs, with the signal becoming decoupled. This scenario can be likened to that of a cell phone conversation in which you had to wait a long time for your phone to decode the received audio signal from your correspondent. When there is too long of a delay many of the benefits of a synchronous conversation are lost and one would be no worse off sending a written letter.

Today, by far, the largest implementation of DSPs is in cell phones. Every cell phone sold contains a DSP, which is necessary to perform all the encoding and decoding of the audio signals which are transmitted through the wireless communications network. Additionally, most new phones now have advanced multimedia capabilities which need to be processed in real time. These functions such as picture taking, music playing, and video recording, along with their non-integrated device counterparts, require DSPs for many fast computations. DSPs can be found at the core of many of the products in the growing multimedia device segment.

Within the audio domain, DSPs are rather prevalent and can be found in CD players as well as high end home audio components. Many audio receivers offer the capability of real time equalization adjustments to audio playback. The computationally intensive algorithms for transforming audio to sound as if it is being played in a large concert hall or a dance hall are handled by DSPs. Additionally, many filtering and noise reduction algorithms can be implemented to clean up audio signals. Convolution and Fast Fourier Transforms require many sequential multiplications and additions, a procedure for which DSPs are optimized.

DSPs are also very flexible in their implementation as they can be reprogrammed to perform different calculations. For example, a cell phone provider could decide to support a new data feature on its phones. The DSP could then be updated to decode the new data packets being sent to the phone.

While there are currently many DSPs implemented in everyday devices, they are becoming more prevalent due to their high performance, flexibility, and continually decreasing prices. The capability to perform complex manipulations on data in real time can enable many systems to incorporate more inputs from the world.

II. Common DSP Tasks

In digital signal processing, a few simple types of algorithms dominate. Thus, DSPs highlight their abilities to perform functions to

A. Multiply-Accumulate

Time domain filters are a common operation in digital signal processing. Digital filters are implemented by taking streams of current and prior input and output data, scaling them with coefficients that perform a certain filter function, and summing them to generate the next output.

Some filters have hundreds or thousands of coefficients. Each of these multiplication operations must occur in order to generate each output sample. However, data rates in audio DSP applications are often as high as 96,000 samples per second (video can go much higher), meaning that millions of multiplications and accumulations must happen every second. For this reason, DSPs are optimized for and benchmarked on their ability to perform multiply-accumulate operations. A common metric for DSP processors is MMACS, or millions of multiply-accumulates per second.

B. Fast Fourier Transform (FFT)

In most situations, signal data is represented in the time domain. That is, each separate word of signal data represents a discrete sample of the signal in time. However, many functions, such as equalization and pitch-shifting, operate on

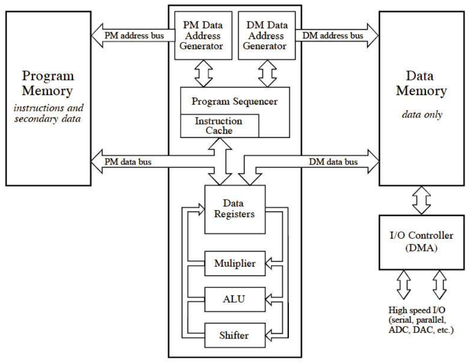


Fig. 1. Typical architecture for a DSP (taken from[1])

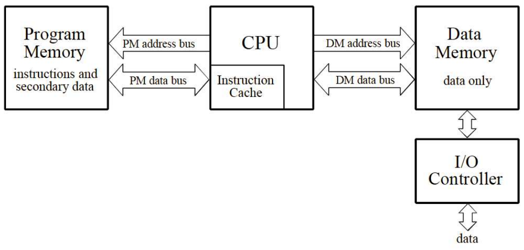


Fig. 2. Super Harvard Architecture (taken from [1])

the frequency domain, where each word represents the amount of information present in the signal at a particular frequency.

The Fast Fourier Transform is an algorithm which takes time domain data and converts it into corresponding frequency domain data. In order to do so, the data must be reshuffled and processed through multiple loops. Using a similar routine, the Inverse Fast Fourier Transform converts frequency domain data back into time domain data.

III. Architecture and Datapath

DSPs have several architectural properties that set them apart from other classes of processors. Each optimization has the common applications of DSPs in mind and aims to reduce the overhead associated with manipulating large digital signals in a repetitive fashion.

A. Memory

DSPs access memory in differing ways. Unlike traditional microprocessors, some DSPs have two data buses and memory files, which allows for better performance.

1) Von Neumann Architecture: The von Neumann architecture, named after the important mathematician John von Neumann, contains a single memory bus connected to a single processor. In this simple architecture, one memory file contains both instructions and data.

The data bus in the von Neumann architecture is serial, allowing only one value to be read from memory at a time. Consider a simple multiply-add instruction, which involves multiplying two values and adding them to a third value. This instruction requires three separate fetches from memory, namely the instruction, the two multiplication operands, and the addition operand.

If the memory is clocked at the same speed as the main processor, then this multiply-add example will necessarily take at least four clock cycles to complete. Therefore, some DSP chips use a modified von Neumann architecture in which clock the memory at a multiple of the processor clock and thus allow a single instruction to access multiple memory locations in one clock cycle.

2) Harvard Architecture: The Harvard architecture was developed by a group led by Howard Aiken at Harvard University during the 1940s. The Harvard architecture employs two separate memories, one for instructions and one for data. The processor has independent buses, so that program instructions and data can be fetched simultaneously.

Most DSPs use some sort of Harvard architecture. However, most digital signal processing routines are data intensive and thus require more than one load from data memory for each load from instruction memory. The original Harvard architecture does not support

3) Super Harvard Architecture (SHARC: Some products from Analog Devices use what they call a “Super Harvard architecture” (SHARC), such as their ADSP-2016x and ADSP-211xx families of DSPs. The SHARC processors add an instruction cache to the CPU and a dedicated I/O controller to the data memory.

The SHARC instruction cache takes advantage of the fact that digital signal processing routines make use of long repetitive loops. Since the same instructions are performed over and over, the instruction bus does not need to repeatedly access the instruction memory. Cache hits will happen a vast majority of the time, freeing the instruction bus to allow the CPU to read an additional operand.

However, since the freed bus connects to instruction memory, the programmer must place one set of operand data in instruction memory and the other in regular data memory. This requirement adds complexity to the organization of DSP application programs, but increases the throughput significantly.

The dedicated I/O controller connects directly to the data memory, allowing signals to be written and read to RAM directly at very high speeds without using the CPU. For example, some of Analog Devices’s models have two serial ports and six parallel ports which each operate on a 40 MHz clock. Using all six ports together gives data transfer at 240 MHz, which is more than enough for most real-time DSP applications and helps speed asynchronous processing. Some processors also include analog-to-digital and digital-to-analog converters, and thus can input and output analog signals directly.

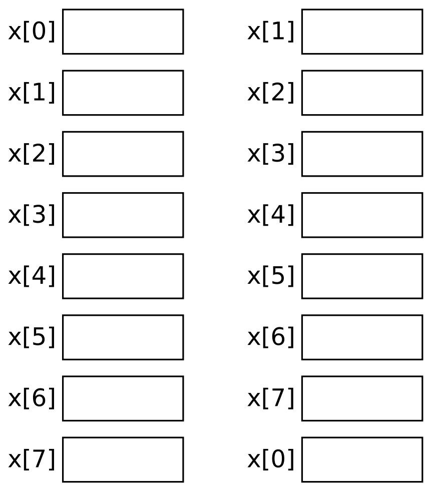


Fig. 3. Circular buffer, before and after incrementing loop

B. Addressing

DSPs have some optimizations that allow them to read memory more efficiently for digital signal processing tasks.

1) Circular Buffers: A circular buffer, shown in 3 is a continuous section of memory that a DSP increments through repeatedly, looping back to the first position after it accesses the final position. DSPs often contain Data Address Generators (DAGs) that can control multiple circular buffers. Each DAG holds the memory location, size, and other information about a particular circular buffer. In addition, a DAG can perform logic on these variables such as loop incrementing. This logic prevents applications from needing to tie up the processor with instructions to perform math and branching in order to keep repetitive loops running.

2) Bit-reversed Addresses: In addition to managing circular buffers, DAGs can be switched into a special bit-reversed mode in which the order of the address bits is inverted. Part of the Fast Fourier Transform algorithm involves inverting the bit order of the addresses which point to the data and then sorting the resulting addresses. This process shuffles the addresses, allowing the DSP to operate on the pieces of data in precisely the order the FFT algorithm calls for.

C. Parallelism

DSPs are optimized to run several unrelated options all at once. For instance, a DSP might allow a multiplier, an arithmetic logic unit, and a shifter to run in parallel, each processing a different value from a data register. For example, an Analog Devices SHARP processor can perform a multiply, an addition, two data moves, update two circular buffer pointers, and perform loop control all in one clock cycle.

On traditional microprocessors each of these procedures might take its own instruction. Instead of performing loop logic after each iteration, a program could instead perform a few initialization instructions which tell the DSP how to automatically loop through the next set of instructions, which for a typical loop might reduce the number of clock cycles by a factor of ten or more from that of a traditional microprocessor.

D. Precision

Not all DSPs represent numerical values in the same way. Depending on the application and certain trade-offs in performance and price, one representation might make more sense than another.

1) Fixed-point: The cheapest and most basic types of DSPs perform operations on fixed-point numbers. At the most basic level, all DSPs must be able to manipulate integers, in order to process loops, access memory locations, and perform other basic operations.

For fixed-point representation, DSPs often employ a “fractional” notation in addition to the standard integer notation. Fractional notation places all values between 0 and 1 for unsigned fractions and -1 and 1 for signed fractions. For example, a 16-bit unsigned fraction representation has 65,536 different possible values, ranging evenly from 0 to 1. A fractional notation is useful for multiply-accumulate operations, since coefficients for finite impulse response and infinite impulse response filters need to be less than 1 in order to prevent excessive resonance.

2) Floating-point: For higher quality applications, DSPs often employ floating-point representations. The most common format, ANSI/IEEE Std. 754-1985, has 32-bit words and ranges from as high as $\pm 3.4 \times 10^{38}$ and as low as $\pm 1.2 \times 10^{-38}$, a range that dwarves that of comparable fixed-point representations.

However, since floating-point values often have twice as many bits per word as fixed-point values, data buses must be doubled in width. If memory is located off-chip, this translates to many more pins, leading to more difficult board layouts. Also, the larger size of words takes up more memory to represent the same amount of data.

Another important feature of floating-point representation is that the difference between two adjacent numbers is proportional to their values. In 16-bit fixed-point fractional notation two adjacent values might be separated by a difference as low as one-tenth the value, as with $\frac{10}{32,767}$ and $\frac{11}{32,767}$, for example. In the standard 32-bit floating-point representation, the difference between two adjacent numbers is set at about ten-millionth times smaller than their value.

The higher resolution of floating-point values allows for a higher signal-to-noise ratio. Since multiple multiply-accumulate operations introduce rounding errors over time, fixed-point representations have significant noise. Overall, using a standard deviation method to measure noise, 32-bit floating point outperforms 16-bit fixed-point by a factor of about 3,000.

3) Extended precision: Consider a routine that involves 100 fixed-point multiply-accumulate instructions. Since each instruction introduces a significant amount of noise, the final result has accumulated 100 times the noise of each instruction.

In order to prevent this situation, most DSP architectures have intermediate fixed-point registers of much higher resolution. For example, a 16-bit floating-point DSP might have a 40-bit extended precision accumulator, only stepping the signal down to 16-bit once the final value is ready to be written to memory.

IV. Special DSP Instructions

DSP processors have highly specialized instruction sets which allow assemblers and assembly programmers to get the best performance out of their architectures. Consider the following examples found at [2] from the Lucent DSP32C processor instruction set.

A. Parallel modification of address pointers

One common technique in DSP instructions is to manipulate the value of registers acting as pointers to memory. This manipulation occurs whenever the address of the register appears in code, depending on how it is referenced. For example, think of an assembly programmer implementing a multiply-accumulate operation using a value from memory that is pointed to by the register `rP`. The programmer can also tell the processor to increment the register in the same instruction by calling the register `rP++` right in the assembly code. Unlike in general purpose processors, the increment operator happens immediately within another instruction rather than requiring an instruction unto itself. A similar post-decrement can occur after reading the value. The Lucent DSP32C can perform as many as three of these pointer modifications, while performing another function, all within one instruction.

B. Bit-reversed post-increment

A highly specific instruction found in the Lucent DSP32C is the bit-reversed post-increment. Within one instruction, as in the previous example, the processor can also increment the pointer to the next value in an array as if the address were in reverse order. This bit-reversed traversal of an array only ever gets used for FFTs and inverse FFTs in practice. Frequency domain operations are common enough in digital signal processing that chip designers spend the extra hard work to incorporate this highly specialized feature. With this special instruction, DSPs can perform real-time FFT and frequency domain functions, whereas most general purpose processors might only be able to perform frequency domain operations asynchronously.

C. Instruction length

Due to the number of operations that DSP processors perform in parallel, some specialized designs have emerged. Some processors with 32-bit instructions, for example, have an option to run two 16-bit multiply-accumulate instructions in one clock cycle. In addition, some DSPs have moved toward a Very Long Instruction Word (VLIW), as big as 64 bits in width, to run several operations in parallel. A VLIW instruction gives the programmer limited control over which four 16-bit sub-instructions should run in parallel.

V. Advantages of DSPs over other processors

Digital Signal Processors perform operations that focus entirely on specific applications, such as MP3 decompression or realtime video filters. However, DSPs are not the only processors able to perform these tasks. Every day millions of people listen to MP3s on their desktop computers, for example. There are several alternatives to DSPs, each with its own benefits and problems.

A. Field-Programmable Gate Arrays (FGPAs)

Field-Programmable Gate Arrays (FPGA) are logic chips that can, as the name implies, be reprogrammed in the field to have entirely different functionality. An FPGA is programmed with a certain desired gate array and operates accordingly. Thus it is very fast in that its implementation is customized for the particular task at the gate level. It is essentially like achieving custom silicon results without the hurdle of actually manufacturing a unique chip. When the desired operation of the FPGA changes, a new gate array can be written over the chip.

An FPGA would seem to be a viable alternative to a DSP as it could be customized for the same sorts of computationally intensive tasks. While the FPGA would certainly be faster than the DSP due to its dedicated logic, it is much more expensive and requires more energy. Additionally, until recently many FPGAs did not offer enough programmable gates to provide the full functionality of a DSP. In an attempt to address this deficiency, FPGAs are being manufactured with enhanced DSP features to enable this capability. While the DSPs have a significant price advantage at under at average of $6 per chip [3], the added flexibility of a DSP-capable FPGA could justify the additional expense.

B. Application-Specific Integrated Circuits (ASICs)

Application-Specific Integrated Circuits (ASIC) are essentially customized silicon chips that are specifically configured for a particular task. ASICs are very efficient in their execution of tasks and utilize very dissipate very little power. However, a big limitation to their utility is their lack of adaptability. DSPs are a combination of hardware and software that allow the operation and functionality of a chip to be constantly changed. If new features or computations need to be completed, a new software program can be loaded into the DSP on the fly. If a change to an ASIC is desired, a new silicon chip must be designed and manufactured. Thus there are large financial and time impediments limiting the flexibility of ASIC design. In a related application, DSP cores are sometimes used within ASICs, creating custom DSP-based chips with specific functionality.

C. Traditional processors

General-Purpose Multiprocessors (GPP) are optimized to run entire systems, such as a desktop computer, and thus must be able to complete many different tasks, most of which are unnecessary for dedicated signal processing. They are large chips that dissipate a lot of power. DSPs are optimized for the completion of relatively few tasks many times in succession. Thus their entire architecture is customized to this functionality and results in significant performance increases over GPPs. For example, DSPs can complete multiple arithmetic functions within a single cycle. In fact, current DSPs execute up to eight operations within one clock cycle. Additionally, customized addressing modes as well as parallel memory access capabilities enhance the performance of a DSP. GPPs are typically not well-suited for the computationally intensive roles served by DSPs.

VI. General Citations

Portions of this paper were inspired by [4], [5], [6], and [7].

References

[1] S. W. Smith, *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing, 1997.
[2] (2004) Bores signal processing - introduction to dsp. [Online]. Available: http://www.bores.com/courses/intro/
[3] (2001) Ee times: Dsps. [Online]. Available: http://www.eetimes.com/story/OEG20010604S0057
[4] (2002) Comp.dsp faq: Dsp chips. [Online]. Available: http://www.bdti.com/faq/3.htm
[5] Howstuffworks - inside a digital cell phone. [Online]. Available: http://electronics.howstuffworks.com/inside-cell-phone.htm
[6] (2004) multimedia phones. [Online]. Available: http://www.1st-in-cell-phones.com/21797-multimedia-phones.html
[7] (2004) Getting started : What is dsp? [Online]. Available: http://dspvillage.ti.com/docs/catalog/dspplatform/details.jhtml?templateId=5121 &path=templatedata/cm/dspdetail/data/vil_getstd_whatis

FPGA Architecture, Algorithms, and Applications

Kathleen King and Sarah Zwicker
Franklin W. Olin College of Engineering
Needham, Massachusetts 02492
Email: {kathleen.king, sarah.zwicker}@students.olin.edu

***Abstract*— FPGAs are reconfigurable devices that have received much attention for their ability to exploit parallelism in algorithms to produce speedup. The reasons for their success lie in the architecture of the FPGA, the algorithms involved in programming FPGAs, and the type of the program being run on the FPGA. This paper presents an overview of the development of the current FPGA architecture, the algorithms that have been developed to automate the process of programming FPGAs, and the common reasons for speedup in applications.**

I. Introduction

Field Programmable Gate Arrays (FPGAs) belong to the family of reconfigurable devices. A reconfigurable device is a piece of hardware that can be customized easily, often making it faster than regular computer processors and more flexible to change than hardwired circuits. Application-Specific Integrated Circuits (ASICs) are hardwired circuits that can be used to provide efficient solutions to very specific problems. It is difficult and time consuming to design and produce an ASIC, even though once the machinery is in place to produce one, millions more can be made easily. They are thus excellent solutions when mass production is required or when execution speed is desirable at any cost. Processors, on the other hand, are affordable and extremely flexible; they can do many different jobs using the same hardware. Unfortunately, processors are also much slower than ASICs because they are not optimized for any particular task. Thus, it is really unsurprising that reconfigurable devices like FPGAs have become very popular in recent years; they fill the gap between these two popular devices [1].

This paper attempts to explain to an audience of computer architecture students what FPGAs are, why they have the architecture they do and what they are good at. An overview of FPGAs in the context of other reconfigurable devices is given to provide context. Then, the advantages and disadvantages of using various control technologies are considered. The FPGA architecture section discusses the development of the current 4-LUT-based island-style architecture. Next, the process of programming an FPGA is examined, including automated software algorithms that have been developed. Finally some current applications of FPGAs are explored.

II. Reconfigurable Devices

Reconfigurable devices consist of arrays of gates connected by a network of wires and switches. Opening or closing the switches controls the behavior of the circuit. The simplest of these devices is the Programmable Logic Array (PLA), which can easily be used to emulate any truth table. PLAs connect a set of inputs to an array of switches. These switches select inputs and send them to various AND gates, whose outputs can then be selected by another set of switches to be delivered to a column of OR gates. A sample PLA is shown in Figure 1. A Programmable Array Logic (PAL) is sometimes defined exactly the same as a PLA, but other times it is specified that the outputs of a PAL may be connected back to the device as inputs, which is not allowed in PLAs.

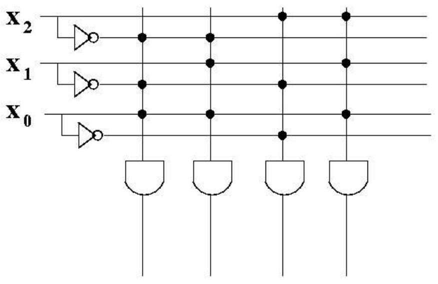


Fig. 1. A Programmable Logic Array

PLAs and PALs devices are the simplest type of reconfigurable devices. One step above these arrays is the Generic Array Logic (GAL), which is similar to a PAL, except the output of each OR gate of a GAL feeds directly into an output logic macro cell. Each of these cells contains a D-flip-flop and so has the option of acting as a state holding element.

All of the devices discussed thus far fall under the category of Programmable Logic Devices (PLDs). While they may too simple to be widely useful, they are extremely inexpensive and easy to use so they are still commercially available and quite common. However, for more complicated functions, none of these simple pieces of logic are sufficient so Complex Programmable Logic Devices (CPLDs) were developed. A CPLD is a collection of PLDs along with a switch matrix, which is a set of wires with programmable switches that give the user greater flexibility, as shown in Figure 2.

However, CPLDs also have limited functionality, since the switch matrix is not usually large enough to be capable of connecting any two logic blocks. As more PLDs, wires, and switches are added to the CPLD, it eventually becomes an FPGA [2]. FPGAs can be used to replicate almost any kind of hardware, including full processors, although not every function takes advantage of the FPGAs unique abilities. Applications of FPGAs will be discussed in a later section of

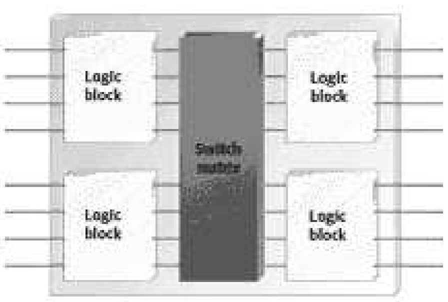


Fig. 2. A Complex Programmable Logic Device

this paper. First the architecture and programming of an FPGA shall be considered.

III. FPGA Control Technology

The function of the FPGA chip is programmed by controlling millions of switches. The technology chosen for these switches has the potential to greatly enhance or detract from the efficiency of the FPGA. One early switching method was to use Erasable Programmable Read-Only Memory (EPROM). EPROM, once programmed, remains so for up to twenty years, unless it is exposed to intense ultraviolet light. It does not require any power source to retain its programming. However, it is significantly slower than other forms of memory, so it has become obsolete in modern FPGAs.

The most common form of control in commercial FPGAs is Static Random Access Memory (SRAM). SRAM is composed of several transistors which form two inverters that have two stable states, which correspond to 0 and 1. However, SRAM requires a continuous power supply in order to retain its state, although unlike dynamic RAM, which uses a capacitor to maintain its value, SRAM does not need to be refreshed periodically. The main advantages of SRAM are that it is extremely fast and very easy to reprogram; when its power is turned off, its memory is erased. Of course, these values can be saved off-chip on a non-volatile type of memory and reloaded when power is restored to the FPGA [4].

Antifuses are the other form of control still in use, although they are far less common than SRAM. Antifuses have very high resistance, so when they are placed between wires they stop current from flowing, acting as an open switch. When programming an antifuse-based FPGA, a high voltage is sent through the antifuses that need to act as closed switches. This causes the fuse to ”blow”, creating a low resistance path that allows current to flow. One obvious disadvantage of antifuse FPGAs is that they are not reprogrammable [4].

This may seem to defeat the entire purpose of an FPGA, but they still fill a special niche in the market. First of all, antifuse-based boards are faster even than SRAM, since once they have been programmed the FPGAs are essentially hardwired circuits; there are no switches to delay signals. Also, while it may be necessary to pay $100 to replace an FPGA board after one discovers that a mistake has been made in its programming, $100 is far less than the millions of dollars necessary to fabricate a new ASIC. Finally, and perhaps most importantly, antifuses are not subject to damage from cosmic rays. Cosmic rays are capable of interfering with transistors, the building blocks of SRAM, which means that they can cause an SRAM bit to suddenly change from a 0 to a 1, or vice versa. Under normal circumstances, this is not a major concern; SRAM FPGAs can be shielded, and on earth cosmic ray interference is fairly uncommon. However, FPGAs are often used in space, where heavy shielding is an unpractical expense and there is no atmosphere to protect devices from cosmic rays. Thus, antifuse FPGAs offer a fitting solution, since cosmic rays have no effect on them.

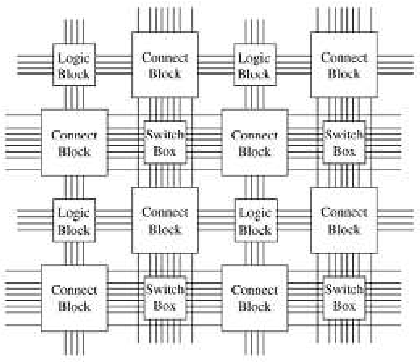


Fig. 3. An Island-Style FPGA Architecture

However, despite the utility of antifuses in specialized realms, SRAM FPGAs are almost universally used in terrestrial applications. Therefore, throughout the remainder of this paper it shall be assumed that SRAM is used for the switches otherwise explicitly stated.

IV. FPGA Architecture

Architecturally, the essential ingredients of FPGAs are logic blocks and a routing network that connects the logic blocks. The logic blocks may each contain some sort of PLD and possibly some small amount of memory. They are laid out in a regular pattern over the FPGA chip. This can be done in a variety of ways, but the most common is an island style, where islands of logic blocks float in a sea of routing wires and switches, as shown in Figure 3. There are also other types of islands in the sea, called connection blocks and switch blocks. Switch blocks allow signals to be passed off onto different wires. Connection blocks control what signals are sent in to logic blocks as inputs and what signals are accepted as outputs. These will be discussed in greater detail later [1].

First, logic blocks will be considered. The goal in designing FPGAs is to use the minimum amount of area used by the logic blocks while minimizing delays, thereby making the circuit as fast and efficient as possible. Numerous studies have been done, mostly in the early development of FPGAs, to determine the optimal size of a logic block. Size, in this case, is talked about as corresponding to the amount of work that the block can do. The size of the logic blocks on a chip is described as its granularity. Small blocks are said to be fine-grained, while larger blocks are more coarse-grained. Some extremely fine-grained logic blocks used in early FPGAs contained as little as several transistors, while extremely coarse-grained blocks might have all the functionality of a full ALU. FPGAs that are homogeneous with only one extreme grain size are now obsolete. Fine-grained logic blocks require too many wires and switches and routing is a primary cause of delay in FPGAs. Exceedingly coarse-grained blocks lead to faster FPGAs on some occasions, but they minimize the overall flexibility of the device [4].

Thus, logic block sizes falling somewhere between these extremes were destined to become the popular choice. Medium-grained logic blocks can be based on one of three ideas. The blocks could contain arrays of simple gates, making them virtually identical to PALs or GALs. These blocks have a great deal of flexibility, but it is difficult to take full advantage of all the available gates, so this design for a logic block ends up wasting significant amounts of space. Multiplexers and Look-Up Tables (LUTs) offer other possible solutions [4].

A LUT is in essence a piece of SRAM. The inputs to a LUT give the address where the desired value is stored. For a particular boolean function, a LUT can be made by storing the correct outputs in the slots to which the inputs point. Thus, LUTs can output in a single step for any number of inputs, while multiplexers require $log_2(n)$ steps for an n-input function. Clearly, LUTs work more efficiently than multiplexers. Therefore current logic blocks are based on LUTs in order to minimize delay and avoid wasting space. LUTs may have any number of inputs, leading to logic blocks of anywhere from medium to very coarse granularity. The optimum size of logic blocks has been determined experimentally. One particularly important paper was [5], which will now be reviewed briefly.

Logic blocks with one input and some number, K, outputs were tested on various benchmark circuits implemented on FPGAs. Figures 4, 5, and 6 show the results of these experiments for various numbers of inputs, assuming different switch delays, τ_s. Recall that switch delays are the result of switch choice; that is, whether control is performed with SRAM, antifuses, or EPROM determines the switch delay. Since modern FPGAs almost universally use SRAM, this element of the results presented is obsolete.

Figure 4 shows that if blocks with more inputs are used, then a smaller total number blocks is necessary, with a sharp decrease until $K = 4$ and flattening after that point. However, Figure 5 shows that blocks with more inputs have significantly greater associated delay per block, which is to be expected since more inputs require more routing. Figure 6 surprisingly shows that the total path delay per block had a local or absolute minimum at $K = 4$. Thus, [5] concludes that 4, or possibly 5, input logic blocks are optimal for performance.

One may get a sense from this research, which shows that 4-input logic blocks are preferable, and the discussion above, in which it was decided that logic blocks based on LUTs are optimal, that 4-LUTs are the best choice for LUTs. Others have done more research and confirmed that 4-LUTs are indeed best for optimizing both speed and area of FPGAs, as is discussed in [4]. In fact, 4-LUTs remain the industry standard for FPGAs, although it is no longer universally accepted that 4-inputs are ideal for every logic block.

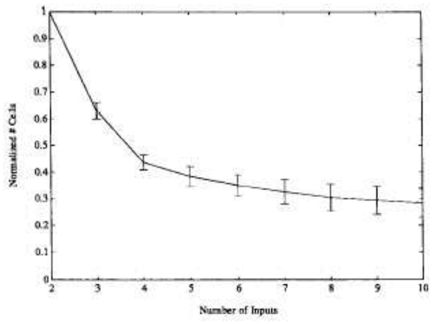


Fig. 4. Number of Logic Blocks, for Blocks with K Inputs

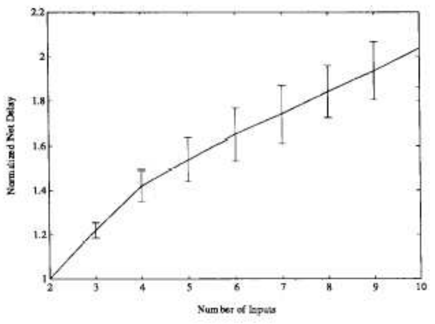


Fig. 5. Delay per Block, for Blocks with K Inputs

More recent papers, such as [6], have discovered that sometimes grouping several connected 4-LUTs into a single logic block minimizes delays and area. The idea behind [6] is that hardwired connections eliminate switches, which take up space and delay signals. Through experimentation similar to [5], it was discovered hardwiring four 4-LUTs in a particular tree structure maximizes the efficiency of a logic block.

Of course, logic blocks are not the only part of an FPGA which may be improved. In fact, most modern research is directed toward optimizing the routing of wires between logic

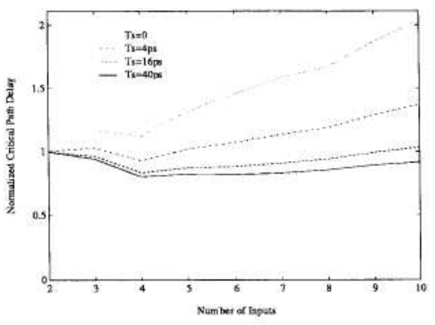


Fig. 6. Total Path Delay, for Blocks with K Inputs

blocks. Since only ten percent of an FPGA's area is used for logic while ninety percent is used for routing, this focus is not surprising [1]. Since routing also has the potential to cause a great deal of delay, many people work on trying to find ways to place wires efficiently to minimize both area and delay. The research done in [6] addressed this problem by attempting to determine the best lengths of wires. Adding switches to the network of wires allows for greater flexibility, but switches cause delays and take up space. However, while long wires are faster, if they are not necessary then they waste area. In [6], it was determined that it is preferable to have 56 percent of wires be short, 25 percent medium, and 19 percent long.

The routing research example describes a very high-level architectural decision. However, knowing that architectural features of FPGAs is only a starting point. To actually implement a circuit on an FPGA requires that the logic blocks be arranged on the board and connected with wires. These tasks are extremely difficult to perform, and currently much research is being done to determine the best methods of doing them. The next sections of this paper shall present common algorithms used in implementing programs on FPGAs.

V. Software

There are currently many automatic software solutions capable of taking high level code, designing the corresponding logic blocks and wiring scheme, for downloading to an FPGA. The following section discusses the process by which software algorithms perform these tasks. First, it is necessary to have code that describes a list of connections between logic gates in a low-level language like Verilog. The automated process begins with the technology mapping of the designer's code. The technology mapping process takes the code and translates it into a netlist of logic blocks specific to the FPGA. Often circuit libraries, which contain the logic-block-based structures of commonly used elements, such as adders and multipliers are used to facilitate this process. Also at this stage, if the system contains more than one FPGA, the netlist is partitioned into the parts that will go on each FPGA.

Next, a placement algorithm is used to assign the netlist of logic blocs to specific locations on the FPGA. Floorplanning, or the grouping of highly connected logic blocks into particular regions on the FPGA, can help the placement algorithm to achieve a reasonable solution. Often the placement algorithm is based on simulated annealing, a heuristic process that can be run until a desired goal is reached. The goal of the placement algorithm is closely tied to the next step in the automated process, which is routing. Better placement means that wires will need to be routed through fewer connections, or switches, making the circuit run faster.

After placement is complete, the layout and netlist are used to route the wires, or signals, through the FPGA. Routing is limited by the fact that there is a specific number of wires that can be connected to each logic block and that can run in the channels between the logic block islands of the FPGA. Thus, the routing algorithms have to work within that constraint to find a good scheme. Routing has the most direct affect on the performance of the circuit; if there were inefficiencies in earlier processes, they increase delays in routing. Once routing is complete, a translation of the logic block and connection layout information can be easily created and downloaded onto the FPGA.

In the early days of FPGAs, placement and routing were done by hand, as was the individually programming each LUT and multiplexer. Manual circuit description is a time-consuming process and requires low-level customization for specific architectures. It can produce high-quality circuits in terms of algorithm speedups and exploited parallelism. However, as the FPGA has grown in number of logic blocks and routing wires, it has become increasingly convenient to have automated design tools available to do mapping, placement and routing.

A. Technology Mapping

Technology mapping converts code into circuit components based on the architecture of the logic blocks in the FPGA. If the code is high level like Java or C, it will need to be processed into a netlist. However, if it is low level such as Verilog or VHDL, it can be run through a technology map algorithm directly. As an example of technology mapping, if the FPGA hardware was composed of logic blocks containing 4-LUTs, some logic functions in the code that had more than 4 inputs would have to be spread over multiple logic blocks. Technology mapping algorithms can try to minimize the number of logic blocks used or optimize for routeability. Routing is made more complex by the increasing heterogeneity of commercial FPGAs, meaning that they involve a variety of sizes and types of logic blocks.

Because each FPGA may have different types of logic blocks arrayed in unique layouts, it has become the standard for FPGA manufacturers to offer circuit libraries. Circuit libraries incorporate low level details into an abstraction of common components such as adders and multipliers. In this way, common components can be designed and optimized by those who have specialized knowledge of their specific FPGA architecture. The FPGA can then be programmed from a higher level. The abstractions of these tools are generally architecture independent so that once a user learns how to program one FPGA, he does not need to spend time learning how to use the tools for another type.

Technology mapping can include more than one FPGA, in which case the netlist must first undergo a process called partitioning. Manual partitioning of pieces of the netlist can yeild highly optimized layouts. However, it is sometimes time consuming and complex. To address this problem, [7] discusses a method for automatically partitioning netlists for a variety of FPGA architectures. The best partitions have the fewest interconnections that have to run between different FPGAs. Finding the best partition is the NP-hard mincut problem. The algorithm presented in [7] attempts to solve the mincut problem heuristically.

B. Placement

After the logic blocks and their connections have been mapped, the logic blocks are placed on the FPGA board. Placement has also been shown to be an NP-hard problem. An intermediate step that can facilitate the placement process is a global placement or floorplanning. The floorplanning algorithm groups highly-connected components into clusters of logic cells and places these onto bounded regions of the FPGA board [1]. Also, macros, or repeated data paths, in circuits can be laid out as single units in their own regions. The placement algorithms can then be used to configure logic blocks within each region.

The most common placement algorithm is based on a simulated annealing as described in [8]. The metallurgical definition of annealing involves heating up a material, usually a metal, to allow the atoms to move. The atoms are capable of moving greater distances at higher temperatures. At higher energy states, dislocations begin to disappear and grains begin to grow, giving the desired effect of making the material more ductile. In simulated annealing, an initial placement of ’atoms’ (in this case logic blocks) is chosen and modified through repeated cycles until an acceptable solution is reached. The cycles begin with a ’high energy state,’ in which random logic blocks are allowed to swap places. For each potential switch, a cost function is used to estimate how good or bad this swap will be. As each cycle progresses, the threshold is raised for how good a swap has to be in order to be allowed. The annealing technique helps avoid local minima (as often occur in a greedy algorithm) by sometimes allowing swaps that do not have good cost values. Just as the cooling schedule is important in determining the ductility of a metal, the cooling schedule of the simulated annealing cycle is an important factor in the success of the algorithm at obtaining a good placement. Therefore, the annealing procedure is often time-consuming, on the order of hours [10].

Also important to the simulated annealing algorithm is the accuracy of the cost function in determining good choices. At this point, it can only be estimated how the logic blocks will be routed, and the estimates have to be decided quickly. One of the challenges of automating FPGA programming is that placement and routing are both NP hard problems. Thus, it is not surprising that a good algorithm that considers both problems at once has not yet been found. The flexibility of the resources of the board is also important. If too much of the board is in use, the algorithm may not be able to find an acceptable solution. The factors that affect flexibility include the architecture of the FPGA and the amount of the board that is in use.

C. Routing

Once placement of the logic blocks has been done, routing will determine which wires and switches, or nodes, to use for each signal. FPGA routing is fundamentally different from ASIC routing because the wires and connection points are limited, which means that global and local routing need to be done simultaneously. This and other distinctions make research done on ASIC routing of little use in helping to determine FPGA routing. The basic routing problem tries to minimize path length by minimizing the use of routing resources, while simultaneously trying to minimize delay. However, minimizing delay often requires taking longer but less congested paths.

One approach to minimizing delay is to attempt the routing of the entire FPGA and then search for areas that are not optimal. At these areas, ’rip-up and retry;’ that is, remove all of the wires and try to find a better local routing. The success of this method is highly order dependent: if the program does not start with a good node to optimize from, it will get stuck in a local minimum. Many software applications use the PathFinder, which is an iterative algorithm that allows signals to pick their preferred path and then negotiate with other signals for resources. During the first iteration, the signals are treated as independent with each node only costing its intrinsic delay. During subsequent iterations, the history of congestion is added into the cost of each node. This cost is multiplied by the number of signals using the node. Thus, signals that are sharing the same node in the most direct path gradually decide to use less congested nodes. Also, because of the history term, the cost of using congested nodes is permanently increased so paths that move off of congested nodes do not try to return. If the history term does not exceed the intrinsic delay, then the algorithm that PathFinder uses for negotiating congestion and delay routing has been proven to achieve the fastest implementation for the placement. In practice, the history term will exceed the intrinsic delay for very congested circuits, but even then, results are close to optimal [11].

All of these software algorithms facilitate the generation of efficient designs in a shorter time span than it would take to hand-map the program onto a modern FPGA. It is now possible to implement specific applications and utilize the qualities of the FPGA to achieve speedup on common programs.

VI. Applications

In general, FPGAs are good at applications that involve rapid prototyping of circuits, algorithm acceleration, and multiple processors, when the FPGA works alongside a CPU. Image processing is generally a good candidate for FPGAs because of the high level of fine-grain parallelism they can support. The ability of FPGAs to act as coprocessors makes them desirable for use in satellites. FPGAs can perform such functions as image compression for the CPU and also serve as a backup in case of problems.

One example of an application that took advantage of the computing power of FPGAs is a genetic algorithm used to solve the traveling salesman problem in [12]. Using a system of four FPGAs, an execution time speedup factor of four was achieved, compared to a software implementation of the same problem. Fine-grain parallelism accounted for more than a 38X overall speedup factor. Hard-wired control intrinsic to the FPGA hardware accounted for an overall speedup factor of 3X. There was also some speedup due to coarse-grain parallelism and random number generation. Parallelism in FPGAs often accounts for their speedup over software algorithms; however the algorithm used in this case was not ”embarrassingly parallel” as are some hardware design, which showed that FPGAs are valuable beyond their parallelism.

Many applications of FPGAs for algorithm acceleration have been found in the field of cryptography. For example, a certain process called sieving is used to factoring large numbers in the fastest published algorithms for breaking the public key cryptosystem designed by Rivest, Shamir and Adleman (RSA). [13] used an FPGA system to get a speedup of over 28X compared to the more traditional UltraSPARC Workstation running at 16 MHz. They estimate that an upgrade to FPGAs using 8ns SRAM will give a speedup factor of 160. If this change were made, theoretically the only two months would be required to break RSA-129 (the RSA encryption using 129 digit primes).

VII. Summary

This paper has attempted to present an overview of FPGAs at a level accessible to undergraduate engineering students. It has discussed where FPGAs fit into the family of reconfigurable devices and explained why SRAM is most commonly used in FPGA switches. The common island style architecture has been examined and the tradeoffs between space for logic blocks and routing considered. The development of 4-LUTs as the prominent element in logic blocks was discussed. Algorithms that perform mapping, placement, and routing on FPGAs were explained. Finally some applications that have witnessed significant speedup due to FPGA use were discussed, along with the reasons for the speedup. Thus, this paper has given a broad survey of the architecture, algorithms, and applications of FPGAs.

Acknowledgment

The authors would like to thank Mark Chang for his extensive help with the research for this paper. The authors would also like thank Sean Munson and Clara Cho for their generous commitment of time and camaraderie.

References

[1] K. Compton and S. Hauck, “Reconfi gurable Computing: A Survey of Systems and Software,” *ACM Computing Surveys*, vol. 34, no. 2, June 2002, pp. 171-210.

[2] M. Barr, “How Programmable Logic Workds,” *Netrino Publications*, url: http://www.netrino.com/Articles/ProgrammableLogic, 1999.

[3] B. Arbaugh and M. Hugue, *Programmable Logic array*, Computer Organization Notes, Dept. Comp. Sci., Univ. Maryland, url: http://www.cs.umd.edu/class/spring2003/cmsc311/Notes/Comb/pla.html 2003.

[4] J. Rose, A. El Gamal, and A. Sangiovanni-Vincentelli, “Architecture of Field-Programmable Gate Arrays,” *Proc. of the IEEE*, vol. 81, no. 7, July 1993, pp. 1013-1029.

[5] J. L. Kouloheris and A. El Gamal, “FPGA Performance versus Cell Granularity,” *Proc. Custom Integrated Circuits Conference*, 1991, pp 6.2.1-6.2.4.

[6] P. Chow, S. O. Seo, J. Rose, K. Chung, G. Páez-Monzón, and I. Rahardja, “The Design of an SRAM-Based Field-Programmable Gate Array - Part I: Architecture,” *IEEE Transactions On Very Large Scale Integration Systems*, vol. 7, no. 2, June 1999, pp. 191-197.

[7] U. Ober and M. Glesner, *Multiway Netlist Partitioning onto FPGA-based Board Architectures*, *IEEE Proc. Conf. on European Design Automation*, 1995.

[8] S. Kirkpatrick, C. D. Gelatt, and M. P. Vechi, “Optimization by Simulated Annealing,” *Science*, vol. 220, no. 4598, May 1983.

[9] M. L. Chang, *Variable Precision Analysis for FPGA Synthesis*, Ph.D. Dissertation, Dept. Elect. Eng., Univ. Washington, 2004.

[10] S. Hauck, *Multi-FPGA Systems*, Ph.D. Dissertation, Dept. Comp. Sci. and Eng., Univ. Washington, Sept. 1995, pp. 36-42.

[11] L. McMurchie and C. Ebeling, “PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs,” *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 1995, pp. 111-117.

[12] P. Graham and B. Nelson, “Genetic Algorithms in Software and in Hardware - a Performance Analysis of Workstation and Custom Computing Machine Implementations,” *IEEE Symposium on FPGAs for Custom Computing Machines*, 1996.

[13] H. J. Kim and W. H. Mangione-Smith, “Factoring Large Numbers with Programmable Hardware,” *ACM/SIGDA Int’l. Symp. on FPGAs*, 2000.

A History of Nintendo Architecture

Jesus Fernandez, Steve Shannon

***Abstract*— The video game industry shakedown of 1983 seemed to be the end of the industry in the States. In spite of this, it is at this time that Nintendo released the NES as an astronomical success for the company. The architectures of the NES, SNES and N64 systems are not only influenced by the ambitions of Nintendo's engineers, but also from external circumstances like economic climate, business partnerships, and marketing.**

***Index Terms*— Nintendo, NES, SNES, N64, Computer Architecture, Console Design**

I. Introduction

THE year is 1983. Atari has declared bankruptcy after reporting a loss of over $500 million from the failed releases of *E.T.* and *Pacman* for the Atari 2600. The economy itself is in a slump. The bottom has fallen out of the video game market, and the public as a whole has all but renounced the console system in favor of the emerging personal computer. From this description, one would think that introducing another new console only two years later would be the quickest form of corporate suicide. However, that's just what Hiroshi Yamauchi, Masayuki Uemura, and the engineers of Nintendo did, and through a series of intelligent marketing and business decisions, the "big N" single-handedly revitalized and redefined the video game industry as they saw fit.

Not surprisingly, the decisions that led to the success of Nintendo's first mass-market system played a direct role in how the system's infrastructure was designed. Then, as the industry grew back steadily and the pertaining technology caught on, competitors immerged once more. Nintendo had to keep evolving with the market in order to stay competitive, which meant updating its design and system capabilities. In addition, Nintendo's constant desire to push the limit of the industry and be the innovator had significant impact on the infrastructure as years went on.

II. Nintendo Entertainment System

The Nintendo Entertainment System (NES) – known as the Famicom in Japan – was first envisioned by Yamauchi to be a 16-bit console selling at a street price of $75 dollars. The goal was clear: to develop an inexpensive system that would be better than its competitors' offerings. While Yamauchi was able to secure some fairly advantageous deals with semiconductor companies and distributors, many decisions had to be made to cut costs in the system's design. Many desired "extras" like keyboards and modems were dropped completely despite being implemented in the hardware itself. [1]

Later on, the video game industry shakedown of 1983 also had an effect on the design of the system Nintendo would release in the US. Most noticeably, the look and feel of the system was changed so that it would not remind consumers of other home consoles which had fallen out of style in the States. More importantly –however – was the development of a "Lockout" circuit that prevented the running of unlicensed code, giving Nintendo control of the distribution of games on its console. This measure would prevent any future oversaturation of the video game market for the Nintendo console.

The Nintendo Entertainment System was released in the US in 1985 at a price of approximately $200 with the following technical specifications:

- 2A03 CPU – a Modified MOS 6502 8-bit processor with 5 sound channels and a DMA controller on-chip clocked at ~1.79Mhz
- 2C02 PPU – Picture Processing Unit
- Main RAM: 2KB
- Video memory: PPU contains 2 KB of tile and attribute RAM, 256 bytes of sprite position RAM ("OAM"), and 28 bytes of palette RAM

From these specifications it is possible to see where Nintendo's engineers were able to cut costs within the design of the system. The MOS 6502 was not only less expensive than other processors at the time, but it also outperformed them. Integrating the MOS 6502 and a sound generating unit into the 2A03 made it so that there would need to be fewer independent components on the NES' mainboard to manage using costly "glue logic".

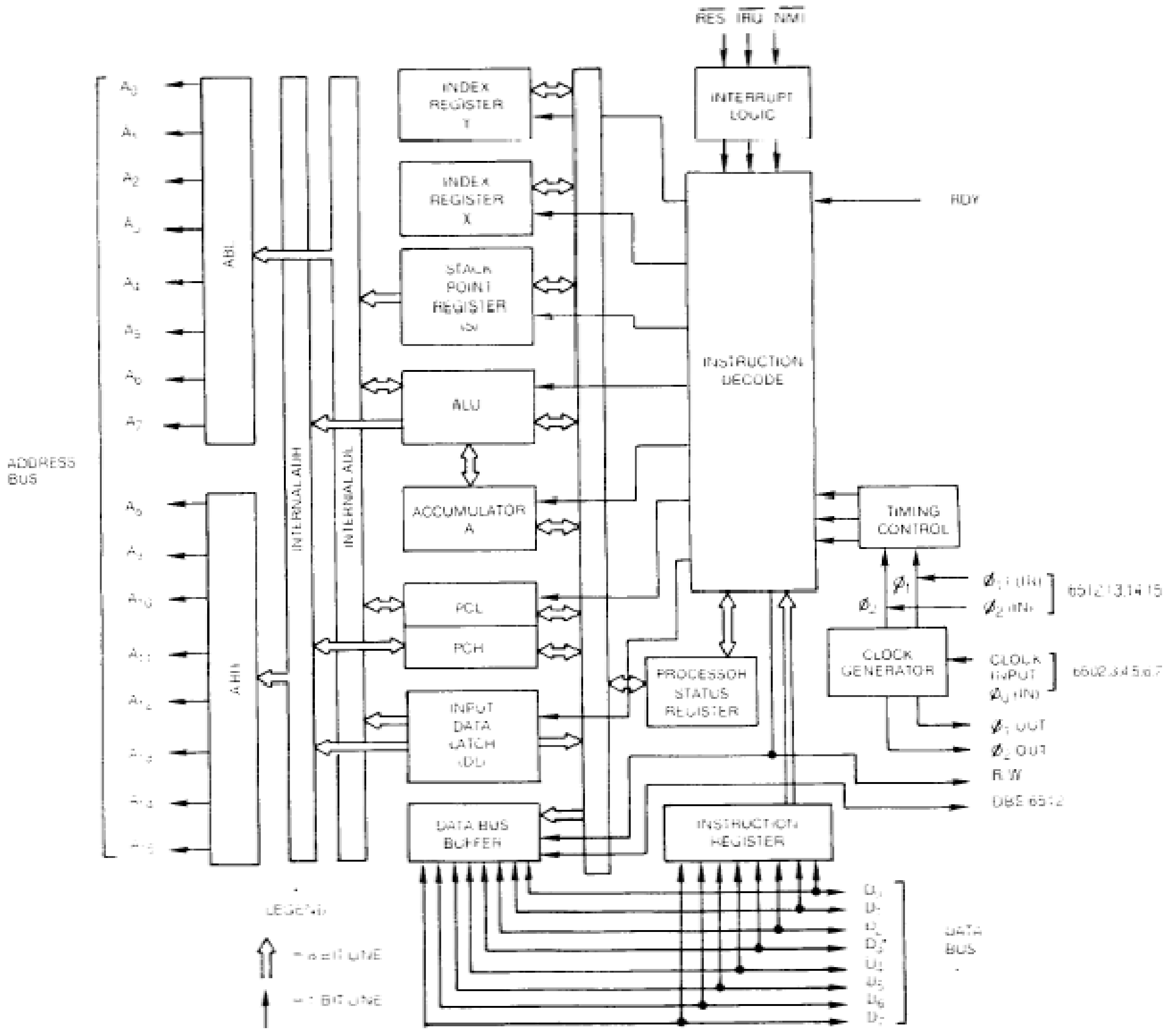


Fig 1. The MOS 6502. [3]

The MOS 6502 is an 8-bit pipelined processor with a 16-bit address bus and an 8-bit two-way data bus. It also had an 8-bit ALU as well as the following internal registers:

- Two 8-bit index registers (X and Y)
- One 8-bit stack pointer register (SP)
- One 8-bit accumulator index (A)
- One 16-bit program counter register (PC)
- One 8-bit processor status register (SR)

There were no internal data registers, since it was optimized for RAM access – which at the time, was much faster than the CPU. [2]

Each of the MOS 6502's instructions makes use of one of 13 addressing modes that effectively used each of the internal data registers on the chip. Since the 16-bit address bus is split into two 8-bit buses, these addressing modes make it easy to traverse different "pages" – defined by the high 8 bits – of memory each representing a different 8-bit address space.

There are a set of addressing modes that implicitly direct to the 0th page of memory, which in most MOS 6502 programs becomes a sort of register.

Nintendo made some modifications to the MOS 6502 in creating the 2A03. The MOS 6502's decimal mode was disabled in the 2A03. Additional features connected to the CPU through specific addresses, giving it access to the sound hardware on the 2A03, the 2C03's VRAM, the ROM and RAM on the cartridge, joysticks and other hardware.

The 2C02 was a Picture Processing Unit capable of producing 256x240 color video signals on a television set one scan-line at a time. It has a palette of 48 colors and 5 shades of grey, but only 25 would be visible on a given scan-line at a time. Additional colors could be "created" by the PPU's ability to dim the red, green, or blue value on any given scan-line. The CPU would write to the PPU's VRAM during "Blanks", periods of time during which it was guaranteed that the PPU would not be accessing its VRAM.

In the NES, it was responsible for drawing the playfield as well as the interactive objects in the screen. Additionally it was capable of detecting collisions between objects in the image and report those back to the CPU. Images were drawn on the screen in 8x8 or 8x16 blocks using colors defined in 4-color palettes all defined in cartridge ROM.
The PPU was capable of 4-directional scrolling. Advanced techniques such as parallax scrolling and split-screen scrolling could be done on the NES but it requires cycle-timed code and accesses to the PPU VRAM outside of the Blank period to implement.

III. Super NES

The wild success of the NES made it practically the only choice for home video gaming, completely marginalizing competing consoles, and Nintendo's licensing practices – while almost draconian in the degree of oversight the company had over its third-party publishes – still managed to exclusively secure the best programming talent of the day. That is until new competition arrived in the form of the Sega Genesis in 1988. While not interested in developing a new console to compete with the Genesis, it became evident that Nintendo had to act to be able to compete with the Genesis' superior technology.

Their response was the Super NES (SNES), released in 1990, essentially an evolution of the NES architecture, the SNES was considered at the time to be nearly flawless, boasting a slick new graphics chip, a Sony-designed proprietary sound chip, and several hardware modes that allowed for complex manipulations of sprites to simulate 3D graphics. Indeed, the only qualms that many had with the SNES was that it had a rather sluggish main CPU. However, when push came to shove, the SNES was able to outperform the Genesis in almost every way, and it was indicated that the SNES had outsold the Genesis by about one million units by 1996.

The controversial CPU was the MOS 65c816, a 16-bit version of the MOS 6502 found in the NES. It was able to run in an 8-bit emulation mode as well as the 16-bit native mode, making it almost 100% compatible with the 6502, but it was clocked at an unimpressive 2.68-3.58 MHz (compared to the 7.61 MHz M68000 processor running in the Genesis). Though Nintendo later found ways to get around this crater by incorporating additional chips into the game cartridges themselves, the real stars of the show were the graphics and sound co-processors, providing the power and performance that tipped the market scales in Nintendo's favor.

The graphics chip had several different hardware modes built-in that performed complex calculations automatically:

- Rotation: On the NES, in order to perform a smooth rotation on a sprite you had to construct an animation consisting of a large number of frames. The more frames used, the smoother the rotation appeared, but also the more memory required to store the costly animation. The Rotation Mode on the SNES graphics chip allowed it to rotate sprites on the fly in realtime without any need for animation frames.
- Scaling: Resizing sprites on the NES also necessitated the use of a multi-frame animation being stored in memory. The Scaling Mode did all the scaling animation in realtime, allowing games to simulate the "nearness" and "farness" of 3D distances in a 2D environment.
- Transparency: An effect not even conceptualized for the NES, the Transparency Mode could make sprites and backgrounds appear transparent or translucent, allowing for amazing new elemental effects like water, fire, smoke, and fog.
- Mode 7: The most famous of the SNES hardware modes, the evolutionary Mode 7 made it possible to scale and rotate large backgrounds to simulate 3D environmental illusions, such as flying over a large terrain in a plane.

These hardware modes, coupled with the ability to simultaneously display 256 colors from a palette of 32768, let the SNES quickly and effortlessly push advanced, 3D-like graphics to the screen that were vibrant and colorful, whereas the Sega Genesis was often defamed for its often grainy and drab-looking displays.

The SPC700 sound chip was capable of handling 44 KHz worth of sound data on 8 channels, and it had 64 Kb of dedicated memory. However, its truly ground-breaking aspect was the inclusion of the first Wavetable MIDI synthesizer ever to appear on a console, meaning it could play "sampled" sound from a generous selection of built-in MIDI instruments. Compared to the poor radio-quality sound on the Genesis, the SNES was in its own league. The chip was created by Sony exclusively for the SNES with the intention that they and Nintendo would partner to create a CD-ROM add-on for the main console. This deal eventually fell through though, leading to the development of Sony's own 32-bit, CD-based system called the Playstation.

IV. Nintendo 64

A few years passed, and gamers were expecting richer, more immersive game experiences making full use of the latest technologies available. "Multimedia" was the buzzword of the moment and companies were racing to develop the next "killer app." Developers were reaching the limits of what the Super NES could do, so a new frontier had to be explored.

That new frontier would be 3D, computer graphics have started to find their way into movies like *Jurassic Park* and consumers were hoping to have the same experience in their home video game systems. It eventually became necessary for game processors to now be able to easily compute 64-bit floating point numbers in order to handle the more complex calculations involved in drawing polygons in 3D space (instead of simulating 3D with rotating sprites).

Nintendo's new system would not be an *evolution* from the Super NES, but in fact be a totally *revolutionary* new architecture. They switched from a MOS to a MIPS processor and completely redesigned the internal architecture, and thus the Nintendo 64 was born.

The emergence of the Nintendo 64, aptly named for its new 64-bit processor, helped to usher in the era of next-generation 3D graphics; it was essentially designed to (a) facilitate the programmer's transition from creating 32-bit games to creating the new 64-bit games, and (b) be as efficient as possible while dealing with the new meatier calculations and instructions.

Nintendo's most interesting decision with regards to the N64 was the fact that it was still a cartridge based system in a time when game developers and consumers were looking to CD-ROMs as the new medium for video games. CD-ROMs promised a vast amount of storage, enabling CD quality music and full motion video to be incorporated into games. CD-ROMs were also inexpensive to produce and distribute. Cartridges are more difficult and expensive to manufacture, and the expense of silicon memory chips limited the capacity to a fraction of that available on optical media.

The reason Nintendo must have chosen to go with cartridge based media lies is the speed at which data on a cartridge is made available to the N64's CPU. There is no "loading time" with cartridges. The ability to record information in on-cartridge RAM is also not available on CD-ROMs.

The main processor, the MIPS R4300i, was based on the R4400 series which was a popular desktop PC processor at the time. The R4300 was configurable in both Big and Little Endean, and it was capable of running in both 32-bit and 64-bit modes, containing sixty-four registers that acted as either 32- or 64-bit depending on what mode was running. It had a variable clock rate, running at about 93.75 MHz at top speed but providing several lower frequencies. Also, though it had a relatively low power dissipation to begin with (about 1.8W), it had the ability to run in a reduced power mode. All of this versatility, plus the existence of a co-processor unit that could provide additional processing power, made it a prime choice for Nintendo's new gaming system. The N64 could save on resources since not every word needed to be treated as 64-bits long. It could easily support future low-speed external add-ons, and it was flexible and easy to write for from the programmer's standpoint.

Half of the registers (called General Purpose Registers or GPRs) were reserved for integer operations, while the other half (called Floating Point General Purpose Registers or FGRs) were used exclusively for the floating point operations. Most of the actual calculations were performed "off-chip" by the R4300's co-processor unit which also helped to free up resources in the main CPU (plus the coder was spared the experience of having to write a bunch of messy algorithms to do all these calculations from their end).

The co-processor was a special feature of the R4300 that made it great for gaming systems. Dubbed the Reality Co-Processor (RCP) in the N64, it was the workhorse of the R4300 handling all of the memory management and most of the audio and graphics processing. The RCP was split into two sections—the Signal Processor performed all 3D manipulations and audio functions, and the Drawing Processor pushed all the pixels and textures to the screen.

Another new and innovative feature of the N64 was that it incorporated a Unified Memory Architecture (UMA). This means that none of the memory was designated for any one specific purpose--the game programmer had full control over how much of the 4Mb of RAM would be allocated to what task. Additionally, the RAM could be expanded to 8Mb in order to provide even more flexibility and performance.

V. Conclusion

The NES, SNES, and N64 showcase the different ways Nintendo approached the design and implementation of each of their consoles from the early 80s to the late 90s. The NES was developed carefully with an eye toward American attitudes toward video games after the industry shakedown of 1983. Nintendo aimed to recapture market share lost to the Genesis console with the SNES. The Nintendo 64 was designed to push the envelope and bring a new generation of gaming to the home with immersive 3d environments.

These and other external influences – like business partnerships, third-party developers, and economic climate – affected the architectures and technologies used in the design of each console. These systems are as much a product of the times that they were designed in as the engineering that made them possible.

References

[1] "NES History." http://www.games4nintendo.com/nes/history.php *
[2] "Nintendo Entertainment System." http://en.wikipedia.org/wiki/Nintendo_Entertainment_System
[3] "The 6500 Microprocessor Family." http://6502.org/archive/datasheets/mos_6500_mpu_nov_1985.pdf
[4] Regel, Julian, "Inside Nintendo 64." http://n64.icequake.net/mirror/www.white-tower.demon.co.uk/n64/
[5] "SNES – A Brief History of the Console." http://home.uchicago.edu/~thomas/xbox-history-nintendo-snes.html
[6] Valta, Jouko, "65c816 Tech Sheet", http://members.tripod.com/FDwR/docs/65c816.txt
[7] MIPS, "R4300i Product Information." http://www.mips.com/content/PressRoom/TechLibrary/RSeriesDocs/content_html/documents/R4300i%20Product%20Information.pdf
[8] E. H. Miller, "A note on reflector arrays (Periodical style—Accepted for publication)," *IEEE Trans. Antennas Propagat.*, to be published.
[9] J. Wang, "Fundamentals of erbium-doped fiber amplifiers arrays (Periodical style—Submitted for publication)," *IEEE J. Quantum Electron.*, submitted for publication.
[10] C. J. Kaufman, Rocky Mountain Research Lab., Boulder, CO, private communication, May 1995.

*This source is particularly problematic as the same text that appears on this page appears on the open internet in several locations. There did not appear to be a definitive source for this text, but it was useful for perspective into the design of the NES.

Graphical Processing Units: An Examination of Use and Function

Michael Crayton
Mike Foss
Dan Lindquist
Kevin Tostado

Abstract

Before graphical processing units (GPUs) became commonplace on home computers, application specific processors were generally thought of as smaller market, specialty components. In the GPU, however, a processor tailored to a single purpose has become practically as ubiquitous as the CPU. Because GPUs are essentially parallel processors combined into one unit, they can outperform standard CPUs in a variety of tasks, graphics notwithstanding. Finding out how to put their specialized application to use on processor intensive problems has become a hotbed of research in the last few years, and in this paper we explore what kind of applications GPUs are being used for as well as how they function on a more fundamental level.

1. Introduction

Over the past fifteen years, there have been a remarkable amount of changes in the world of computing. During that time, the general computing power of CPUs has increased dramatically. However, GPUs have improved at an even more amazing pace. Today's GPUs have more than 25 times the performance of a 3D graphics workstation of ten years ago at less than five percent of the cost. [9] As recently as 6 years ago, the graphics card simply displayed the results as a "dumb" frame buffer, and the majority of consumer 3D graphics operations were mostly calculated on the CPU. Today, pretty much any graphics functions can be delegated to the GPU, thus freeing up the CPU to do even more calculations in order to create realistic graphics. [9]

The term GPU (Graphical Processor Unit) was coined by NVIDIA in the late 1990s to more accurately describe the advancing state of graphics hardware. [10] Initially, most GPUs were connected to the CPU via a shared PCI bus. However, with 3D animation and streaming video becoming more ubiquitous, computer architects developed the Accelerated Graphics Port (AGP), a modification of the PCI bus built to help with the use of streaming video and high-performance graphics. [11]

It is generally recognized that the existing GPUs have gone through four generations of evolution. Each generation has provided better performance and a more sophisticated programming interface, as well as driving the development of the two major 3D programming interfaces, DirectX and OpenGL. DirectX is a set of interfaces developed by Microsoft including Direct3D for 3D programming. OpenGL is an open standard for 3D programming available for most platforms. [10]

The first generation of GPUs (TNT2, Rage, Voodoo3; pre 1999) was capable of rasterizing pre-transformed triangles and applying one or two textures. This generation of GPUs completely relieved the CPU from updating individual pixels, but lacked the ability to perform transformations on the vertices of 3D objects. It also had a very limited set of math operations for computing the color of rasterized pixels. [10]

Before the second generation of GPUs (GeForce2, Radeon 7500, Savage3D; 1999-2000), fast vertex transformation was one of the key differentiations between PCs and high-end workstations. This generation expanded the math operations for combining textures and coloring pixels to include signed math operations and cube map textures, but was still fairly limited. The second generation was more configurable, but not fully programmable. [10]

The third generation of GPUs (GeForce 3, GeForce4 Ti, Xbox, Radeon 8500; 2001) provided vertex programmability rather than just offering more configurability. This generation of GPUs let the application set a sequence of instructions to process vertices. There is more configurability at the pixel-level, but the GPUs were not powerful to be considered "truly programmable." [10]

The fourth and current generation of GPUs (GeForce FX family with CineFX, Radeon 9700; 2002 to present) provide both pixel-level and vertex-level programmability. This opens up the possibility of passing off complex vertex transformation and pixel-shading operations and computations from the CPU to the GPU. DirectX 9 and various OpenGL extensions take advantage of the programmability of these GPUs. [11]

Recently, graphics hardware architectures have started to shift away from traditional fixed-function pipelines to emphasize versatility. This provides new ways of programming to reconfigure the graphics pipeline. As a result, thanks to graphics chip more and more powerful general-purpose constructs are appearing in PC machines. [12]

Currently, there is less time between releases of new GPUs than CPUs, and the latest GPUs also have more transistors than a Pentium IV. In the graphics hardware community, it is the general consensus that this trend will continue for the next few years. The continued development of the graphics processors will make GPUs much faster than CPUs for certain kinds of computations. [13]

2. Comparisons of GPUs and CPUs

There has always been a trade-off between performing operations on the GPU, which allows better performance due to its "hardwired and heavily optimized architecture," and the CPU, which allows more "flexibility due to its general and programmable nature." [14] Although GPUs have been steadily improving speed as well as the amount of tasks formerly handled by the CPU, many of the techniques that programmers would like to be able to run on the GPUs have not yet been reached due to limitations of the speed and flexibility of the current hardware. [14]

One example of how the GPU hardware has been improving is the NV35 GPU: the NV35 GPU has a transistor count of about 135 million, compared to the Xeon CPU that has only 108 million transistors, of which about two thirds of the CPU's transistors implement cache memory and not arithmetic. Other GPUs, such as the GeForce FX 5900, are primarily made up of a number of pixel pipelines implementing floating point arithmetic. Modern GPUs now provide standard IEEE floating point precision. [15]

On a program conducted to test vision algorithms, the GeForce FX 5900 completed 145 estimations each second, where as the same process, running on an AMD Athlon with clock speed of 2.0 GHz, took 3.5 seconds to complete the same number of estimations. Thus, the GPU is 3.5 times as fast. A GeForce FX 5200 card ran 41 estimations per second on this same program, which makes it roughly equivalent to the processing power of an AMD Athlon 2.0 GHz. [15]

Similar to CPU architectures, the GPU can be broken down into pipeline stages. However, whereas the CPU is a general-purpose design used to execute arbitrary programs, the GPU is architectured to process raw geometry data and to ultimately represent that information as pixels on the display monitor. [9]

3. Applications of GPUs

3.1. Overview

"Imagine that the appendix in primates had evolved. Instead of becoming a prehensile organ routinely removed from humans when it became troublesome, imagine the appendix had grown a dense cluster of complex neurons to become the seat of rational thought, leaving the brain to handle housekeeping and control functions. That scenario would not be far from what has happened to graphics processing units."[2]

The processing power of the Graphics Processing Unit is hardly utilized fully in most current applications. It gets much hype from the entertainment-driven world we live in and this is accompanied by sparked interest and funding in research fields. The GPU might surpass the CPU in possible applications very soon within our lifetime. It has already done so with power – current GPUs have millions of transistors more than CPU counterparts. GPU power is compounded with each new release from its big developers, NVIDIA and ATI. There are ways to harness this awesome power for general-use computing. Even though GPUs have limits (discussed at the end of this section), the field of General Purpose Computing using GPU architecture shows promising results.

Mathematical modeling problems require massive calculation in order to arrive at a final result. In many of these applications, current CPU power is a bottleneck for being able to rapidly progress through the research process. Using GPU architecture to speed this up has enjoyed particular success in getting results in many areas, including:

- Audio and Signal Processing
- Computational Geometry
- Image and Volume Processing
- Scientific Computing

Brief examples from each of these subsections emphasize the strengths of the GPU.

3.2. Room Acoustics Computation on Graphics Hardware [3]

GPU hardware can be used to better model the dynamics of sound in a static (or maybe even dynamic) environment. A sound source is approximated by thousands of sound rays emanating from a single point (up to 64k). Each of these rays will then bounce off of modeled surfaces up to 20 times, taking into account each wall's absorption properties. With time, the sound decays and is linked with an echogram. The point of interest, the receiver, is approximated as a sphere through which any rays that cross through this volume of space can be combined at every moment in time. These results can be used to pick the fastest path for sound between two points and can assist in designing the acoustics of rooms, duct systems, and eventually entire buildings.

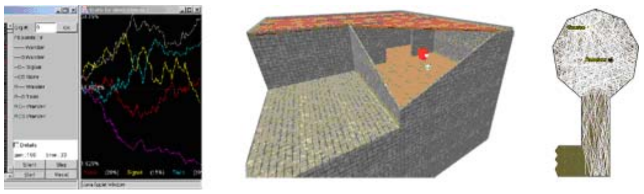


Figure 1: The GPU can power through many iterations of such a complex model as evaluating the exact replica of a sound at any point in a modeled room, given a point source of sound and absorbing materials.

3.3. Interactive Collision Detection between Complex Models Using Graphics Hardware [4]

Objects can be modeled in space with boundaries. This project attempts to harness the computational power of the GPU to detect and calculate what happens when two deformable bodies collide. An example is provided with a bunny crashing through a dragon, each of which is modeled using thousands of triangles. Pieces of interest can be followed and analyzed by these algorithms. The collision detection is performed using visibility queries at rates of up to 35 milliseconds per frame.

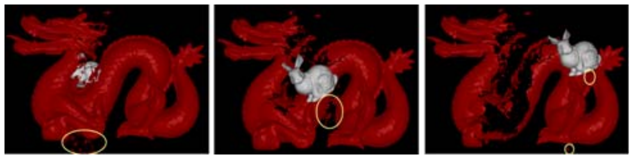


Figure 2: Complex modeling of body collisions done through visible queries requires thousands of calculations per second, per body fragment.

3.4. Interactive, GPU-Based Level Sets for 3D Brain Tumor Segmentation [5]

Currently, skilled neuroscientists take large amounts of time to set initializing spheres on many layers of MRI data as parameters in anatomical segmentation. Normal GPU architecture has been tapped with a new responsibility as a PDE level-set solver that can create models of tumors taken from MRI data. Accuracy from several tests shows an increase in volumetric representation accuracy from 82.65% by hand experts to 94.04% with unskilled users. This is accompanied with a reduction in rendering time for final structure from about 4 hours to 30 minutes. This shows additional promise as it can be used with any body part, where hand analysis requires completion by a part-specific expert.

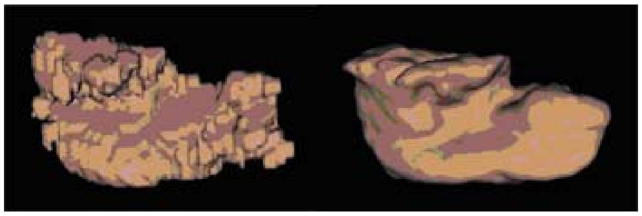


Figure 3: Modeling in biological application benefits particularly well from the GPUs ability to perform incredible amounts of parallel computation. The image (left) shows the many inaccuracies that are accentuated by the better modeled produced by a GPU (right)

3.5. GPU Cluster for High Performance Computing [6]

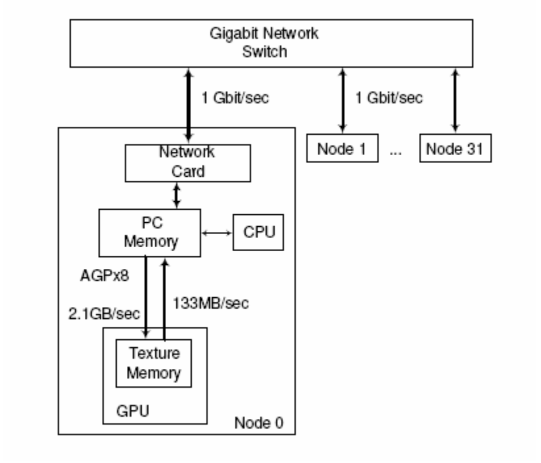
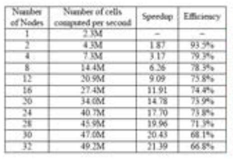


Figure 4: This shows the basic architecture of how to create a powerful cluster of GPUs.

To prove the power of group processing, this project demonstrates a machine that compiles the data for a particle dispersion flow model of a large section of New York City. This machine utilizes a cluster of 30 GPUs, run against a 30 node CPU counterpart. The machine is capable of running 49.2 million cells per second, each cell containing a Boltzmann equation for flow dynamics of particles. "These results are comparable to a supercomputer." This has obvious implications for producing many large-scale models of bio-terrorist threats as well as pollution models.



Number of Nodes	Number of cells computed per second	Speedup	Efficiency
1	2.3M	–	–
2	4.3M	1.87	93.5%
4	7.3M	3.17	79.3%
8	14.4M	6.26	78.3%
12	20.9M	9.09	75.8%
16	27.4M	11.91	74.4%
20	34.0M	14.78	73.9%
24	40.7M	17.70	73.8%
28	45.9M	19.96	71.3%
30	47.0M	20.43	68.1%
32	49.2M	21.39	66.8%

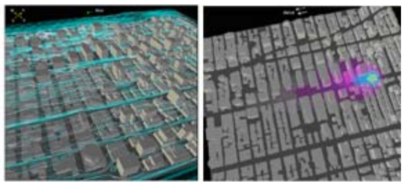


Figure 5: A New York City particle dispersion and flow model boasts the super-powers of combined GPUs. This large model approaches speedups over a similar group of CPUs close to one order of magnitude.

3.6. Limits on GPUs

From these examples, one might almost forget that GPUs exist to fuel our need for entertainment and expanding our imaginations. Over time they have become increasingly powerful to where their other applications cannot be ignored. They are moving towards playing heavier roles as coprocessors, as well as using their power to generate the AI, physics, and full video compiling and sound rendering for games. Additionally, their graphics capabilities might reach the point where "high-quality, real-time rendering may soon give way to creating full-length commercial movies."[7] Additional programmability will also give developers enhanced ability to write programs that increase "real-time rendering effects…such as bump mapping or shadows." Why are these things only occurring now, during our lifetime? As stated earlier, GPUs have many limitations.

Graphics processors are excellent at producing high-volume, parallel computations and iterations. They are not able to make many of the decisions that CPUs can. A basic list of logic functions shows the computational limits of a graphics processor:

ARL MOV MUL ADD SUB MAD ABS RCP
RCC RSQ DP3 DP4 DPH DST MIN MAX SLT

While this list shows extensive, hard-wired functions within the unit, there do not exist any forms of branching or jumping. These are essential in CPUs for even basic forms of programming logic. GPUs rudimentary locations for small programs concerning shading, but simply do not have the ability to react to user input, timing, or make any decisions in general. Other massive limitations to GPUs include:

- Inability to access data within the GPU
- Inability to give more information than yes or no on data status
- Current bottleneck is speed at which data can be transferred to GPU
- No existing operating system for GPU allocation of video memory
- Limits to the accuracy of the processor, difficult to expand floating point [8]

While it is difficult to say if more research is being done currently on how to use the GPU or how to expand it, exciting results suggest that this processor might be able to dominate the CPU in the near-future both in technological ability and profitability. As a concurrent result, mathematical modeling and rendering of valuable problems in society ranging from medical to musical applications will be sped up and enhanced, giving rise to even more cutting edge opportunities.

4. Architecture of the GPU

4.1. Graphics Pipeline

GPUs have been becoming better and better at rapid rates due to the number of transistors that can now be placed on a chip and the demand for better graphics in applications. One key element that changes with each generation of GPU is the degree to which the GPU is programmable. This has evolved from rather simple configurability to more complex programmability of the graphics pipeline. GPUs implement a massively parallel pipeline which allows for extremely efficient computation of large data sets. Figure 6 gives an overview of the programmable GPU pipeline.

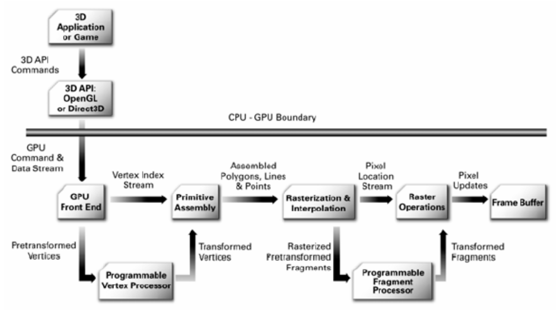


Figure 6: The programmable graphics pipeline (taken from Cg Tutorial [1])

This pipeline behaves similarly to the pipelines in CPUs that we've learned about. The graphics pipeline, however, can be thought to have several sub-pipelines within it. Modern GPUs allow for user-programming in two particular stages—vertex processing and fragment processing. Several APIs (Application Programming Interfaces) allow the programmer to develop programs that can be utilized by different GPUs which support the API. Examples of this include Microsoft's DirectX and the open-source Open GL.

4.2. The Programmable Vertex Processor

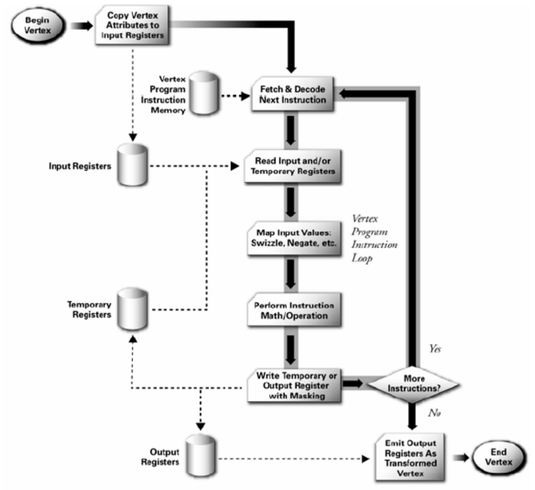

Begin Vertex
Copy Vertex Attributes to Input Registers
Vertex Program Instruction Memory
Fetch & Decode Next Instruction
Input Registers
Read Input and/or Temporary Registers
Map Input Values: Swizzle, Negate, etc.
Vertex Program Instruction Loop
Temporary Registers
Perform Instruction Math/Operation
Yes
Write Temporary or Output Register with Masking
More Instructions?
No
Output Registers
Emit Output Registers As Transformed Vertex
End Vertex

Figure 7: Programmable Vertex Processor Flow Chart (taken from Cg Tutorial [1])

The programmable vertex processor is the first stage in the graphics pipeline. This is not unlike the CPU pipeline in many ways. It has analogs to the five stages in the MIPS pipelined processor. The GPU, however, has some significant differences from the CPU in terms of architecture. Much of CPUs 'real estate' is devoted to branch prediction and caches. The GPU, on the other hand, has very little space for caches and no branch prediction. In fact, the GPU is not at all geared towards branching and conditional-type operations. GPUs are geared towards pumping as many instructions in parallel through the pipeline as possible. Indeed there are roundabout ways to force conditionals, but the hardware was not designed for this. The vertex processor performs a sequence of math operations on each vertex as quickly as possible. As GPUs become more powerful, they are able to perform even more complex mathematical operations in the hardware. This is important because higher-level math functions are key to complex algorithms that determine the lighting, shading, fog, and complex geometrical functions. Fortunately, these instructions are useful for scientific modeling and other high-powered simulations, so there is a great motivation to exploit the efficiency of the GPU here in order to run the applications more quickly than on a CPU. The bottom line is that the CPU is more suitable for running logically structured instructions in a program, and the GPU is more suitable for running a smaller set of instructions on massive amounts of data.

4.3. Programmable Fragment Processor

The fragment processor is very similar to the vertex processor, but it handles texture shading and the post-rasterization processing. After the vertices have been through the vertex shaders, assembled into primitives, and rasterized, the fragments, which are collections of flat data sent to be actually displayed onscreen, are modified in the fragment processor. This is the segment of the pipeline where things like bump maps and other textures are applied based on texture coordinate information that is included with each fragment. After each fragment has been transformed, one or several of them get transformed into a pixel which is then sent to the framebuffer to be displayed onscreen.

5. Programming the GPU

One of the main problems with past GPUs was their inflexibility in allowing the user to render objects precisely as they intended. Many had built in routines that would help the processor by taking care of graphical issues like transformation and lighting, but a user couldn’t change those routines or add his own. Most GPUs today are programmable, meaning they allow users to modify the underlying methods that transform the vertices and apply textures to change the output. Naturally, this allows programmers to tailor graphics much more closely to their specifications in real time, which in turn leads to some spectacular graphics.

GPUs, like other processors, have their own instruction sets that can be strung together to write programs. The biggest difference between a GPU’s instruction set and a CPU’s instruction set is the lack of program flow instructions. As mentioned earlier, there are no jump or branch commands, but rather a multitude of commands intended to perform quick vector transformation, like dot products and distance instructions. Most of the data that GPUs operate on is in the form of vectors with up to 4 floating point numbers. Typically, a GPU program will take a vertex (or fragment if it’s on the fragment processor) and other relevant information, like lighting, and change the properties of that vertex based on what the programmer wants.

It turns out that writing for GPUs in their assembly is somewhat unwieldy, so developers at NVIDIA and Microsoft created a language called Cg, C for graphics, to make it easier to write shaders and vertex transformations. This in turn will make development time for high impact 3D effects less and so should allow more artists to take advantage of them.

6. References

[1] R. Fernando and M. Kilgard. *The Cg Tutorial*. Addison-Wesley. USA: 2003.

[2] R. Wilson. “The Graphics Chip as Supercomputer.” *EETimes*. Dec. 13, 2004.

[3] M. Jedrzejewski. “Computation of Room Acoustics Using Programable[sic] Video Hardware.”

[4] N. Govindaraju et al. “CULLIDE: Interactive Collision Detection between Complex Models in Large Environments using Graphics Hardware.”

[5] A. Lefohn et al. “Interactive, GPU-Based Level Sets for 3D Segmentation.”

[6] Z Fan. “GPU Cluster for High Performance Computing.”

[7] M Macedonia. “The GPU Enters Computing's Mainstream.” *Computer Magazine*.

[8] Thompson Hahn and Oskin. “Using Modern Graphics Architectures for General-Purpose Computing.”

[9] Zenz, Dave. “Advances in Graphic Architecture.” Dell White Paper, September 2002.

[10] R. Fernando and M. Kilgard. The Cg Tutorial. Addison-Wesley. USA: 2003.

[11] Bruyns, Cynthia and Bryan Feldman. “Image Processing on the GPU: a Canonical Example.”

[12] Thompson, Chris et al. “Using Modern Graphics Architectures for General-Purpose Computing: A Framework and Analysis.”

[13] Krakiwsky, Sean E. et al. "Graphics Processor Unit (GPU) Acceleration of Finite-Difference Time-Domain (FTDT) Algorithm."

[14] ATI Technologies. "Smartshader™ Technology White Paper."

[15] Fung, James and Steve Mann. "Computer Vision Signal Processing on Graphics Processing Units."

Dual Core On-Chip Multiprocessors and the IBM Power5

Thomas C. Cecil

***Abstract*—As the ability to place more transistors on a chip continually increases, and while clock speed seems to have hit a wall before reaching 4 GHz, Intel and AMD have made a move away from just increasing clock speed and now have begun to focus on multicore processors. This paper investigates why single core processors are losing out to multicore processors and investigates the technology used in IBM's Power5 chip.**

***Index Terms*—microprocessors, multiprocessors, MP-on-a-chip, processor scheduling**

I. Introduction

THE time has obviously arrived when raw clock speed cannot satisfy the needs of high performance software. Intel has abandoned its push to hit the 4GHz mark and instead followed the moves of AMD to focus on development of new processors which have multiple cores. Current plans are for the release of dual-core processors in 2005, with an eight-core processor to be released in 2006 and a plan for a sixteen-core processor to be released sometime afterwards. One chip designer has even gone so far as to design a single chip with 96 different cores [3].

Multicore processors are becoming the standard in the multiprocessor market for reasons other than just pure clock speed. As the ability to put more transistors on a chip increases, chip manufacturers can add more and more complexity to their dies. Despite the ability to place nearly a billion transistors on a chip, the size of using superscalar technology on a single-processor chip is rather large. The area devoted to the logic that goes towards issue queues, register renaming, and branch prediction is so large, that instead of having such complex logic, a chip could hold four simpler processors [1]. Additionally, by some measures, superscalar processors do not offer much advantage over a regular pipelined processor. Since the everyday integer program has some limit of parallelism that can be exploited. In fact, when comparing an integer program on the 200MHz MIPS R5000 (a single issue machine) and the 200MHz MIPS R10000 (a four issue machine), the R5000 achieves a score on the SPEC95 benchmark that is about 70% of the R10000, nowhere near the 25% that would be expected if parallelism could be fully exploited [1].

Another argument for multicore processors is the current trend in software use. Now that computers are often running several processes in parallel, a multiprocessor aware operating system can utilize the multicore processors by dividing the processes between the two cores of the chip. Given the more common use of visualizations and other processor intensive processes these days, multi-core processors seem even more practical.

Significant data is not yet available for the multicore processors planned by Intel and AMD, but IBM has released details about their Power5 processor. We will take a look at how IBM managed to overcome some of the challenges presented by switching to a multicore processor.

Submitted to Mark Chang on 12/16/2004 for ENGR3410: Computer Architecture.

Thomas C. Cecil is a student in Electrical and Computer Engineering at the Franklin W. Olin College Of Engineering , Needham MA 02492. (e-mail: thomas.cecil@students.olin.edu).

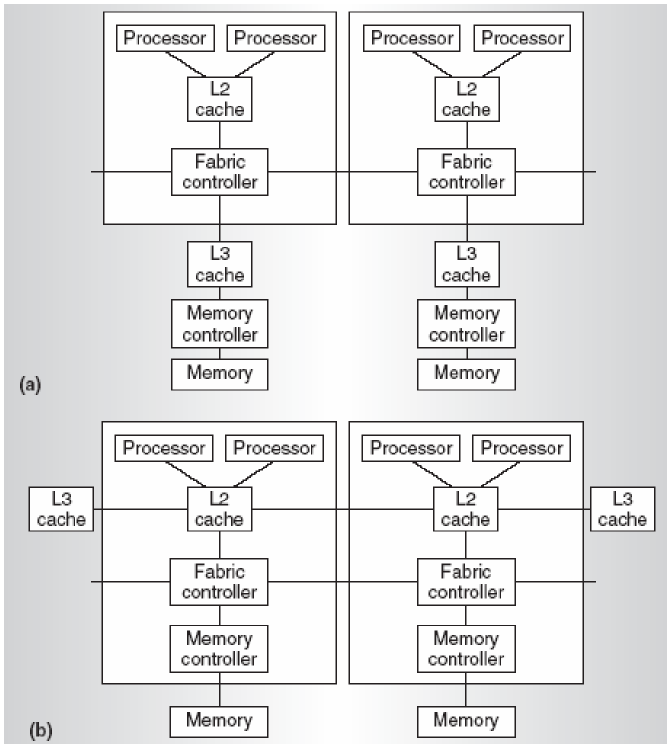


Fig. 2. Overview of the Power4 (a) and Power5 (b) chips. In the Power5, the L3 cache has been moved above the fabric controller to reduce the traffic to memory and the memory controller has been moved on chip thanks to 130nm technology, thus eliminating an extra chip and lowering the total number of chips needed for the system[4] .

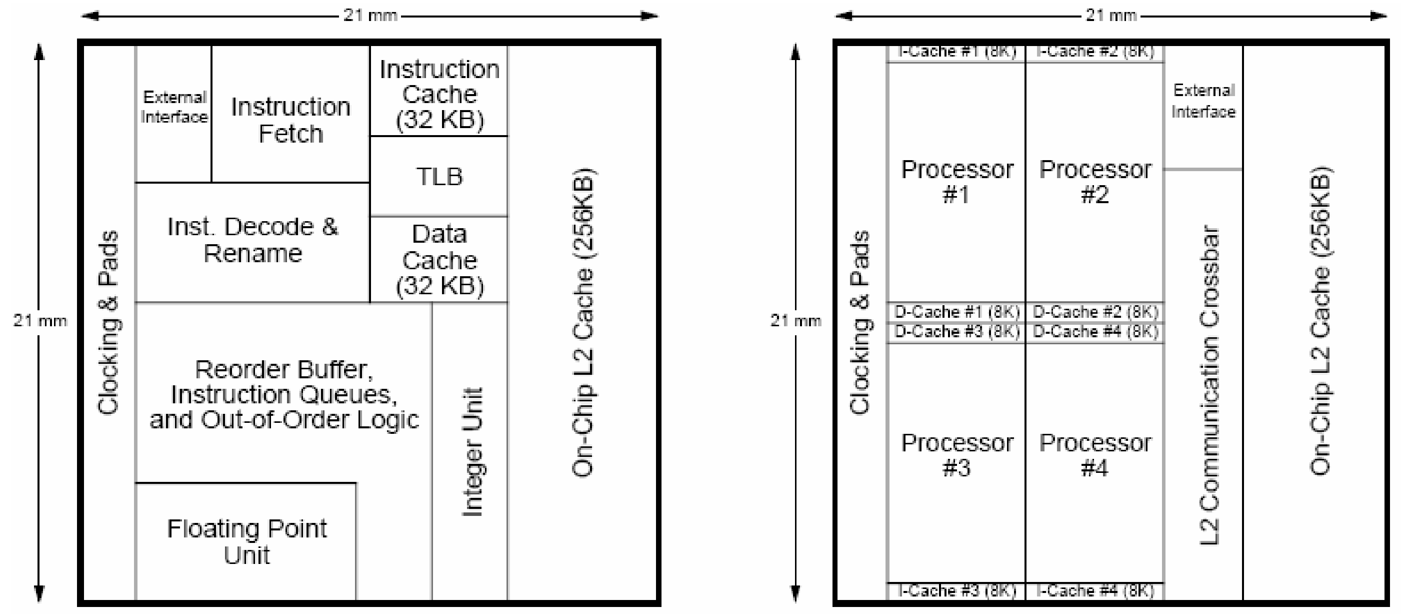


Fig. 1. Diagram of proposed restructuring of a chip from [1]. The chip floorplan on the right is a six-issue dynamic superscalar processor and the chip on the left is a four-way, single chip multiprocessor. Since the four-way processor needs less area to handle the superscalar logic, four simple processors can fit in the same area that the previous single core chip had used.

II. THE IBM POWER5

As the newest in the line of the Power4 series, the Power5 is IBM's new multicore chip. The Power5 is packed with several features that make it an excellent example of multicore processing.

In a typical processor, because the execution unit is not used much more than a quarter of the time, microprocessor architects began to use multithreading to increasing the time that the processor was actually doing work. To the operating system, it looks as if the chip actually has more than one processor, but on the inside, the processor is just dividing the task up into threads and executing it in a different fashion.

There are several different ways in which multithreading can occur. One is to use coarse grained threading in which only one thread is executing at a time. When the thread reaches a point where there is some long latency, for example a cache miss, the other thread gets some of the processor time. This turns out to be an expensive operation in itself since there is some overhead assigned to swapping threads, usually at the cost of a few cycles.

Another form of multithreading is fine grained, where each thread is executed in a round-robin style. The disadvantages here include a need for more hardware and that periods of high latency end up leaving the processor idle.

For their Power5 processor, IBM chose to use Simultaneous Multithreading, a process that allows the processor to fetch instructions from more than one thread and can dynamically schedule the instructions in a manner that allows for maximum performance. The Power5 allows each processor to use two threads, which maximizes performance and minimizes issues such as cache thrashing.

Figure 2 shows a diagram of the system structures of the IBM Power4 and Power5 chips. There is a noticeable difference between the two in that the L3 has been moved from the memory side of the fabric controller to the processor side of the fabric controller. By moving the L3 cache away from the fabric controller, traffic going to memory is reduced, which is useful in multicore processors when multiple threads might be seeking memory or L3 cache data. In addition, the L3 cache directory is on the chip, which means that cache misses can be predicted without having to deal with an off chip delay. These changes in structure allow the Power5 to support up to 64 physical processors, whereas the Power4 could only support 32.

To aid in another possibly troubling cache issue, the L2 cache is divided up into three identical slices. The data's actual address determines which of the three slices it will be cached in, and either core can access any of the three slices.

The processor core is able to operate in both simultaneous multithreaded mode and singlethreaded mode. In SMT mode, the two cores have two separate instruction fetch address registers for the two separate threads. The instruction fetch stages alternate between the two threads since there is only one instruction translation facility. When the processor switches to singlethreaded mode, it uses just on program counter and instruction fetch unit. In any given cycle, up to eight instructions can be fetched, but all fetched instructions come from the same thread.

Cache	Shared ?	Type	Size	On Chip?
L1 Data	N	4 way set assoc	32K	Y
L1 Instr	Y	2 way set assoc	64K	Y
L2	Y	10 way set assoc	1.875M	Y
L3	Y	?	36M	N

Table 1. Summary of cache resources in the Power5.

The two cores also share a trio of branch history tables and a global completion table (GCT) which is utilized to check for hazards and to keep track of up to twenty entries which are made up of groups of five instructions. A group will be committed when all instructions in that group is complete and

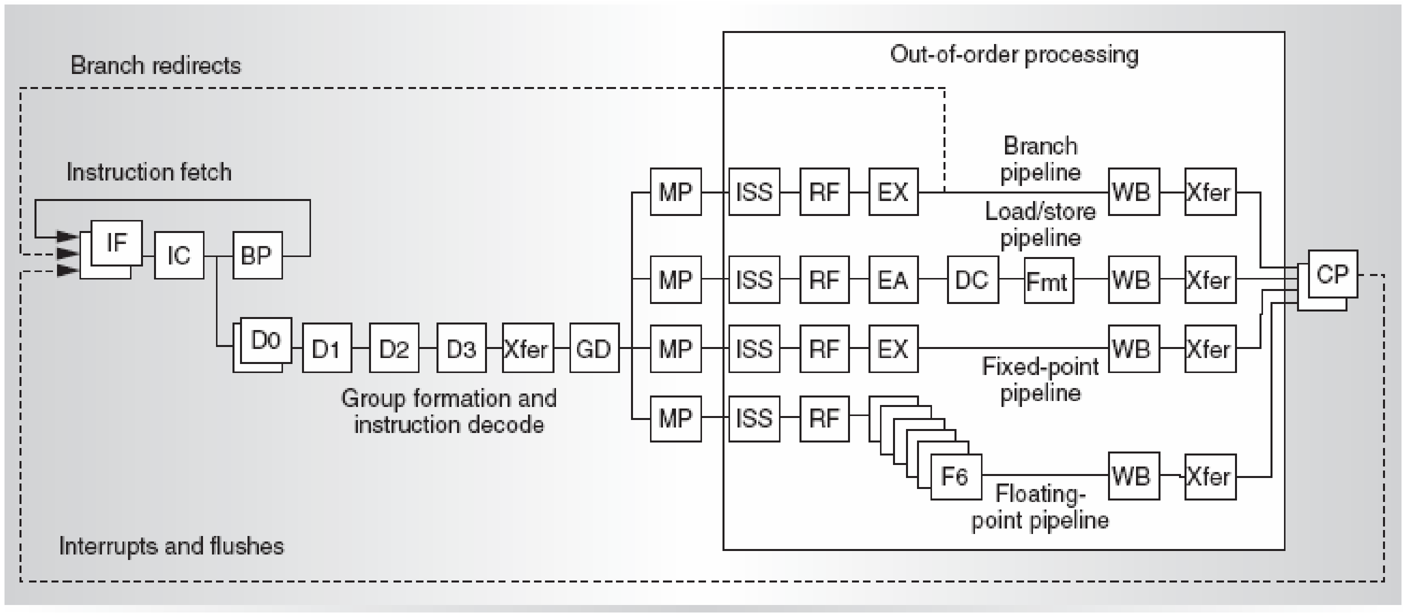


Fig. 3. The pipeline used in the Power5 Processor. (IF = instruction fetch, IC = instruction cache, BP = branch predict, D0 = decode stage 0, Xfer = transfer, GD = group dispatch, MP = mapping, ISS = instruction issue, RF = register file read, EX = execute, EA = compute address, DC = data caches, F6 = six-cycle floating-point execution pipe, Fmt = data format, WB = write back, and CP = group commit) [4].

the group is the oldest in entry in the GCT for that particular thread.

The dispatch unit can dispatch up to five instructions every cycle, and chooses which instructions to dispatch based upon thread priorities. Before being dispatched, a group must first update the GCT, resources are allocated, and hazards are detected. Afterwards, the datapath in figure 3 is followed.

It should be noted that there are 120 general purpose registers and 120 floating point registers. Those registers are shared between the two threads, and when the processor operates in singlethread mode, all registers are made available to the single thread.

The Power5 has a few additional features which make it an even more powerful chip. The first is dynamic resource balancing, which has the job of making sure that the two threads are getting through processor properly. The dynamic resource balancing tools are able to keep track of how the GCT is being used by the different threads as well as keeping track of cache misses to determine if one thread is using more than its fair share of the resources. The tool can then make a decision to throttle the offending thread by decreasing the thread priority, inhibiting the decode stage, or flushing a thread's instructions and inhibiting decode until traffic clears up.

The adjustable thread priority feature of the chip gives software the ability to decide the amount a particular thread can dominate resources. This power can be granted on any level of software, from the operating system all the way to a particular application.

By allowing custom priority levels, software that is idling, or software that is spinning while waiting for an unlock can turn down their priorities. Background tasks can take a backseat to more mission critical tasks.

The adjustable thread priority also provides power saving features. Figure 4 shows the how performance changes for various thread priority settings. Note that if the threads have priority levels (1,1), (0,1), or (1,0) then the processor enters a power saving mode to reduces power consumption while not performing important processing tasks. Additionally, if a thread is not in use, it becomes either dormant or null. When dormant, the operating system does not assign any tasks for the thread, but the thread can be activated by the active thread or by an external or decrementer interrupt. When a thread is null, the operating system is oblivious to the existence of that thread, which is useful for singlethread mode.

There are a few other power saving features of the Power5, including dynamic clock gating, which shuts off clocks in an area if that area will not be used in a specific cycle.

III. Conclusion

Although not much information is currently available regarding the current designs of multicore processors, they are quickly becoming the focus of the processor industry.

Acknowledgments

Much gratitude to IBM for actually publishing a paper about their Power5 processor. Additional thanks to the Computer Science Laboratory at Stanford University for their work on the Hydra processor. Even more gratitude is expressed to IBM and Stanford for their figures, which are used extensively both in this paper and in the accompanying presentation.

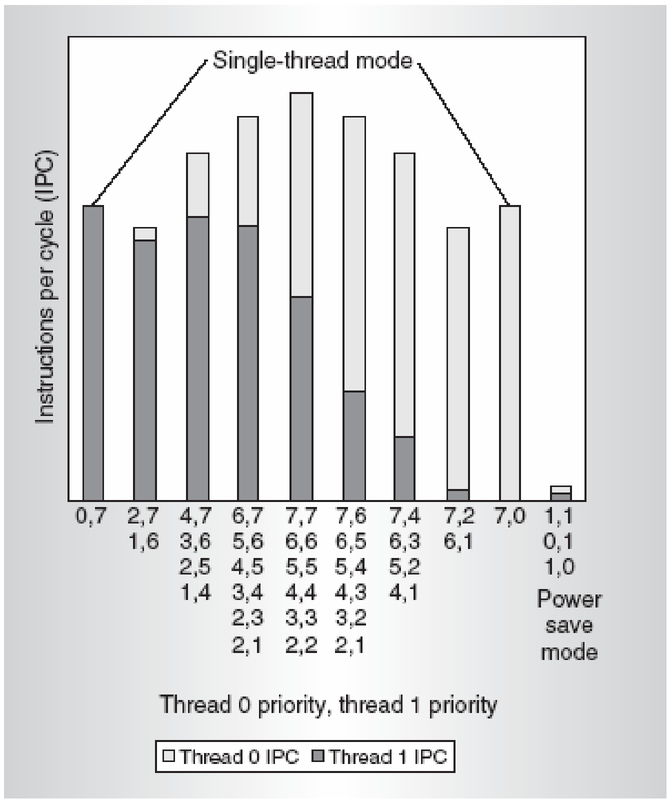


Fig. 4. Various thread priorities and resulting processor resource allocation[4].

REFERENCES

[1] K. Olukotun, K. Chang, L. Hammond, B Nayfeh, and K. Wilson, "The Case for a Single Chip Multiprocessor," *Proceedings of the 7th Int. Conf. for Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII),* pp.2-11, Cambridge, MA 1996.

[2] L. Hammond, K. Olukotun, "Considerations in the Design of Hydra: A Multiprocessor-On-A-Chip Microarchitecture. Stanford University, Stanford, CA, February 1998.

[3] Kanellos, Michael "Designer Puts 96 cores on single chip." *CNET News.com,* 10/6/04.

[4] R. Kalla, B. Sinharoy, J. Tendler, "IBM Power5 Chip: A Dual Core Multithreaded Processor," *IEEE Micro* 2004 pp.40-47.

Pong on a Xilinx Spartan3 Field Programmable Gate Array

Kate Cummings, Chris Murphy, Joy Poisel

***Abstract*— This paper describes the implementation of the game Pong on a Xilinx Spartan3 FPGA and discusses the tool flow for programming an FPGA with the Xilinx ISE Webpack. The game has the same rules as the classic arcade version of Pong, but adds a unique infrared user interface for controlling the paddles. The FPGA is programmed to serve as a simple video card, a controller for the ball and paddles, and a scorekeeper with a segmented LED display.**

I. Introduction

A. History of Pong

The history of Pong is closely interwoven with the history of video games in general. There is controversy surrounding who created the first video game. The main competitors for the title of the "Father of Video Games" are Willy Higginbotham, Steve Russell, Ralph Baer and Nolan Bushnell. Between the years 1958 and 1962 each of these men created a video game of sorts. One game used an oscilloscope and an analog computer, another used a Digital PDP-1 computer, and another used a TV. Some of the inventors knew their idea had potential and a market, others did not. Ralph Baer created a game based on the leisure sport of ping-pong. By the 1970s, there were many different video games on the market, some based off of a ping-pong concept.

B. The Rules of the Game

Pong is played by two players, using similar rules to the rules of Ping-Pong. Each player has a paddle on one side of the screen; the goal is to hit a moving ball back at the other player. Each time a player misses the ball, the other player scores a point. Once a player reaches a certain number of points, that player wins and the game is over.

C. Updating Pong

This project focused on creating an updated version of Pong. The basic rules of the game were retained, but the game was implemented on an FPGA and given a new user interface. The FPGA was programmed with modules to create a simple video controller, a scorekeeper, and a controller for the movement of the paddles and the ball on the screen. The paddles were controlled by an array of photosensors placed along the edge of the computer monitor. Players move their hand along the array of photosensors, and the paddle moves with the hand.

II. Game Development

A. VGA

The first step in the development of the modernized version of Pong was to create a simple video card. To start, a simple VGA controller was created. The VGA (Video Graphics Array) standard was introduced by IBM in 1987 and supported monitor resolutions up to a maximum of 640 pixels wide by 480 pixels tall. Since then, improved standards such as XGA (Extended Graphics Array) and UXGA (Ultra Extended Graphics Array) have been introduced, but nearly all monitors sold today still support the VGA standard. A male video connector with pinout information is pictured in figure 1.

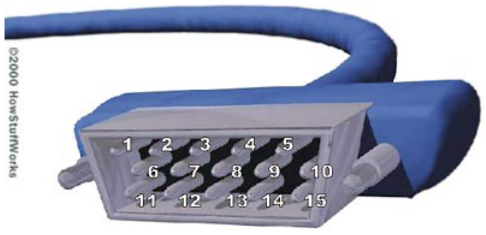


Fig. 1. Pinout information for a standard Male VGA Connector

1: Red out	6: Red Ground	11: Monitor ID 0
2: Green out	7: Green Ground	12: Monitor ID 1
3: Blue out	8: Blue Ground	13: Horizontal Sync
4: Unused	9: Unused	14: Vertical Sync
5: Ground	10: Sync Ground	15: Monitor ID 3

TABLE I
PIN CONNECTIONS FOR VGA CONNECTOR

Information about color is contained on the red, green, and blue video signals. If these were digital inputs, the monitor would be able to display a total of eight colors. Red, green, and blue could each be on or off, allowing for $2^3 = 8$ discrete states. The colors created by mixing these colors additively are shown in figure 2. Video signals from modern computers are typically analog, allowing for more colors by setting varied levels of red, green, and blue. On the monitor, the color of each pixel is determined by the magnitude of the red, green, and blue signals at the time the pixel is being drawn. A simple video card was created which was capable of producing digital signals for each of the three color outputs. It operated according to the VGA standard, at a resolution of 640 horizontal pixels by 480 vertical pixels.

VGA timing information is contained on the horizontal and vertical synchronization signals (HSync and VSync). When the end of a line is reached, the HSync signal goes high for a brief period, telling the monitor to move to the next line. The frequency at which this occurs, in kHz, is known as the horizontal refresh rate. The VSync signal goes high when the

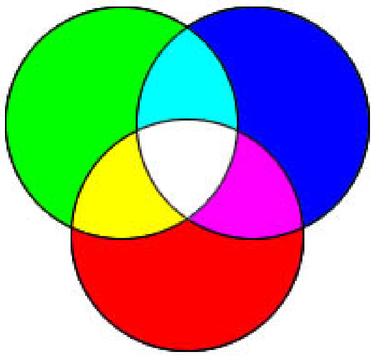


Fig. 2. Mixing Colors of Light

end of the last line is reached, and tells the monitor to go back to the top of the screen. The frequency at which the VSync signal spikes is also known as the vertical refresh rate of the monitor. Details of the synchronization signals can be seen in figure 3. If the HSync and VSync signals go high on their spikes, they are said to have a positive polarization. If they go from high to low on the spike, they have a negative polarization. The polarization of the signals depends on factors such as the refresh rate and resolution, and varies from monitor to monitor. After developing the VGA video controller, the timing was adjusted to operate at a resolution of 1024x768 pixels in accordance with the XGA standard, with a vertical refresh rate of 75Hz.

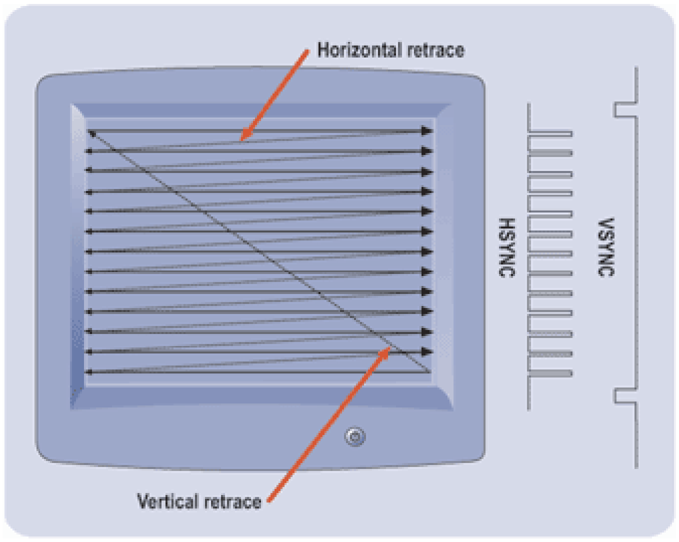


Fig. 3. Synchronization signals for VGA monitors

When a the electron beam in a Cathode Ray Tube, or CRT, screen reaches the end of a line the beam must move quickly to the start of the next line. To avoid image problems caused by the changing speed of the beam, the video signal is blanked during the retracing process. The blank part of the video signal before the HSync pulse is called the front porch, and the blank part after the pulse is called the back porch (see figure 4). This also assures that the displayed information will line up correctly on the monitor. For a CRT, as much as 25% of video signal time needs to be blanked. Digital flat panel screens do not require as long of a blanking time because they do not utilize a scanning electron beam, so blanking time standards differ for CRTs and flat panel displays.

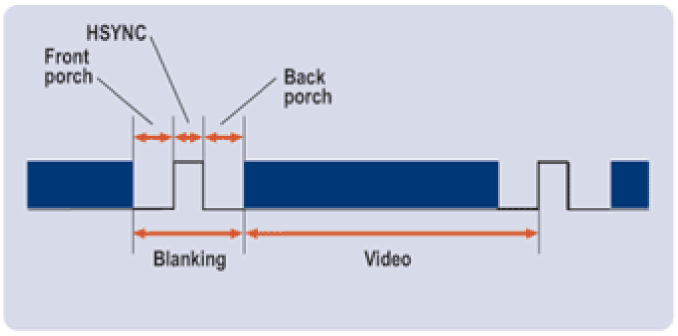


Fig. 4. The blanking period in a video color signal

III. Tool Flow

A. Programming

The first step in developing for an FPGA is to create a description of what is being implemented in the hardware. Development for this project was done using the freely available Xilinx ISE WebPack. Development can be done using a number of different formats, though all of the code for this project was written in Verilog. Xilinx WebPack also supports VHDL, and can translate schematics and state diagrams from graphical representations into hardware descriptions.

B. Synthesis

The first step in converting the description of the hardware into an actual FPGA implementation is synthesis. This process involves compiling the code to check syntax and optimizing for the design architecture being used. Code errors and warnings are detected at this step. After synthesis completes, the input file, whether Verilog, VHDL, or other, has been transformed into a description using only logical operations, flip flops, adders, and other elements.

C. Implementation

After synthesis, it is possible to assign package pins. This allows the user to assign outputs and inputs in the Verilog modules to specific pins on the physical FPGA. For example, given the following Verilog module, we would assign pins for the input ‘light’, and the output ‘button’.

```
module lightandbutton(light, button);
    input button;
    output light;
    assign button = light;
endmodule
```

On prototyping boards, some pins on the FPGA have already been assigned to functionality on the board. The Digilent board for the Spartan3 has several light emitting diodes and buttons on it, as well as a clock. Each pin on the FPGA can be assigned as an input or an output, so any pin can be used for interfacing. These assignments act as constraints in the following steps to control how the software chooses to layout the FPGA. Rules can also be added on what areas of the chip are used for laying out the design, and on specific timing constraints. For simple designs these steps are often not necessary, but are essential for larger projects or projects with high clock rates.

D. Translation and Mapping

After the synthesis process has been successfully completed, translation to a format useable by the FPGA starts. First, the results of the synthesis operation are converted into a generic Xilinx format. The next step, known as the 'mapping' step, converts this generic file format into a design which is optimized for the specific FPGA being targetted.

E. Mapping, Placing and Routing

After the specific design has been prepared, the location of each flip flop and logic element must be selected on the FPGA, and signals must be routed via the on-chip wiring. The user has the option to manually place and route the design, which involves deciding where the components will be located on the FPGA chip, as well as how the signals will be routed. If the system automatically places the elements, their locations can be tweaked after the fact. After placing and routing of the signals has completed, a file can be generated for downloading to the actual FPGA device.

F. Programming and Running

The final step in the FPGA design process is to actually configure the FPGA. This consists of generating a file in a format suitable for transferring to the FPGA, and transferring it. If a prototyping board is used, the design can also be downloaded to an on-board flash memory. The design will thus be reloaded if the board loses electrical power.

IV. The Controllers

The controllers for the Pong game consist of a linear array of infrared sensors mounted in a casing that the players put their hand into (see Figure 5). Phototransistors sense the position of the hand within the casing, and the player controls the Pong paddle by moving their hand up and down.

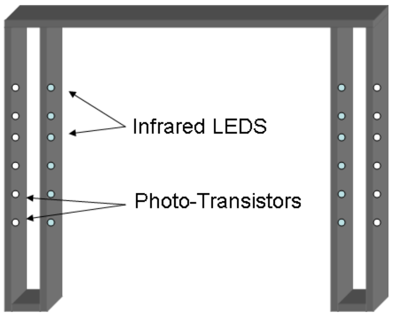


Fig. 5. Infrared pong controller

A. The Infrared Sensors

Figure 6 shows the circuit for the infrared sensors. Each sensor consists of an LED and an N-P-N phototransistor. The LED emits infrared light, which is sensed by a phototransistor located directly across from the LED. The emitter side of the transistor outputs an analog signal proportional to the amount of light that the phototransistor senses. This analog signal goes into a comparator, which compares the voltage of the signal to a reference voltage created by a voltage divider. The output of the comparator is a digital signal which indicates whether there is an object between the LED and the phototransistor. This digital signal is fed into the FPGA, which changes the VGA signal to display the correct paddle position.

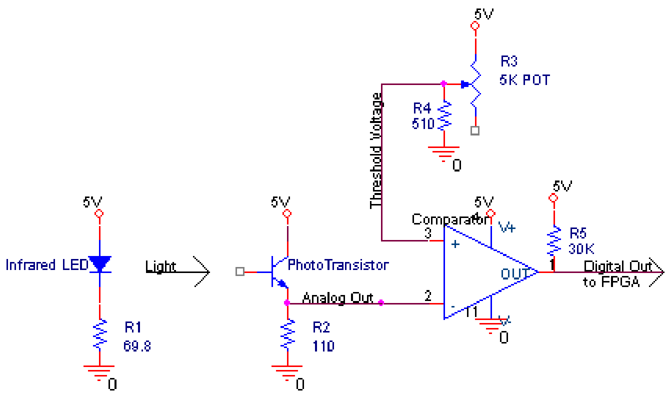


Fig. 6. Schematic for a single sensor element of the controller

B. The Array

Controllers are placed on both sides of a 19" monitor. Along the 12 vertical inches of display, there is a photosensor every 2 inches. The player moves their hand to the position along the screen where they want the Pong paddle to move. If the hand is blocking one photosensor, the center of the paddle is positioned at the photosensor. If the hand is blocking two adjacent photosensors, the center of the paddle is positioned halfway between the two photosensors. If the hand is blocking more than two photosensors or two non-adjacent photosensors the paddle does not move.

V. Conclusion

The FPGA is well suited for the task of creating the updated version of Pong due to its capacity for handling large quantities of inputs and outputs. The FPGA offers an interesting format in contrast to a standard CPU for a video game console because it is a stand alone unit and is extremely more customizable. Writing code for an FPGA is clearly more difficult than software programming, as developers must be acutely aware of the hardware being targeted. The advantages of an FPGA over a standard integrated circuit are numerous, especially for rapid prototyping purposes. The hardware is reconfigurable, allowing for easy troubleshooting and testing. Although not the best option for all applications, the FPGA proved a good platform for Pong.

Fig. 7.

References

[1] Compton, Katherine and Hauck, Scott. *Reconfiguring Computing: A Survey of Systems and Software.* http://ca.ece.olin.edu/handouts/Compton_ReconfigSurvey.pdf
[2] De Chantal, Sylvain. *Who's the real father of Videogames.* Kinox Articles. http://www.kinox.org/articles/vgfather.html
[3] Don, Steinberg. *25 Years of Video Games.* The Philadelphia Inquireer. June 22, 1997. http://www.bluedonut.com/vidgames.htm
[4] Tyson, Jeff. *How Computer Monitors Work.* HowStuffWorks.com. http://computer.howstuffworks.com/monitor.htm
[5] Wikipedia. *Field-programmable gate array.* Wikipedia. 2004. http://en.wikipedia.org/wiki/FPGA
[6] Winter, David. *Pong-Story.* 1999. http://www.pong-story.com
[7] Xilinks. *Pong-Story.* ISE Help Contents. file:///C:/Xilinx/doc/usenglish/help/iseguide/iseguide.html/

The PowerPC Architecture: A Taxonomy, History, and Analysis

Alex Dorsk, Katie Rivard, and Nick Zola

Abstract—**The PowerPC processor has been at the heart of all Apple Macintosh computers since the 1991 alliance of Apple, IBM, and Motorola. Using the RISC architecture, it has key features that distinguish it from both the MIPS and x86 style of processors. This paper outlines the PowerPC development timeline, explains the PowerPC instruction set and noteworthy functional units (vector processing, floating point, and dispatch), compares the PowerPC to the MIPS and x86 processors, and concludes with an exploration of the many other uses of the PowerPC processor beyond the Macintosh.**

I. History

A. IBM POWER and the Motorola 68K

In the early 90s, Apple Computer's machines were powered by the 68000 series of processors by Motorola. At the same time, IBM was producing the POWER (Performance Optimization With Enhanced RISC) processor, designed for mainframes and servers. IBM was looking for more applications of its RISC processor, and Apple wanted a more powerful processor for its personal computers that would also be backwards compatible with the 68K. Apple, IBM, and Motorola joined forces to create the AIM alliance, and thus, the PowerPC was born.

B. 601 (March 1994)

- Transistor Count: 2.8 million
- Die size: 121 mm^2
- Clock speed at introduction: 60-80MHz
- Cache sizes: 32KB unified L1

Key points:

- First successful RISC processor in the desktop market.
- Large, unified cache was uncommon and gave the 601 a significant edge on other processors.
- A unit in the instruction fetch phase of the pipeline called the *instruction queue* held up to 8 instructions and facillitated branch prediction and handling.
- *Branch folding* replaced "always branch" branches with their target instruction, saving two clock cycles.
- Since the *integer unit* handled memory address calculations, during long stretches of floating-point code the *floating point unit* essentially used it as a dedicated load-store unit.
- Had a *condition register* which stored various bits of metadata about the last calculation, accessible to the programmer.

C. 603 (May 1995) and 603e (Oct 1995)

The 603 came out in May of 1995. Six months later, an update was issued with a larger cache and a faster clock, making the chip more feasible for 68K emulation. Data below is for the newer chip, the 603e.

- Transistor Count: 2.6 million
- Die size: 98mm^2
- Clock speed: 100MHz
- L1 cache size: 32K split L1

Key points:

- Enter the `multiply-add` floating-point instruction, which completes both floating point operations before rounding.
- Added a *load-store unit*, possibly due to the successful partnership of the integer and floating point units in the 601.
- Added a *system unit* to handle the condition register.

D. 604 (May 1995)

The 604 was introduced in May of 1995. Just over a year later, an upgrade came out with a larger cache and a faster clock.

- Transistor Count: 3.6 million
- Die size: 197mm^2
- Clock speed: 120MHz
- L1 cache size: 32K split L1

Key points:

- Added two pipeline phases, *Dispatch* and *Complete*.
- Integer unit expanded to contain three subunits: two fast units to do simple operations like add and xor, and one slower unit to do complex operations like multiply and divide.
- Modified the *branch unit*'s prediction method from static to dynamic, adding a 512-entry by 2-bit branch history table to keep track of branches. Improving the branch unit was important to counteract the increased branch penalties of a longer pipeline.

The upgrade to the 604, the 604e, added a dedicated *Condition Register Unit* and doubled the sizes of the instruction and data cache.

E. 750, Apple G3 (November 1997)

- Transistor Count: 10.5 million
- Die size: 83mm^2
- Clock speed at introduction: 350-450MHz

- Cache sizes: 32KB L1 (instructions), 32KB L1 (data), 512KB L2

Introduced in November of 1997, the PPC 750 was very much like its grandparent, the 603e, especially in its four-stage pipeline, load-store unit, and system register unit.

The integer unit had two subunits for simple and complex operations, and the floating-point unit had been improved especially with respect to the speed of execution of complex floating-point operations. The biggest improvement for the 750 was in its branch unit, which included a 64-entry *branch target instruction cache* to store the target instruction of a branch rather than the branch address, saving valuable cycles by eliminating the fetch on branch instructions.

The 750 had a larger reorder buffer than the 603, but a smaller one than the 604. The 750 also had only half the rename registers of the 604 and fewer reservation stations – perhaps the designers expected the shorter pipeline and better branch handling would make up for the difference. An additional respect in which the 750 was lacking was that Intel and AMD already had vector capabilities in their current processors, but the PPC wouldn't support vector processing until the 7400.

F. 7400, Apple G4 (Aug 1999)

- Transistor Count: 10.5 million
- Die size: 83mm^2
- Clock speed at introduction: 350-450MHz
- Cache sizes: 32KB L1 (instructions), 32KB L1 (data), 512KB L2

The MPC7400 came out in August of 1999 essentially the same chip as the 750 with added vector processing functionality. Its 4-stage pipeline kept the clock speed of the PPC line down for quite some time, which cost the PPC credibility in the desktop market until the arrival of the newer G4s (PPC7450).

Key points:

- Floating-point unit now full double-precision; all operations have the same relatively low latency. This hadn't happened before – complex FP operations often took much longer.
- Addition of a *Vector Unit*, which operated on 128-bit chunks of data. Contained subunits to do permutations and ALU operations on vectors of data. 32 128-bit vector registers and 6 vector rename registers came along for the ride.

G. 7450, Apple G4 (Jan 2001)

- Transistor Count: 33 million
- Die size: 106mm^2
- Clock speed at introduction: 667-733MHz
- Cache sizes: 32KB L1 (instructions), 32KB L1 (data), 256KB L2, 512KB-2MB L3 cache

Key improvements:

- Pipeline deepened to 7 phases:
  1) Fetch-1
  2) Fetch-2
  3) Decode/Dispatch
  4) Issue (new) - eliminates stall in previous processors if reservation stations on a necessary execution unit was full. There are three issue queues: a general queue, a vector queue, and a floating-point queue
  5) Execute
  6) Complete
  7) Writeback/Commit
- Branch prediction uses both static and dynamic techniques, with a 2048-entry branch history table and a 128-entry branch target instruction cache storing not just the branch target instruction but the first *four* instructions after the target. Again note that the deeper pipeline required more impressive branch prediction.
- Floating point unit downsized; operations now had a 5-cycle latency, and only 4 instructions could be issued in every 5 cycles.
- Vector unit, now called the "Velocity Engine", was significantly expanded to create more specialized execution units.

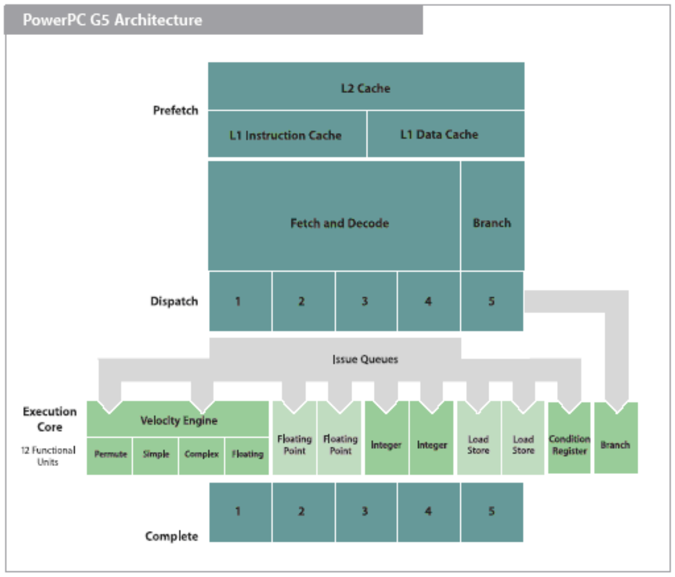


Fig. 1. Diagram of the PPC970/G5 architecture

H. 970, Apple G5 (June 2003)

Special attention will be given to the current PowerPC processor.

- Transistor Count: 58 million
- Die Size: 66mm^2
- Clock speed at introduction: 1.8-2.5 GHz
- Cache sizes: 32K L1 (data), 64K L1 (instructions), 512K L2

The most recent processor at the heart of Macintosh computers is now the PowerPC G5, introduced in June 2003 and advertised by Apple as the world's first 64-bit desktop processor. Key features include its native support for 32-bit applications, a dual-channel gigahertz frontside bus, two double-precision floating point units, a superscalar execution core supporting up to 215 in-flight instructions, group instruction dispatching, and a 128-bit vector processing unit (the "Velocity Engine").

Cleverly, the PowerPC instruction set was conceived from the beginning to accommodate both 32-bit and 64-bit code.

Thus all existing 32-bit applications run natively on the G5 without any modification or emulation required. In order to speed up communication between the processor and the memory controller, the G5 carries a dual-channel frontside bus with one link traveling into the processor and one traveling out, avoiding the time wasted by the traditional method of carrying data in only one direction at a time.

The G5 comes with twice the double-precision floating hardware of the PowerPC G4. It can complete at least two 64-bit mathematical calculations per clock cycle, and possesses a fused multiply-add feature allowing each floating-point unit to perform both an add and a multiply operation with a single instruction, as well as full precision square root calculations. The fused multiply-add instruction accelerates matrix multiplication and vector dot products, and round-off occurs only once as opposed to twice, increasing the accuracy of the result.

The PPC dispatches instructions in groups of up to five, sending them to the queues where they are further broken down for out-of-order processing. By tracking groups rather than individual instructions, the PPC is able to execute and keep organized a larger number of in-flight instructions, 215 in all when including instructions in the various fetch, decode and queue stages.

The Velocity Engine of the G5, introduced with the PowerPC G4 (7450) and still using that same set of 162 AltiVec instructions, is the famed vector processing unit of the PowerPC. The engine utilizes SIMD processing, applying a single instruction to multiple data simultaneously, ideal for computationally intensive tasks like transforming large sets of data. The Velocity Engine is used for rendering 3D images, encoding live media, and encrypting data.

The PowerPC is fabricated in IBM's facility in East Fishkill, New York, using a 90 nm process with over 58 million transistors and eight layers of copper interconnects. Each transistor is built on a layer of silicon on insulator (SOI), connected by over 400 meters of copper interconnects per processor. The G5 represents the transition from the traditional aluminum wiring on which the semiconductor industry has relied for more than 30 years, yielding a 40% gain in conductivity and faster processing operations. [9][10]

II. Compare: What makes PPC different?

A. Functional Units

1) AltiVec "Velocity Engine" Vector Processing: In the PowerPC G4 and G5, the Velocity Engine is the 128-bit vector execution unit operating in tandem with the regular integer and floating point units. It provides highly parallel internal operations for simultaneous processing of up to 128 bits of data – four 32-bit integers, eight 16-bit integers, sixteen 8-bit integers, or four 32-bit single floating-point values. The engine is designed for algorithmic intensive computations and high bandwidth data processing, all performed on-chip using 162 AltiVec instructions and single instruction, multiple data (SIMD) processing.

The vector unit is composed of thirty-two 128-bit wide vector registers. On the G5 (and first-generation G4s) there are two fully pipelined processing units: a vector permute unit and

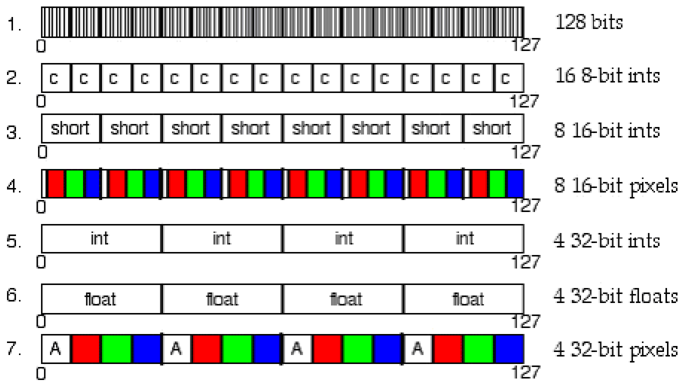


Fig. 2. Ways to split up one of the 32 128-bit vector registers.

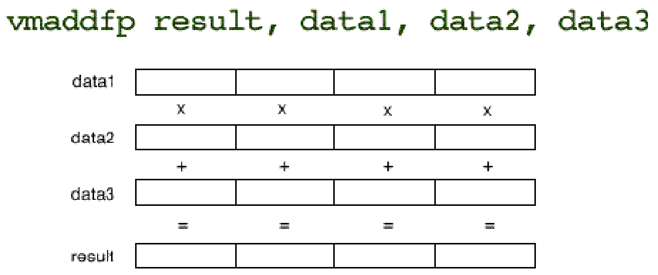


Fig. 3. Diagramming the vector multiply-add floating point operation

a vector ALU unit (for simple and complex integer and floating point operations). In the newer G4s there are four units individually dedicated to permute, simple integer, complex integer, and floating-point operations. The registers may hold either integers or single precision floating-point values, and the 128 bits may be divided as in figure 2. The value contained in each register is a vector of elements; an AltiVec instruction is performed on all elements of a vector at once. The AltiVec operations include the traditional add, subtract, multiply, and the PPC fused multiply-add, as well as some vector-specific operations like sum-across (sum all the elements in a vector), multiply-sum (multiplies and sums elements in 3 vectors), permute (fills a register with bytes from 2 other registers), and merge (merges two vectors into one).

As an example to demonstrate its capability, consider the operation `vmaddfp`, which multiplies 4 floats by 4 floats and then adds the result to 4 more floats, in a single instruction (See figure 3). This instruction is the equivalent of the fused multiply-add available in the scalar floating-point PowerPC instruction set. This is one of the ways in which the Velocity Engine is able to encapsulate a large combination of several regular scalar operations in a single instruction, exceeding the general purpose processor instruction set.

There are five general classes of AltiVec instructions: load/store, prefetch, data manipulation, arithmetic and logic, and control flow. The processor executes a vector instruction out of the same instruction stream as other PowerPC instructions. The vector instructions are RISC-style and designed to be fully pipelineable. While optimized for digital signal processing, the instructions are still general in nature.

The AltiVec instructions function as an extension to the PowerPC instruction set. The vector unit is optimized to enhance performance on dynamic, media-rich applications such as video and audio, but has also found large application in the scientific, engineering, and educational community. Some examples are the large computations necessary for modern cryptography, the dependency of many applications on the fast Fourier transform (FFTs), and the biotechnology industry's completion of the human genome project and DNA sequencing. Its design toward computationally intensive repetitive tasks also enables it to competently perform modem and telephony functions. The SETI@home (Search for Extra-Terrestrial Intelligence) project's reliance on the FFT has even led to the proposed slogan: "When ET phones, the G4 will be listening." [9][11][12]

2) Floating Point: The PowerPC 970 (G5) floating point unit consists of two parallel double-precision floating point units. Each unit has a ten-entry queue that issues instructions into a nine-stage pipeline. Floating point instructions use a set of 32 dedicated double-precision floating point registers for target and source data, and each register can also store single-precision floating point data. A status and control register records exceptions, rounding modes, result types, and whether to enable or disable exceptions.

The PowerPC's floating point unit gives good floating-point performance. By separating the floating-point operations from integer or branch operations, the overall pipeline speed is increased and more instructions can be executed simultaneously. The dedicated floating-point status and control register allows for better floating-point control, as rounding modes can be specified and intermediate results can be calculated to any accuracy before rounding. The use of two floating-point only units and 32 floating-point registers improves floating-point accuracy and overall processor speed.[5][6]

3) Dispatch: Issue Queues, Reservation Stations, OOOE: With an efficient dispatch strategy, a processor can be working on many different instructions at the same time, and keep as few functional units idle as possible. Three features the PowerPC processors have used to facilitate dispatch are Issue Queues, Reservation Stations, and Out-of-Order Execution (OOOE).

Reservation stations have been a part of the PPC architecture since the 603. They basically allow an instruction to be dispatched to its execution unit before all of its operands are available. When the instruction has collected all of its bits and is ready to be executed, it proceeds forward to the attached execution unit. Reservation stations allow instructions to be executed in the most time-efficient manner regardless of program order: if one instruction is waiting for a load from memory, and a later unrelated instruction just needs data from a register, then a multiple-entry reservation station allows the second instruction to execute as soon as its operands come out of the register file. It need not wait for the previous instruction to be executed first.

The generalized case of the reordering of instructions that reservation stations allow is called Out-of-Order Execution, abbreviated OOOE. Several other functional features of the PPC processor facillitate OOOE, including issue queues, the instruction queue, and the commit/reorder units in the final stages of the pipeline.

Issue Queues came about with the PPC 7450 or G4, presumably due to the expectation that stalls at reservation stations would cause more serious problems with the 7450's longer pipeline. If a reservation station is a waiting room, then the issue queue is the line at the door to get in, allowing further flexibility in reordering instructions but one further level removed from the execution unit itself. Issue queues in the 7450 were between 1 and 6 entries in length, and could issue 1 to 3 instructions from the bottom of the queue each cycle. [1][2]

4) Reorder Buffer: The Reorder Buffer lives in one of the last phases of the pipeline, and repairs the effects of reordering instructions so that the programmer may assume that his instructions are executed exactly in the order he wrote them. The longer the pipeline and the more instructions there are "in-flight" in the processor at a time, the more complicated the job of the reorder buffer, as there is more chance for an instruction to be executed far away from its original location in the program.

B. Instruction Set

The PowerPC implements a RISC instruction set based on IBM's POWER instruction set. There are 188 total PowerPC instructions that are grouped into integer instructions, floating point instructions, or branch/control instructions. In addition, there are 162 AltiVec instructions. Each PowerPC instruction has a fixed length and instructions are aligned as 4-byte words. There are five instruction forms (I, B, SC, D, and X) that contain various combinations of thirty-five instruction fields. Instructions are big-endian by default, but the PowerPC can support either big- or little-endian formats.

The PowerPC supports MIPS addressing modes (Register, Base, Immediate, PC-Relative, and Pseudodirect) as well as indexed and update addressing. Indexed addressing allows data stored in a register to be used as an offset. For example, the PowerPC supports the instruction `lw $t1, $a0 + $s3`. Update addressing increments a base register to the start of the next word as part of an instruction. This is useful for improving loop performance.

Integer and floating point instructions provide operations for arithmetic, comparisons, logic, rotation, shifting, loads, and stores. Each instruction can take either 64-bit or 32 bit operands. Double-precision floating-point instructions support numbers from 2.2×10^{308} to 1.8×10^{308}, and results can be rounded towards the nearest integer, towards zero, or $\pm\infty$. Branch and control instructions provide operations for branching, moving special purpose register data, memory synchronization, cache management, trapping, and system calls.

As was previously discussed, the AltiVec instructions allow for better performance for multimedia and signal processing by combining similar streams of instructions into single instructions. The vector instructions have both integer and floating-point operations similar to the non-vector integer and floating point instructions. Each vector instruction uses 128 bit registers and support 16, 8, or 4 way parallelism.

The PowerPC has notable non-standard instructions. The floating-point multiply-add instruction combines multiply and add into a single instruction. This allows for better floating-point accuracy and faster execution, as the result is rounded after both the add and multiply are complete. Additionally, the load/store multiple instruction loads or stores up to 32 words at once for more efficient memory access.

The instruction set is one of the reasons for the success of the PowerPC. The special combined floating-point instructions give better accuracy and performance for floating-point operations, and additional addressing modes improve address ranges and make loop operations more efficient. The flexibility of using either 64 or 32 bit operands for instructions allows the PowerPC to maintain legacy code without hindering progress. Special AltiVec vector instructions give the PowerPC improved performance for graphics and signal processing applications by lumping operations together for more efficient execution. [5][6][7]

C. PPC v. MIPS and x86

There are many comparisons that can be made between the PowerPC architecture and MIPS or Pentium x86 architectures. The PowerPC is similar to MIPS, as both are RISC architectures with fixed-length instructions. Notable differences between the PowerPC and MIPS include the combined floating-point multiply-add instructions, load/store multiple data instructions, and indexed/update addressing modes. The PowerPC is also far more complex, with a longer data path for superscalar pipelining with multiple processing units, out-of-order execution, and branch prediction hardware.

These differences give the PowerPC better performance than MIPS architectures. Floating-point operations are separate and can thus be more accurate without slowing down other types of operations, while more processing units and branch prediction improves pipeline speed.

The Pentium x86 architecture differs greatly from the PowerPC. Both architectures support integer, floating-point, vector, and system instructions, but Pentium x86 instructions have variable lengths and specialized operations. In addition, the x86 instruction set has had many changes due to the addition of new SSE (streaming SIMD extensions) for recent Pentiums. With regard to datapaths, the x86 pipeline has smaller, more numerous pipeline stages and more specialized execution units. While this increases clockspeed, it does not give a performance advantage over shorter, simpler pipelines.

The PowerPC 970 can have up to 215 in-flight instructions, compared to the Pentium 4's 126. Because the 970 groups instructions in sets of five, while the Pentium tracks instructions individually, the PPC processor can handle and re-order more in-flight instructions with less hardware.

The PowerPC also has a smaller die size and better power dissipation than the x86. The PowerPC 970 has an area of 61 mm^2, and dissipates 42 W. The Pentium 4 has a die size of 131 mm^2, and dissipates 68.4 W. The size and power differences are related to processor complexity. The Pentium requires more hardware for its complicated instruction set and long pipeline, and thus generates more heat and has a larger area.

The x86 style requires more hardware to implement both its legacy and its newer instructions. Longer x86 pipelines also need better branch prediction to avoid the longer pipeline penalties for incorrect branch predictions. However, as transistor density improves, extra hardware may not be as much of hindrance, and the x86 CISC architecture may have faster performance due to its efficient specialized instructions and execution units. Floating-point accuracy is greater in the PowerPC, as combined floating-point instructions with intermediate results give better accuracy. The PowerPC's longer history of 64-bit support will likely have an advantage over new Pentium 64-bit architectures, as current 32-bit programs will still run effectively on 64-bit PowerPC's. [8]

III. PPC: NOT JUST FOR MACINTOSH

The PowerPC processor as manufactured by IBM and Motorola is used in many devices beyond the Apple Macintosh. In fact there are probably more consumers using the PowerPC in other devices than there are using them in Apple computers, which make up only an estimated 3% of the personal computer market. In contrast, a May 2003 Gartner Dataquest report stated that the PowerQUICC (an extension of the PowerPC for communications applications) had captured nearly 83% of the communications processor industry. These include products such as digital subscriber line (DSL) modems, DSL access multiplexers (DSLAMs), wireless local area network (WLAN) equipment, intelligent access devices (IADs), telecom switches, wireless infrastructure, packet telephony equipment, small office/home office (SOHO) equipment, and enterprise virtual private network (VPN) routers, as well as printer, imaging and storage applications.

If communications isn't enough, since the mid-1990s and the advent of the MPC500 (an embedded PowerPC microcontroller), the 32-bit family of PowerPC microcontrollers has found its way into a number of automotive applications, including adaptive cruise control (ACC), electrohydraulic braking (EHB), direct injection, electronic valve control, hybrid and fuel cell engines, spark ignition, transmission control, high-end powertrain systems, and driver information systems (DIS). According to Motorola (which has now branched off its semiconductor sector into the subsidiary Freescale Semiconductor, Inc.), the market for PowerPC applications is only increasing. They currently boast more than 10,000 customers worldwide, and made $4.9 billion in sales in 2003.

In addition, the current Nintendo GameCube uses a PowerPC processor (as will the next one), and perhaps more incredibly the Microsoft Xbox 2 set to ship in 2006 will be PowerPC based. The personal digital video recorder TiVo, appearing in more and more homes across America, is also based on the PowerPC processor. Though the PowerPC was originally designed to meet the needs of the Macintosh and the personal computer industry, where perhaps its name is still best recognized, it is clear that the PowerPC has a wide application beyond the Apple Macintosh. [13][14][15]

REFERENCES

[1] Stokes, Jon. "PowerPC on Apple: An Architectural History" Pt I. ars technica. Online. Available Dec 2004 http://arstechnica.com/articles/paedia/cpu/ppc-1.ars/1

[2] Stokes, Jon. “PowerPC on Apple: An Architectural History” Pt II. ars technica. Online. Available Dec 2004 http://arstechnica.com/articles/paedia/cpu/ppc-2.ars/1

[3] Stokes, Jon. “Inside the IBM PowerPC 970 Pt I: Design Philosophy and Front End”. ars technica. 28 Oct 2002. Online: Available Dec 2004 http://arstechnica.com/cpu/02q2/ppc970/ppc970-1.html

[4] Stokes, Jon. “Inside the PowerPC 970 Part II: The Execution Core” ars technica. 14 May 2003. Online: Available Dec 2004 http://arstechnica.com/cpu/03q1/ppc970/ppc970-1.html

[5] “PowerPC User Instruction Set Architecture, Version 2.01”. C IBM 2003.

[6] “IBM 970FX RISC Microprocessor User’s Manual, Version 1.4”. C IBM 2003-2004. 4 Nov 2004.

[7] “PowerPC Microprocessor Family: The Programming Environments for 32-Bit Microprocessors”. C IBM 2000. 21 Feb 2000.

[8] “Intel Architecture Software Developer’s Manual Vol. 2: Instruction Set Reference”. 1997.

[9] “PowerPC G5 White Paper”. Apple Computer. January 2004.

[10] “The G5 Processor”. Apple Computer. Online: Available Dec 2004 http://www.apple.com/g5processor/

[11] Crandall, Richard E. “PowerPC G4 for Engineering, Science, and Education”. Advanced Computation Group. Oct 2000.

[12] “Velocity Engine”. Apple Computer. Online: Available Dec 2004 http://developer.apple.com/hardware/ve/

[13] “From Somerset to SoC: Scalable PowerPC©Systems-on-Chip (SoC) Platforms”. Freescale Semiconductor, Inc. 2004.

[14] “PowerPC Processors”. freescale semiconductor. Online: Available Dec 2004 http://www.freescale.com/webapp/sps/site/homepage.jsp?nodeId=018rH3bTdG

[15] “PowerPC”. Wikipedia: The Free Encyclopedia. 4 Dec 2004. Online: Available Dec 2004 http://en.wikipedia.org/wiki/Powerpc

[16] Patterson, David. *Computer Organization and Design: the Hardware/Software Interface*, 2nd ed. Morgan Kaufmann, Aug 1997.

A Brief Introduction to Quantum Computation

Michael W. Curtis

Abstract—**The field of quantum computation is just beginning to emerge as both a theoretical and experimental discipline. We wish to introduce the concept of using quantum mechanical systems to accomplish calculation more efficiently than by classical methods. This paper covers some basic quantum mechanical theory and its applications to computing with brief examples of plausible quantum procedures to replace current classical algorithms.**

I. Introduction

THE technological centerpiece of all modern digital computers is the transistor. While the semiconducting properties that allow transistors to function are manifestations of quantum-mechanical effects such as the band gap, these devices, inasmuch as their realization of computing machines is concerned, operate in the classical regime. The fundamental difference between classical and quantum computing paradigms lies in how we consider the nature of the "state" of a system.

In classical computing the fundamental state of a digital system consists of a finite number of *bits*. Bits take on any number of manifestations (transistor, capacitor, flip-flop, etc.), but when we abstract away from the hardware, we consider a bit to be in one of two mutually exclusive states, "on" or "off," "true" or "false," "1" or "0." The state of an entire computer could then be thought of as the compilation of the individual states of all the bits on the computer.

In quantum mechanics, the "state" of a system can take on a number of values, and its description represents everything we can know about the system. In particular, quantum systems can have discrete states similar to the classical analog, but they are also allowed to be in a state that represents a *quantum superposition* of any number of measurable states. What this means is that a quantum system can literally exist in a state that includes multiple measurable states simultaneously.

This opens up an entirely new realm of computing possibility, since quantum computers exploit what is known as quantum parallelism. The computer can exist as a superposition of multiple states that are of interest, and in performing operations on the quantum state, the resulting state is a superposition of all the results of that particular operation. This allows a quantum computer, for example, to theoretically examine an expansive search space in a single pass.

M. W. Curtis is with the Franklin W. Olin College of Engineering, Needham, MA 02492 USA (phone: 617-780-9302; e-mail: mike@students.olin.edu).

II. History & Developments

Quantum computing is a relatively new field. While the laws of mechanics that govern quantum computers were discovered in the 1930s, the idea to build a quantum computer is credited to the physicist Richard Feynman, who published a paper in 1982 [1] suggesting the possibility of building a quantum computer which could be used to simulate other quantum systems.

The big leap forward occurred in 1985, when David Deutsch of Oxford University published a paper [2] outlining the theoretical construction of a universal quantum computer-one that could be used as a general purpose computer, and not just as a quantum simulator. His work extended and generalized the work of Church [3] and Turning [4], who had described the theoretical basis for universal classical computers in the 1930s. Additionally, the formalization of so-called reversible Boolean logic by Toffoli [5] and others was critical to a complete description of a quantum computer (we shall explore why in a later section).

Quantum computation, while theoretically interesting, did not receive much initial attention because the quantum algorithms suggested could only outperform classical algorithms in a relatively limited set of obscure calculations. In 1994, Peter Shor from AT&T Labs developed an algorithm [6] which could be used in both discrete logarithm and prime factorization calculations. Complexity theory placed all previous classical algorithms in the class of problems know as NP-Hard. Effectively this means that the time and space required to solve them is not bounded by a polynomial of the input size. In the case of prime factorization, the best known algorithm scales exponentially with the input size.

To illustrate the scaling problem with exponential algorithms, consider that when factored by classic computers, a 129 digit number took approximately 8 months on 1600 workstations scattered around the world [7]. It would take the same set of computers approximately 800,000 years to factor a number with 250 digits. This complexity of the discrete logarithm and prime factorization problem makes them the basis of essentially all public-key cryptosystems in use today.

The Shor algorithm, however, when run on a quantum computer can be executed in a time bounded by a polynomial. What this means is that it could be used to crack any public-key cryptosystem currently in use in what is considered to be "computably reasonable" time.

The importance of such an application generated an increased interest in quantum computation, and many researchers are now attempting to build a working, general-purpose quantum computer. In August of 2000, researchers at the IBM-Almaden Research Center developed a 5-bit quantum computer programmed by radio-frequency pulses and detected by nuclear magnetic resonance (NMR) instruments[8]. The IBM team was able to implement a quantum procedure that is part of the Shor algorithm called order-finding in a single step on the quantum computer – a step that would have taken repeated operations on a classical computer.

Current public-key cryptosystems, while vulnerable to quantum algorithms, are still safe for now. Until a quantum computer can be build with at least the number of qubits as required to fit the number to be factored, the Shor algorithm is useless and the factorization problem is still NP-Hard. Theoretically, cryptographers could just increase the key sizes to keep pace with developing quantum computers–however this requires the identification of large prime numbers on which to base their keys.

III. Quantum Basics

A. Quantum States, Orthogonality, and Wavefunctions

A *quantum state* describes the properties of a quantum system. The quantum state of a particular system is expressed as a complex-valued function, known as the quantum *wavefunction*, usually denoted ψ. A description of what measurable quantities are available to us when we know the wavefunction, while interesting from a quantum physics standpoint, will not lend us much insight into how to use the states for computation, and thus we will instead suffice to say that the state of the system encompasses everything we are allowed by the laws of physics to know about the system.

The quantum wavefunction, ψ, has certain properties which are described by a complex-valued partial differential equation known as the Schrödinger Equation. We shall not discuss details, but instead suffice to say that in order to be an acceptable wavefunction, ψ must be a solution to the Schrödinger Equation. As a consequence of this, we know that if we have two wavefunctions, ψ_1 and ψ_2, such that both solve the Schrödinger Equation, then any linear combination, $\alpha\ \psi_1 + \beta\ \psi_2$, of the two solutions is also a solution, where α and β are complex coefficients. A more convenient notation for examining quantum computation instead of manipulating wavefunctions directly, instead manipulates the unique states that each wavefunction represents. We shall adopt *bra-ket notation* to manipulate states for the remainder of this paper, so it is useful to introduce this notation now. To denote the quantum state we write the symbol $\left|x\right\rangle$, called a "ket," which means the state labeled x, having the wavefunction ψ_x.

Consider two arbitrary wavefunctions, ψ_a and ψ_b. If ψ_a cannot be formed by multiplying ψ_b by any complex constant then ψ_a and ψ_b are said to be *orthogonal*. Furthermore, the states corresponding to the wavefunctions, $\left|a\right\rangle$ and $\left|b\right\rangle$ are said to be orthogonal. In fact, for any set of states (or wavefunctions), the set is said to be *mutually orthogonal* if none of the states can be formed by linear combinations of the other states in the set.

If we perform an experiment to measure the state of a system, then we find that we will always measure the system to be in a discrete set of states called *eigenstates*. It turns out that these eigenstates are the only possible states we can measure the system to be in, and each eigenstate is orthogonal to every other eigenstate. We can express every possible state the system can be in as a linear combination of eigenstates. However, even though the system can exist in a state that is any linear combination of eigenstates, as soon as we measure what state the system is in, we will always get just one eigenstate as the measured value. The probability of measuring any one particular eigenstate in a particular run of the experiment is equal to the absolute value of the square of the complex coefficient for that eigenstate in the system state. For example, say we have a system with two eigenstates: call them 0 and 1. The state of this system can be expressed as

$$\alpha\left|0\right\rangle + \beta\left|1\right\rangle \tag{1}$$

If we were to measure the system, then the probability of obtaining a 0 would be $\left|\alpha^2\right|$ and the probability of measuring a 1 would be $\left|\beta^2\right|$. Notice that this implies the relationship

$$\left|\alpha^2\right| + \left|\beta^2\right| = 1 \tag{2}$$

since the system can only be measured to be a 0 or 1. Additionally, *the act of measurement affects the state of the system, and always leaves it in an eigenstate*. What this means is that if we were to measure the system described above, the first time we measure we don't know whether we will get a 0 or a 1, but once we measure it once, every time we measure it subsequently, we will always get the same result as the first measurement. The act of measurement itself is said to have *collapsed the wavefunction* of the system to one of the eigenstate wavefunctions.

Such a system as described above is called a quantum bit, or *qubit*.

B. Qubits and Quregs

Just like in the classical analog, if we wish to store a piece information more complex than the answer to a yes-or-no question, then we arrange our qubits into quantum registers, or *quregs*. A size-n qureg is simply an ordered group of n qubits. Consider, for example, a 2-qubit qureg. We can describe its state as follows:

$$\alpha\left|00\right\rangle + \beta\left|01\right\rangle + \gamma\left|10\right\rangle + \delta\left|11\right\rangle \tag{3}$$

with

$$\left|\alpha^2\right| + \left|\beta^2\right| + \left|\gamma^2\right| + \left|\delta^2\right| = 1 \tag{4}$$

We could therefore think of describing the state of a qureg as a length 2^n vector with complex components, each representing the complex amplitude of all the possible eigenstates. This vector notion will be important when we begin to consider the transformations of the quantum state that are the essential workings of quantum computing.

C. Entanglement and Product States

$$\frac{1}{\sqrt{2}}\left|01\right\rangle + \frac{1}{\sqrt{2}}\left|10\right\rangle \tag{5}$$

Consider the state of a 2-qubit qureg in (5). Notice that each eigenstate has probability of ½ if we were to measure the system. What would happen if we measured only one of the qubits? Since the two eigenstates for the entire system are $\left|01\right\rangle$ and $\left|10\right\rangle$, we know that we should be equally likely to measure a 0 or a 1 in a single qubit. However, when we measure the bottom qubit, we immediately know that the top qubit must have the opposite eigenstate with probability 1. In measuring the bottom qubit, we forced not only it into an eigenstate, but the top bit as well. This is because as an effect of the state of the entire system the states of the individual bits were linked to one another. This is called *entanglement*.

$$\frac{1}{2}\left(\left|00\right\rangle + \left|01\right\rangle + \left|10\right\rangle + \left|11\right\rangle\right) \tag{6}$$

If, however, we have a system state such as (6), consider what happens when we measure the bottom qubit. If we measure a 0 then we leave the system in the state (7) and if we measure a 1 then we leave the system in the state (8).

$$\frac{1}{\sqrt{2}}\left(\left|00\right\rangle + \left|10\right\rangle\right) \tag{7}$$

$$\frac{1}{\sqrt{2}}\left(\left|01\right\rangle + \left|11\right\rangle\right) \tag{8}$$

In either case, we still are equally likely to measure a 0 or 1 in the top qubit. In fact, this is true whether or not we took the first measurement. This means that the bottom and top qubits are not entangled. We refer to this arrangement as a *product state*, since the overall system is simply the product of the states of the two subsystems. This is important in quantum computation, since we may want to allow qubits within quregs to be entangled with one another, but we definitely do not want quregs to be entangled with other quregs on our quantum computer since this would mean that what we do to one qureg would affect other quregs on our system.

IV. Advantages of Quantum Computing

The laws of classical mechanics explain most of the physical effects that we can observe on length scales larger than a few nanometers (on the order of atom radii). However, at length scales smaller than that, quantum mechanical effects take over and attempting to tackle these problems with classical laws produces wildly inaccurate results. The current "gold standard" in the semiconductor industry involves laying down features on semiconductors which are sized on the order of 90 nanometers (nm) [9]. As we continue to shrink circuits down further and further, quantum effects become more and more important, and we will eventually reach a point where we cannot simply assume that a transistor operates in the classical régime of having discrete "on" and "off" states. In order to continue to miniaturize, we cannot ignore quantum mechanical effects, and it may be significantly more difficult to build a classical computer on tiny length scales than to build a quantum computer.

Another interesting property of quantum mechanical processes is a consequence of the requirement of reversibility. In order to do calculations using a quantum system, the system must be manipulated in a way that is absolutely reversible, otherwise an effect known as *decoherence* occurs and information is lost to the system. This means that any transformation on the system can in theory and in practice be run in reverse ("uncomputing"). The second law of thermodynamics guarantees that any physically reversible process *dissipates no heat*. Unlike classical computers which necessarily dissipate heat as a consequence of having to move electrons through wires with non-zero resistance to charge and discharge the field effect transistors, quantum computation could in principle be performed with no loss of energy as heat.

Algorithms that have been shown to be more efficient when performed on a quantum computer share an important feature in common. They are designed to exploit quantum parallelism. Since a system can be prepared as a superposition of multiple states, when an operation is performed on the system the operation affects every state in the superposition simultaneously.

Feynman first proposed quantum computers as a useful way to simulate other quantum systems. While classical computers can be used to simulate quantum systems, when dealing with complex systems with many eigenstates, the classical computer must calculate probabilities for every possible eigenstate. The total number of eigenstates of a system with n qubits is 2^n. Suppose, for example we could represent our system with 128 qubits. This would mean there are approximately 10^{38} eigenstates of the system. A quantum computer could theoretically manipulate this system as easily as a classic computer manipulates a 128-bit number. Additionally, even for small systems a classic computer must be able to simulate randomness to handle the probabilistic nature of measurement. True randomness is in fact very hard to simulate, and thus classic computers would be limited by the quality of their random number generators.

Additionally, even though quantum computers can be operated with quantum superpositions as their system states, they certainly do not have to be. By simply preparing quregs in eigenstates, we can use them as classical registers. A quantum computer is capable of doing anything a classical computer can do, with one caveat: all operations on a quantum computer must be reversible. This is no problem if the computation we wish to execute has a unique output for every input, a so-called *one-to-one function*, however, if the computation is not one-to-one we need to make a slight modification to make the operation reversible. We can construct a *pseudo-classical operation* by having the operation pass-through the input value as well as the computational output–this allows us to be able to reverse the process and go back to the initial state.

V. Computing with Quantum Systems

Recall the notion of describing the quantum state of a system as a 2^n-dimensional vector with complex entries. The vector space corresponding to this state vector is called the Hilbert space of the system. Also recall that the sum of the probabilities of measuring each of the eigenstates is 1. The probabilities are just the squares of the entries in the state vector. This means that if ν is the state vector, with entries ν_i, then

$$\sum_i \left| \nu_i^2 \right| = 1 \tag{9}$$

This is the same thing as saying that the magnitude of the state vector is 1. Visualizing even a simple system such as a single qubit is quite difficult, since each complex number has two dimensions, a real and an imaginary. This means that visualizing the Hilbert space of a single qubit, which has two complex entries, would require four dimensions. Remember that the state vector always has a magnitude of 1, so it's only the direction of the state vector that tells us about the system. The projection of the state vector on a particular complex plane in the Hilbert space corresponds to the complex amplitude of the eigenstate associated with that plane. This means that if the state vector is ever incident with one of the planes in the Hilbert space, then the probability of measuring that eigenstate is 1; in other words, the system is in that eigenstate.

Quantum computation is accomplished by means of transformations on the state vector. Any of the allowed transformations can be expressed as a 2^n x 2^n complex valued matrix. If ν_0 is the initial state of the system, then the state of the system after the transformation is simply

$$\nu_1 = U\nu_0 \tag{10}$$

where U is the transformation matrix. The class of allowable transformation matrices are called *unitary matrices*, or *unitary transformations*. They have the property

$$U^\dagger U = I_n \tag{11}$$

where $U^\dagger$ is the conjugate transpose of U [10]. The conjugate transpose of a matrix is obtained by taking the element-by-element complex conjugate of the transpose of the matrix. I_n is the identity matrix. This means that

$$U^\dagger U \nu_0 = \nu_0 \tag{12}$$

In other words, $U^\dagger$ is the reverse operation of U. This is important because we require that all of our quantum operations be reversible. For reasons that are beyond the scope of this paper, we also know that unitary transformations will not affect the magnitude if the state vector, which is important since the state vector must always have a magnitude of 1. We can think of unitary transformations as rotations of the state vector in the Hilbert space of the system.

We can use unitary transformations to represent any operation we would like to use in our quantum computer. For example, say we have a qureg that contains some value x (this means that it is in a single eigenstate representing x). Suppose we wish to calculate the value $x + 4$. We first determine the unitary transformation for "add four," called U_{+4}, and then apply the transformation to the qureg. We then measure the contents of our qureg and obtain our answer. Now suppose that our qureg is in a superposition of the values x and y. If we apply U_{+4} to the qureg now, then the final state will be a superposition of the values $x + 4$ and $y + 4$. When we measure the contents of the register we will get either $x + 4$ or $y + 4$ with probabilities equal to the probabilities associated with x and y in the initial superposition. Also, as long as we do not measure the qureg we can always apply the inverse transformation, U_{-4} (obtained by taking the conjugate transpose of U_{+4}) to arrive back at our initial superposition of x and y. If we were to apply the inverse transformation *after* we have already measured the qureg then the result would be either x or y, depending on the result of the measurement–the quantum superposition will have disappeared.

We can build up complex transformations from more simple ones the same way that in classical computing complex Boolean functions are built from simple ones such as AND, NOT, and OR. We could imagine networks of simple *quantum gates* that represent transformations in the same way that networks of Boolean gates work in classical computers. There are a few restrictions, however. According to quantum theory, information can neither be created nor destroyed, and thus quantum circuits must be reversible. This has several consequences when attempting to construct a network of quantum gates. First of all, a quantum network must have the same number of inputs as outputs. Additionally, the "wires" that connect the various gates cannot fork or terminate. These restrictions also apply to the gates themselves: a quantum gate must be reversible and thus it must have the same number of inputs as outputs [11].

An example of a quantum gate is the one-input "quantum NOT" gate. This gate exchanges the coefficients on the two eigenstates, as seen in Fig. 1.

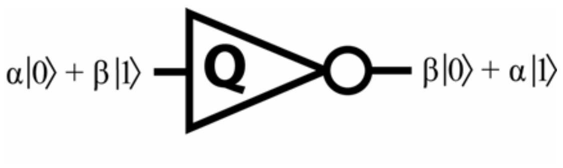


Fig. 1. The quantum NOT gate.

Notice that if the qubit is in an eigenstate (corresponding to a classical 0 or 1) then the gate works exactly as we expect a classical NOT gate to.

Just as in classical Boolean logic, there is a reversible gate that is *universal*. This means that there exists a circuit using only this gate that can calculate any Boolean function. This universal gate is called the *Toffoli gate*, and was proposed in 1980 by its namesake Tommaso Toffoli[5]. Toffoli also described a process by which any non-invertible function could be made invertible by adding additional sources (input constants) and sinks (ignored outputs) to the inputs and outputs. We shall suffice to say that using only Toffoli gates it is possible to construct any unitary transformation that operates on a qureg of any size.

VI. Quantum Computer Architecture

The basic idea of quantum computation mirrors the idea of computation on a classical computer: we prepare the system state according to the input to the computation we will perform, execute one or more unitary transformations that represent the calculation we wish to do, and then measure the system state to receive our output. Complexity arises because the output of the unitary transformations may be a superposition of more than one eigenstate, and thus when we measure the output we only get information about a single eigenstate of the output state.

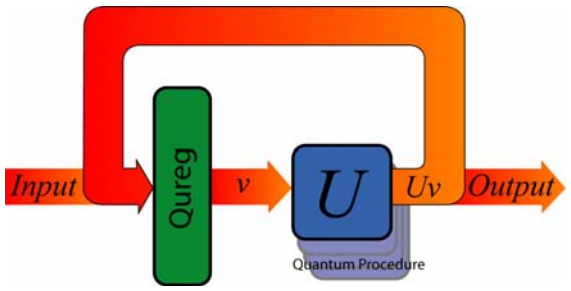
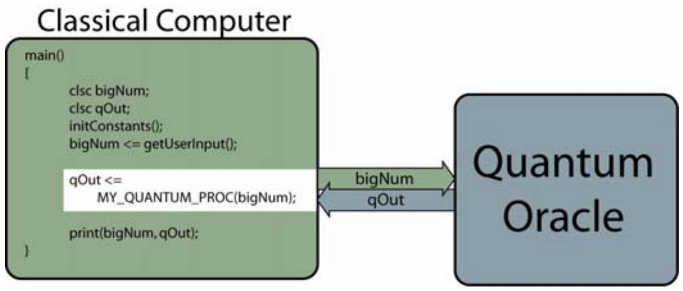


Fig. 2. A quantum feed-forward procedure consisting of a set of unitary transformations executed in sequence.

One model of how a quantum computer might work is a classical-quantum hybrid architecture which is essentially a classical computer wrapped around a quantum computer. We input a classical program into the computer which is allowed in to evoke a quantum procedure that returns a classical result (i.e. the result of measuring the quantum state at the end of the procedure). This means that the quantum procedure can return a different result each time it is evoked. Such a set-up is often termed a *quantum oracle*. An oracle is essentially a black-box procedure. It takes an input and returns an output, and we are not given access to information about how the oracle operates internally. The idea is that there are certain calculations that can be done more efficiently by the quantum oracle than by the classical computer, so the classical computer offloads those procedures to the oracle, which, when the final quantum state is measured, returns a classical result. Note that the "classical computer" could in fact just be part of a quantum computer operating in the classical regime.



```
main()
{
        clsc bigNum;
        clsc qOut;
        initConstants();
        bigNum <= getUserInput();

        qOut <=
             MY_QUANTUM_PROC(bigNum);

        print(bigNum, qOut);
}
```

Fig. 3. A block diagram of the operation of a quantum oracle in a classic program.

A more sophisticated approach would be instead of giving the programmer access to certain predefined quantum routines, a basic quantum instruction set could be defined. Each instruction would correspond to a unitary transformation on the machine state of the quantum computer, with the addition of two extra instructions: RESET and MEASURE. These two instructions are not unitary, nor are they reversible. The RESET instruction resets the qubits on the quantum computer to the "0" eigenstate, and MEASURE takes a qureg, performs the measurement, and records the results in a classical register. This design would allow programmers to use classical control structures to control quantum algorithms.

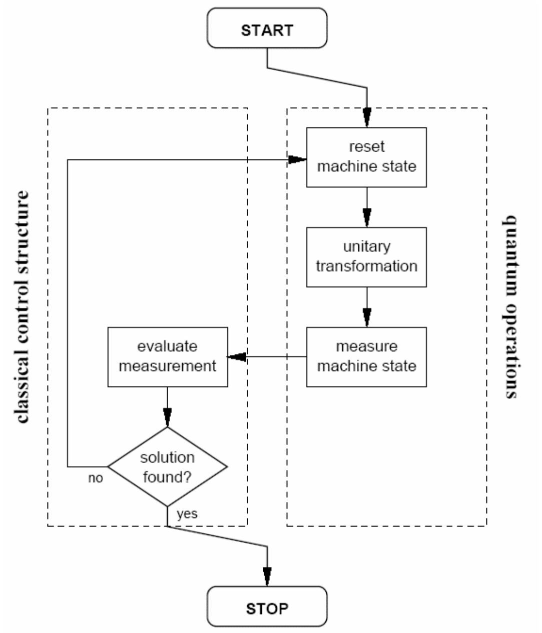


Fig. 4. A Typical non-classical algorithm[11]

Fig. 4 shows the flow control for a typical non-classical algorithm consisting of a simple search loop. Suppose we wish to search for a solution to a certain problem by exploiting quantum parallelism. We could start by initializing a qureg of sufficient size to a superposition of all the possible solutions we wish to search. Remember that with n qubits we can create a superposition of up to 2^n states. We could then define a pseudo-classic function that evaluates whether a particular state is a solution and stores this in an extra qubit. We would then have a quantum superposition of the entire search space with whether or not each state in it is a solution– the problem is then how to get any useful information out of this superposition. If we measure the qureg, we will get a state and whether or not that state is a solution, but since each state has equal quantum amplitude, this is essentially no better than picking an end state at random and classically evaluating whether or not it is a solution. If the space is large and there are few solutions then this is unlikely to be a good way to find the solution.

In order to be effective, our transformation must be able to not only evaluate whether or not something is a solution, but also affect the quantum amplitudes of the various states such that solution states become reasonably likely to be measured by our system.

The Shor algorithm [6] for prime factorization is a mixture of both quantum and classical operations. It works by using quantum parallelism to quickly evaluate the periodicity of a function which can be shown to be related to the factors of the input number. We'll follow a simple example that appears in [12]. Say we wish to factor the number 15. We first choose an arbitrary number between 0 and 15, say, 7, and define a function $f(x) = 7^x$ (mod 15). It has been shown that the period, *r,* of our function is related to the factors of 15. The first 8 values of $f(x)$ are 1, 7, 4, 13, 1, 7, 4, 13, … for x = 0, 1, 2, 3, 4, 5, 6, 7, … We can easily see that $r = 4$. r is related to the factors of 15 as the greatest common divisor of 15 and $7^{r/2} \pm 1$. In this case $7^{4/2} + 1 = 50$, and indeed the greatest common divisor of 15 and 50 is 5 (which is a prime factor of 15). The Shor algorithm works by using quantum parallelism to evaluate $f(x)$ for a large number of values in one fell swoop, and then performs a unitary transformation known as the *quantum Fourier transform*, which is related to the discrete Fourier transform. By measuring the system in this state, along with a little bit of extra mathematical manipulations the Shor algorithm is able to obtain, with high probability, the periodicity of $f(x)$. Then, using classical algorithms it can find the greatest common factor in polynomial time. The complete mathematical description of the algorithm is beyond the scope of this paper, and the reader should consult [6] for further details.

VII. Conclusion

It remains to be seen whether or not quantum computers will become a reality in the near term. However, much in the same way that classical computing had its theoretical basis firmly established by people such as Alan Turing far before an actual general purpose digital computer was built, many quantum theorists are working hard at establishing the foundations that will make quantum computing a reality once technological achievements are in place. Additionally, researchers at facilities all around the world are hard at work to develop the physical devices that can be used to implement the world's first useful quantum computer.

VIII. Suggested Further Reading

In addition to the references explicitly listed in the paper, those interested in further reading may wish to consult David Deutsch's paper [13] which describes additional computations that can be performed more efficiently on a quantum computer than a classical one, or Bennett and Shor's paper on Quantum Information theory [14].

References

[1] R. P. Feynman, "Simulating Physics with Computers," *International Journal of Theoretical Physics*, vol. 21, pp. 467, 1982.

[2] D. Deutsch, "Quantum Theory, the Church-Turing Principle and the Universal Quantum Computer," *Proceedings of the Royal Society of London, Series A*, vol. 400, pp. 97-117, 1985.

[3] A. Church, "An unsolvable problem of elementary number theory," *American Journal of Mathematics*, vol. 21, pp. 345-363, 1936.

[4] A. M. Turing, "On comutable numbers, with an application to the Entscheidungsproblem," *Proc. London Math. Soc. Ser. 2*, vol. 442, pp. 230-265, 1936.

[5] T. Toffoli, "Reversible Computing," in *Automata, Languages and Programming*, d. Bakker and v. Leeuwen, Eds. New York: Springer-Verlag, 1980, pp. 632-644.

[6] P. W. Shor, "Algorithms for Quantum Computation: Discrete Logarithms and Factoring," *Proc. 35th Annual Symposium on the Foundations of Computer Science*, pp. 124-133, 1994.

[7] S. L. Braunstein, "Quantum computation: a tutorial," vol. 2004, 1995.

[8] K. Bonsor, "How Quantum Computers Will Work," in *How Stuff Works*, vol. 2004: HSW Media Networks, 2004.

[9] M. Chang, 2004.

[10] "Unitary matrix," in *Wikipedia*: Wikimedia, 2004.

[11] B. Ömer, "Quantum Programming in QCL," in *Institute of Information Systems*. Vienna: Technical University of Vienna, 2000.

[12] A. Barenco, A. Ekert, A. Sanpera, and C. Machiavello, "A Short Introduction to Quantum Computation," in *La Recherche*, 1996.

[13] D. Deutsch and R. Jonzsa, "Rapid Solution of Problems by Quantum Computation," *Proceedings of the Royal Society of London, Series A*, vol. 439, pp. 553-558, 1992.

[14] C. H. Bennett and P. W. Shor, "Quantum Information Theory," *IEEE Transactions on Information Theory*, vol. 44, pp. 2724-2742, 1998.

Beauty and the Beast: AltiVec and MMX Vector Processing Units

Drew Harry
Olin College of Engineering
Needham, MA 02492
Email: drew.harry@students.olin.edu

Janet Tsai
Olin College of Engineering
Needham, MA 02492
Email: janet.tsai@students.olin.edu

***Abstract*— Vector processing units are becoming a popular way to handle modern multimedia applications like DVD encode/decode, and streaming video at low latency. This paper presents the general problem that vector units address, as well as two current solutions. We compare and contrast the Motorola AltiVec system and Intel's MMX architecture, including discussions of the specific instruction sets, the way they interact with the processor, exception handling, and other topics.**

I. Introduction

Computers are now heavily marketed by their performance in applications like MP3 encode/decode, DVD playback, and communication. Users expect reliable, hiccup free streaming of multimedia data, which require significant calculations at low latency. Luckily, all of these applications are similar in significant ways; the same algorithm is being applied over and over again to a large data set and the implementation must have low latency. There are also some kinds of operations that are common between algorithms that can be implemented in hardware to further increase speed. Because of this commonality, it made sense for CPU manufacturers to build modules specifically for handling these kinds of widespread high-performance needs. These modules are commonly called Vector Processing Units (VPU).

All modern CPUs from AMD, IBM, Motorola, and Intel now contain vector processing units. These functional blocks provide, in some way or another, the ability to do the same operation to many values at once. In general, these units are comprised of two major parts: registers and ALUs. Because VPUs have the same 32 bit instruction words as the main CPU, the VPU needs some way to address larger blocks of data with the same 6 bits it usually uses to address normal registers. Typically, this is done by creating a wider register file. For example, instead of the usual 32 x 32 register in MIPS processors, the AltiVec VPU has a 32 x 128 register and the Intel MMX VPU has eight 64 bit registers. The other main element of the VPU is the vector ALU, which performs similar operations to normal ALUs, but with many more computations, parsing the larger registers into smaller blocks for parallel processing.

This paper presents a comparison of two of the leading implementations of vector unit processing; Intels MMX architecture and Motorola/IBMs AltiVec technology. These implementations provide interesting insights into the design goals of each team, and the restrictions the processor architecture as a whole put on the processor designers.

II. AltiVec

AltiVec was designed by Motorola to add high performance vector processing to its PowerPC line of processors. Unlike the Pentium 3, there was plenty of space on the G3 die for more hardware, so Motorola was able to design the AltiVec unit from the ground up as an encapsulated vector processing unit. Apple Computer is the most notable buyer of chips with AltiVec technology, and all processors since the G3 have included it.[1]

In modern implementations of AltiVec, the VPU promises a throughput of one vector permute operation and one vector ALU operation per cycle, including loading from registers, executing, and writing back to registers. AltiVec maintains a separate vector register file, which must be loaded and stored to independently of the main register file.

The vector register file contains 32 registers, each 128 bits long four times as wide as the registers normally found in a 32 bit processor. Motorola considers a word to be 32 bits, thus each of these registers can be addressed as a series of 4 words, 8 half-words, or 16 bytes, as shown in figure **??**. As in a MIPS 32 bit CPU core, all the instructions operate on some number of registers (one to three) and write the result to another register. The instructions themselves specify whether to treat the 128 bit registers as 4 words, 8 half-words, or 16 bytes. This decision is handled by the compiler. The C function `vec_add()` is overloaded for all of the fundamental data types, letting the compiler handle the decomposition into the appropriate vector instruction, eg add signed bytes (`vaddsbs`), add signed half words (`vaddshs`), or add signed words (`vaddsws`). In this way, the vector unit doesnt need to keep track of how the 128 bit registers are divided into smaller pieces of data, simplifying the hardware.[2] [3] [4]

When an instruction is passed to the AltiVec scheduler, the AltiVec unit behaves like the register fetch, execution and write back units combined. In general, it takes one cycle to read from the AltiVec registers, pass the data to the vector ALU, compute the result, and write back to the register file. The AltiVec unit cant throw exceptions, guaranteeing that every operation will complete in a known number of cycles. This is important in streaming data applications, for which AltiVec is most

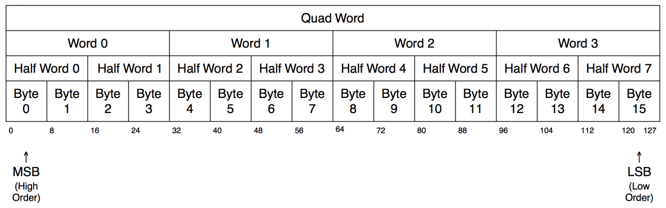


Fig. 1. The different levels of 128 bit addressing available to AltiVec instructions.[3]

useful. To make this possible, the AltiVec unit needs a default behavior for events that would otherwise trigger exceptions in a normal ALU. The most common example of such an exception is overflow. Because the AltiVec unit is frequently used in audio/video/image applications, its not important that overflows be handled in a special way its generally fine for values outside the expected range to be saturated to the maximum or minimum value. Thus any overflow event is handled by returning the appropriate saturated result. For example, attempting to add 7 (`0100`) + 9 (`0101`) in a 4 bit system would usually cause an overflow and wrap around to a negative number. In AltiVec calculations, overflows saturate, returning 15 (1111) for any operation that overflows in the positive direction and (`0000`) for operations overflowing in the negative direction.

The AltiVec instruction set provides 163 new instructions to take advantage of the AltiVec hardware. This is a deceptively large number. As discussed above, each basic operation (add, subtract, multiply, divide, etc) has at least three versions one for words, one for half words, and one for bytes. Beyond the basic arithmetic instructions, the ISA also includes instructions for operations like averaging, finding the min/max of two arguments, and bitwise logic. There is also a hodge-podge of instructions that do things like $A \cdot B + C$, 2^A, $\log_2 A$, $1/A$, rounding, etc (where A, B, C are vectors). All of these operations are handled by the main vector ALU. One instruction of these types can be issued per cycle. Simple instructions are executed in the same cycle they were issued. For more complex instructions like multiply and divide, the process is pipelined. This ensures a throughput of one, though the latency will be 3 or 4 cycles. Regardless of complexity, one vector ALU instruction can be issued per clock cycle.

There is also a class of instructions that permutes the inputs to create a new output vector composed of parts of each input vector. For example, the Vector Merge High Byte (`vmrghb`) instruction takes the 8 high bytes of input A and interleaves them with the 8 high bytes of input B, ignoring the 8 low bytes of each input (see figure II). There are also more complicated rearrangement instructions like vector permute (`vperm`). As shown in figure II, vC specifies which element of each of the two input vectors is assigned to each element of the output vector. Like the simple arithmetic vector ALU instructions, all permute operations take one cycle to execute. Also handled by the vector unit are packing and unpacking operations, in which one data type can be converted into another. This is useful when a program needs higher precision in intermediate calculations to avoid rounding errors in the results. Because these kinds of permutation operations are so common, AltiVec makes the vector permute module independent from the vector ALU module, allowing the instruction scheduler to issue both a permute and ALU instruction every cycle.[3]

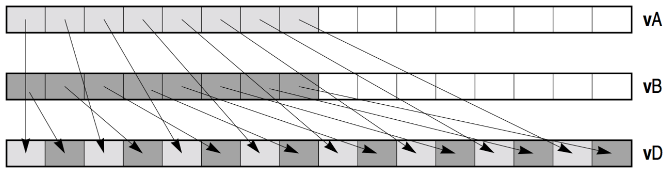


Fig. 2. Example of a specific permutation instruction, commonly used in packing and unpacking.[3]

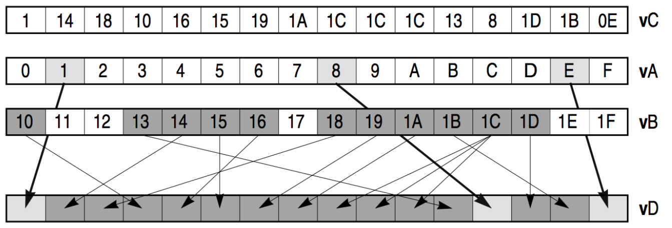

1 14 18 10 16 15 19 1A 1C 1C 1C 13 8 1D 1B 0E
vC
0 1 2 3 4 5 6 7 8 9 A B C D E F
vA
10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
vB
vD

Fig. 3. More general permutation instruction, for flexible rearrangements including substitutions, custom interleaving, sorts, etc.[3]

With the creation of these new vector instructions, it became possible to process data in new and more efficient ways. To take advantage of this, programmers must write high level code in languages like C, using specific AltiVec APIs. [6] [2] The API provides some abstraction, like overloading functions to force the compiler to pick instructions based on argument types, but vector coding requires low level design. Because of this, few general software developers actually write code for the AltiVec unit. For Apple computers, it turns out this is okay. Because of Apples broad standardization of Audio, Video and GUI APIs, as long as the OS and its main multimedia applications (Quicktime, iTunes, iDVD, CD/DVD burning, etc) take advantage of the AltiVec unit, it is still extremely useful to the end user in reducing computation time and ensuring low latency media performance. Also, given the Macs traditional success in graphic arts markets, popular applications like Photoshop and Illustrator rewrote parts of their core to take advantage of AltiVec functionality. Between OS, its core applications, and the largest 3rd party applications, AltiVec is sufficiently useful to justify its continued inclusion in Motorola and IBM processors.

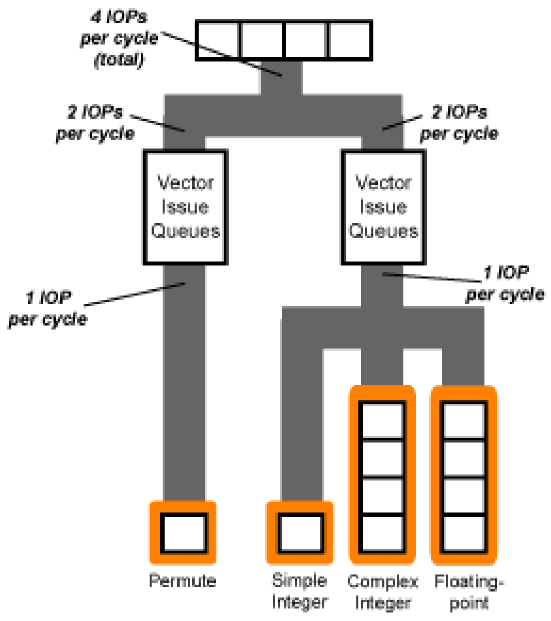


Fig. 4. High level schematic of the AltiVec pipeline. Note the pipeline depth for complex arithmatic instructions, and the ability to issue one instruction of each type per cycle.[5]

III. MMX

"MMX is rumored to stand for MultiMedia eXtensions or Multiple Math eXtension, but officially it is a meaningless [acronym] trademarked by Intel" [7]. Intel's primary goal in adding MMX to Pentium processors was "to substantially improve the performance of multimedia, communications, and emerging Internet applications."[8] The original design team met with external software developers to understand what was needed from Intels side before they began developing the MMX architecture.[9] Their findings indicated that MMX could help the most in computationally time consuming routines involving operands which were small native data types like 8 bit pixels or 16 bit audio samples.

"MMX technologys definition process was an outstanding adventure for its participants, a path with many twists and turns. It was a bottom-up process. Engineering input and managerial drive made MMX technology happen."[8]

At the same time, MMX design was bound by many constraints relating to backward compatibility. To ensure that any programs written with MMX instructions would still run on all existing machines, MMX could not introduce any new states or data registers (no new warning flags, interrupts or exceptions). Furthermore, no new control registers or condition codes could be created, either.[5] Bound by these constraints, MMX was forced to use the same physical registers that contain Floating Point (FP) data as the main MMX data registers. In doing so, the requirement that MMX instructions have to run with existing operating systems and applications was satisfied, as existing programs already know how to deal with the 80-bit FP state and no significant new architecture needed to be added.[8] Avoiding large hardware additions was especially important because the Pentium die was already extremely cramped.

Since its introduction in 1997, programs requiring MMX instructions have two versions. The program checks for the presence of MMX hardware and executes the appropriate code. All programs using MMX code must also contain non-MMX backup code, which is called if the processor doesnt have an MMX unit. While this does require programs to have duplicate instructions, because MMX is meant for small computationally intensive code elements, research estimates that the addition of MMX code causes less than a 10% growth in program size.[9]

MMX introduced four new data types to the Intel Architecture (IA). Since Intel processors already had 64-bit datapaths, the width of the MMX data types was also set at 64 bits. The four data types are different divisions of the 64 bits, so each unit of 64 bits could be 8 packed bytes, 4 packed 16-bit words, 2 packed 32-bit doublewords, or one 64-bit quadword. Full support was defined for the packed word (16 bit) data type, as based on Intels research into the outside world the main data type in many multimedia algorithms is 16 bits.[8] Furthermore, as bytes are the second most frequently used data type, 16 bits was a good higher precision backup for byte operations. Note that this is slightly different than the AltiVec vocabulary (see Figure II).

The first set of MMX instructions contained 57 new operations, all with the general format of the x86: `MMX Instruction mmreg1, mmreg2`.[10] This format is notable because it doesnt include an explicit destination register. Every instruction in the MMX set operates on the contents of `mmreg1` and `mmreg2`, and stores the result in `mmreg1`. This is necessary because of the relatively cramped register situation, but makes for more verbose code to compensate for this limitation. With the exception of multiply, all MMX instructions execute in only one cycleas one Intel programmer puts it, "we made them fast".[9] Instructions involving multiplication have a latency of three cycles, but multiplication is pipelined so it is still possible to have a throughput of one cycle per SIMD operation using MMX. It is interesting to note that the first MMX instruction set had no divide or inverse functionality and only two types of multiply.[10] The first performs four 16 bit by 16 bit multiplies and, depending on the instruction, stores the low or high order parts of the 32-bit result in the destination register. In this type of multiply, both the input operands and the results are packed 16-bit data types. The second type of multiply is multiply and add: the elements from two 16-bit operands are multiplied bitwise, generating four 32-bit results which are then added in pairs and then added again to generate the final packed 32-bit doubleword. This instruction actually performs four multiplies and two 32-bit additions, but is only one instruction. See figure III for a graphical representation of the process.[8]

Similar to the AltiVec, MMX has no overflow or underflow tags, instead, instructions specify whether the arithmetic

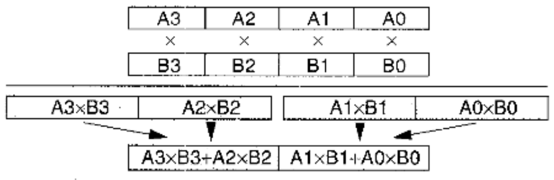


Fig. 5. Diagram of Multiply-Add instruction. The instruction is actually 6 different micro-operations; four multiplies and two additions.[8]

should be saturation or without saturation. Fourteen of the 57 MMX instructions are for adding and subtracting signed and unsigned bytes, words, and doublewords with or without saturation. A similar number of instructions are needed to provide unpacking and packing ability for each of the data types with or without saturation. As with the AltiVec, this unpacking and packing ability is commonly used to maintain precision calculations and achieve an accurate result. MMX also has instructions to shift and rotate words, double words, and quad words within each register, as well as having instructions to move doublewords and quadwords between registers. Comparisons between bytes, words, and doublewords to determine equality count for three instructions, and greater than comparisons for byte integers, word integers, and doubleword integers account for three more. Standard logical instructions are implemented bitwise, and there is one instruction which resets the tag of the FP/MMX registers to make them appear empty and thus ready for any new FP or MMX operations. [10]

While sharing the floating point registers for MMX commands enabled Intel developers to achieve their goal of backward compatibility and saving space on the die, it brought up several other issues involving conflicts between the FP data type and MMX data types. Assembly code can directly address both MMX and FP registers, as two sets of names exist for the same set of physical registers, leading to potential problems in usage.

When used to store FP values, an 80-bit FP register contains the instruction or value in the 64 lowest order bits, while bits 64-78 hold the exponent field and bit 79 contains sign information. When used as an MMX data register, the 64 low-order bits contain the instructions while the exponent field (64-78) and sign bit (79) are all set to 1. This makes any MMX value in a FP register NAN or infinity when viewed as an FP value instead of an MMX, thereby helping reduce confusion regarding using the two data types in the same physical space.[8]

Additionally, FP registers are accessed in a stack model format, whereas MMX registers are random access. These problems, combined with the argument overwriting discussed earlier create significant extra work for MMX programmers. In general usage, most applications rarely have the need to intersperse FP and MMX instructions, so the double-usage of the registers can be avoided. In this way, MMX is considered an extension of the floating-point instructions, not a new independent extension. To further alleviate this problem, MMX instructions can also use the general-purpose registers for a total of 16 registers for MMX instructions. Increasing the number of MMX registers makes register allocation much more simple and saves many instances of having to write to memory. [11]

However, the lack of dedicated registers and the necessity for backwards compatibility severely limited the MMXs usability and complicated the development process. [12] For example, in choosing the opcodes for the 57 new MMX instructions, developers had to take into account not only the opcodes in use, but also any software that may have relied on illegal opcode fault behavior of previously undefined opcodes.[13] While limiting, this type of in-depth consideration of the existing architecture did not stop Intel from moving forward in increasing the functionality of the VPU. MMXs introduction in 1997 was followed by the addition of SSE instructions which were not constrained by using the existing floating point registers.[14]

IV. Conclusions

Without the constraints of back-compatibility and limited die space, the AltiVec processor had the freedom to create a solid standalone vector processing unit. This freedom of development extended to the programmers writing code for the AltiVec, as instead of requiring programs to recognize and provide working alternatives for processors without MMX units, AltiVec programs had no such restrictions. In programming for MMX technology, software developers had to be contentious of issues like interspersing or transitioning floating point instructions with MMX instructions. MMX also restricted the programmers by providing a limited amount instructions; 57 compared to AltiVecs 163. The greater number of AltiVec instructions simply increased the flexibility in programming for AltiVec and made AltiVec easier to use.

The many disadvantages to MMX were seen clearly by the community upon its introduction in 1997. While Intel later recouped somewhat with the introduction of SSE instructions, AltiVec was successful and widely used following its introduction in the Apple branded G3 in 1999. Today, AltiVec VPUs appear in both the new G4 and G5 chips from IBM, and are also commonly used in embedded CPU applications.

Despite the inherent disadvantages of MMX, the simple fact that it added instructions and functionality to the existing IA increased the overall processor performance for most applications. The implementation of a VPU greatly speeds up computation-intense operations on small data types in both the MMX and AltiVec. The speedup is dependent on the operation that needs to be done, as MMX and AltiVec are SIMD (single instruction, multiple data) instruction sets: for instance, if multiple instructions need to be implemented on multiple data streams, adding SIMD functionality will not help.

In modern-day processors, the VPU is just one module on the die to optimize processor performance in many different instruction and data scenarios. While MMX and AltiVec were obviously imperfect in their architecture, they certainly laid the path for exciting new VPU implementations like SSE and AMD's 3DNOW! in applications as varied as the Sony/Toshiba Emotion Engine[15][16] to Sun's Tarantula.[17]

V. Future Directions

As the demand for high bandwidth multimedia increases, work on vector unit implementations has become more important. Competing architectures that provide some of the same functionality, like VLIW and superscalar architectures, have serious disadvantages. As the name implies, VLIW involves the non-trivial task of creating a new set of very long instruction words. This requires extensive rewrites of compilers, as well as a new instruction set for programmers to master. Superscalar architectures attempt to find parallelism within regular assembly code, and then take advantage of it using multiple data paths. This shifts the burden of detecting parallelism from the programmer and compiler to the hardware, creating significant power use issues. Faced with these alternatives, vector units are a fantastic alternative.[18] They operate in parallel to existing datapaths, and can be employed or ignored depending on the program leading to efficient use of power. Because VPUs have large registers, no change in the basic instruction length is necessary, making the compiler's job relatively easy. Thus it is convenient for computers and other high performance devices to rely more and more on VPUs. Recent work in the field has focused on topics like combining VPUs to provide higher performance and take advantage of data level parallelism.[19] There is also some work in writing compilers that can convert otherwise non-vector code into vector code, automatically taking advantage of vector units.[18] As graphics algorithms become standardized, some processors are specifically accelerating specific kinds of 3D vector math, lighting, and shading[20].

The number of modern applications which have large amounts of instruction level parallelism and data level parallelism continue to increase as the popularity of applications like DVD encoding and playback, MP3 encoding and playback, streaming audio and video, and high bandwidth communications increases. Since VPUs satisfy this growing need in a way that is also low latency, power efficient, and relatively easy to implement, they are becoming more prevalent in modern processors.

References

[1] D. K. et. al., “Altivec extension ot powerpc accelerates media processing,” *IEEE Micro*, vol. 20, no. 2, pp. 85–95, 2000.
[2] I. Freescale Semiconductor, *AltiVec Technology Programming Environments Manual*, 2002.
[3] M. Inc., *AltiVec Technology Programming Interface Manual*. Denver, CO: Motorola Inc., 1999.
[4] I. Freescale Semiconductor, “Altivec execution unit and instruction set overview,” 2004.
[5] J. H. Stokes, “3 1/2 simd architectures,” p. Web article, 2000.
[6] C. B. Software, “Vast/altivec code examples,” 2003.
[7] M. Wilcox, “Simd,” 10 December 2004 2004.
[8] P. A. Mittal, M. and U. Weiser, “Mmx technology architecture overview,” 1997.
[9] A. Peleg and U. Weiser, “Mmx technology extension to the intel architecture,” *IEEE Micro*, vol. 16, no. 4, pp. 42–50, 1996.
[10] I. Corp, “Intel architecture software developer's manual,” 1997.
[11] G. S. Kagan, M. and D. Lin, “Mmx microarchitecture of pentium processors with mmx technology and pentium ii microprocessors,” 1997.
[12] J. H. Stokes, “Inside the powerpc 970,” 2000.
[13] S. Thakkar and T. Huff, “The internet streaming simd extensions,” 1999.
[14] D. W. Brooks, “Vector processor,” 4 December 2004.
[15] K. A. et. al., “Vector architecture for emotion synthesis,” *IEEE Micro*, 2000.
[16] M. Technologies, “Mips technologies in the heart of playstation2 computer entertainment system,” 2000.
[17] E. R. et. al., “Tarantula: A vector extension to the alpha architecture.”
[18] C. Kozyrakis and D. Patterson, “Scalable vector processors for embedded systems,” *IEEE Micro*, 2003.
[19] M. Duranton, “Image processing by neural networks,” *IEEE Micro*, 1996.
[20] A. F. et al., “Sh4 risc multimedia microprocessor,” *IEEE Micro*, 1998.

VLIW and Superscalar Processing

Sean Munson
Olin College of Engineering
Needham, Massachusetts 02492
Email: sean.munson@students.olin.edu

Clara Cho
Olin College of Engineering
Needham, Massachusetts 02492
Email: clara.cho@students.olin.edu

***Abstract*— A number of processors add additional arithmetic logic units (ALUs) and other execution elements in order to complete multiple operations simultaneously, a process called parallelism. Superscaler processors achieve this by determining in hardware which instructions can be executed in parallel; Very Long Instruction Word implementations place this burden on the compiler.**

I. Introduction

Modern computer processors tend to have a great number of components inside, not all of which are utilized simultaneously. Maximizing the utilization of existing processor components with few hardware changes can result in significant performance increases.

A number of ways to use processor components in parallel have been invented and implemented, including pipelined, multiple core, superscalar, and very-long instruction word processing.

A. Pipelining

Pipelining groups processor components into stages in which they are used. Instructions can then be implemented over multiple pipelined stages. While this does require the addition of some hardware, the overall goal of pipelining is to find components that are rarely if ever used concurrently on the same instruction, and to separate them so that subsequent instructions can utilize them without waiting for the entire previous instruction to complete.

Pipelining has already been implemented in essentially all desktop or server processors today, and so additional methods are being pursued to continue to increase performance.

B. Multiple Processors

Many processor manufacturers are looking towards multiple core processors to gain an additional speed boost. However, this only offers advantages in limited applications [1].

C. Superscalar Processing

Superscalar implementations are capable of fetching and executing more than one instruction at a time. Hardware is used to evaluate existing instructions and decide which can be run simultaneously, which preserves backwards compatibility with code designed for preexisting processors but makes superscalar processors more complex. Most modern desktop processors have some level of superscalar operations implemented.

TABLE I
Architecture Differences Between RISC and VLIW Processors

Architecture Element	RISC	VLIW
Instruction size	One size (32 bits)	One size
Instruction contents	One simple instruction	Multiple simple, independent instructions
Hardware design	One pipline	Multiple pipelines

D. Very Long Instruction Word Processing

Very long instruction word processing (VLIW) is similar to superscalar implementations in its ability to execute multiple instructions in parallel. However, VLIW processors require the instructions to be compiled into an arrangement optimized for simultaneous execution, which reduces their hardware complexity but does require specifically compiled code in order to improve performance.

II. Superscalar and VLIW Architectures

There are key differences in instruction size, format, and length, as well as in how hardware is optimized for performance between standard RISC and VLIW architectures.

Beyond being some number of RISC instructions strung together, VLIW instructions are not very different from RISC instructions. Most often, a VLIW instruction is the length of three RISC instructions and contains slots for a branch instruction, an execution instruction, and a memory instruction, though not all implementations follow this form.

Superscalar implementations vary; some have been used for CISC architectures and some for RISC. The defining characteristic of superscalar instructions, however, is that they are not different from their parent instruction set as combination and optimization is performed at the hardware level.

VLIW bytecode differs from other code in a key way. Most code – that written for scalar and superscalar processors – is an algorithm defining what has to happen. The hardware either simply executes it in order, or in the case of superscalar implementations or pipelines with scheduling units, reorders it to a more efficient execution plan as it procedes. VLIW code, in contrast, is a very specific set of instructions for the processor that defines not only what must happen but how it must happen [10]; the processor makes no decisions about order of execution.

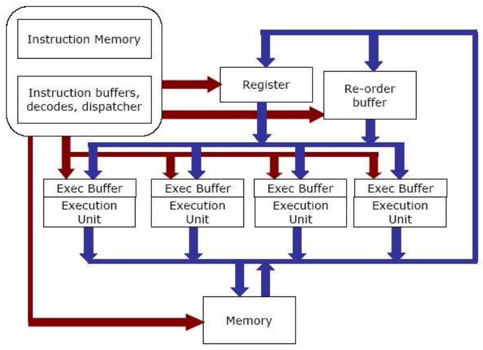


Fig. 1. RISC Superscalar Implementation

III. Implementation of Superscalar and VLIW RISC Processors

A. Superscalar

Most modern desktop computer processors support some level of superscalar operation that is compatible with existing RISC or CISC architectures. To do so, chip designers add additional pipelines as well as complicated instruction buffering and dispatching modules in order to detect statements which can be run concurrently and optimized. An overview of one such implementation is shown in figure 1.

Instructions are fetched from the memory, similar to as in other processors, but are selected in reasonably large batches. These instructions are passed to an instruction decoder and buffer, which determines the order in which the instructions will be sent to the execution units by analyzing the type of instruction and its dependencies on values from previous instructions. Reordered instructions are then dispatched to the execution units, which are a mix of ALUs for integer and floating point operations, branch control units, and memory access units. The complexity of the buffer and dispatch unit grows geometrically with the number of execution units in the processor and limits the number of execution units in superscalar processors [10]. This complexity in today's superscalar dispatch and scheduling units tends to be prohibitive for much more than six execution units [9].

Returns also diminish with increasing numbers of execution units in superscalar processors. Compilers for RISC and CISC processors produce the same instructions regardless of whether or not they are intended for superscalar or more standard (scalar) processor implementations. Since most existing processors tend to be scalar, the compilers are optimized for them and work to reduce code size. The reduction of code size is achieved largely through branch statements, with which superscalar processors do not handle very well[**?**]. In order to search for parallelism, the processor must know the outcome of the branch or at least predict it.

Because branch prediction is just that – a prediction – it is important that instructions executed based on it do not get saved in a way that cannot be undone until the prediction is complete. A large superscalar processor can execute a particularly large number of instructions speculatively, and so a reorder buffer is often added to temporarily store and use these predicted values. Once the branch prediction is confirmed, its contents are written to the register; if the branch is not confirmed the contents are simply discarded. As with the instruction buffer and dispatcher, the complexity of the reorder buffer grows geometrically with the number of execution pipes.

B. VLIW Implementation

VLIW implementation, also known as static superscalar implementation, is essentially the same as a superscalar implementation but without the complex and costly instruction dispatch units and reordering units. This is achieved by putting the responsibility on the compiler; it must output code that is already in optimized, ordered batches of instructions for the VLIW processor. Each batch is one of the "very long words."

1) Compilers for Optimization: In addition to removing the most complicated hardware associated with superscalar processors, VLIW's use of the compiler to optimize the bytecode generated results in further performance gains. A compiler can look at much larger windows of code – up to the entire program – to parallelize and optimize than a hardware dispatcher, and some particularly sophisticated optimization techniques can be applied if the source code is available.

One of these techniques is Trace Scheduling, developed by Joseph Fisher in the 1980s [2]. Trace scheduling works by analyzing the entire source code of the program – in its original programming language – then scheduling, in order of priority, the most likely path of execution, code to accept and discard the speculated values as necessary, the second most path, and so on. Predictions made at the compiler level can be more accurate as it has a picture of the entire program and can run the program a number of times to analyze its behavior.

This leads into another advantage of optimizing at compile-time. Compilation need only be performed once, which is more efficient than superscalar's optimizations that need be performed every time the program is executed. Additionally, compilers can continue to be improved and rerun on existing hardware; this is not possible for hardware-based solutions.

Some accountability for branch mispredictions is still required, but not at the level of the superscalar processor, as the compiler is aware of which statements will be speculatively executed and stores those values in temporary registers until the prediction is confirmed. It also inserts instructions to copy them over once the prediction is confirmed. For this reason, VLIW processors sometimes utilize a somewhat larger register but do not require a reorder buffer. A VLIW implementation is shown in Figure 2.

2) VLIW Drawbacks: The drawback to VLIW processors, consequently, is that code must be compiled specifically for them into order to take advantage of performance gains or sometimes to even run at all. The compiler must also be written for the specific VLIW implementation, so that instruction size is matched to the number of execution units. Some chip manufacturers have developed creative work-arounds, which will be discussed later.

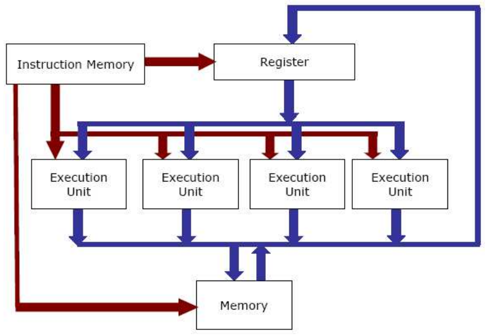


Fig. 2. RISC VLIW Implementation

Additionally, in both superscalar and VLIW operations, programs rarely use all of the execution units simultaneously, which means that some are often idle. Balance between large numbers of execution units – to generate maximum performance at times when they can all be used – and limiting the number to one where most are usually in use to save on cost and chip size is a challenge. Other components on the chip must increase in size proportionally to the number of execution units under most VLIW schemes since the VLIW instruction has a block of bits for each execution unit; adding execution units increases amount of instruction memory and the size of the instruction unit required.

Some work has been done on reducing the size of instructions to limit the additional instruction resources required. One scheme encodes (compresses) the instructions. Another includes flags in the instruction for which execution units should be used during the cycle and only sends an instruction for those units. This offers some improvement, but complexity is somewhat increased, and the maximum possible word size is increased somewhat to include the execution unit flags.

The variety of options for VLIW implementation means that the processor and compiler must be well matched. The general consensus is that they must in fact be codeveloped and targeted specifically for each other [3][4].

This, however, could rapidly become a problem for software developers if there were a number of different VLIW implementations on the market – code would have to be compiled and distributed for each, unless source code were made available for users to compile. Software manufacturers are not inclined to do so. Additionally, since VLIW compilers use knowledge of the processor at the execution-unit level, upgrading to a new processor could require recompilation of all programs. One proposal to get around this problem calls for software to be compiled first into a hardware-independent, intermediate VLIW-compatible code first and then into native code on the end user's machine [9]. A system such as the Open Software Foundation's Architecture-Neutral File Distribution Format (ANDF) would be sufficient to meet these goals but still cause significant additional work for software developers [10].

IV. Existing Superscalar and VLIW Processors

A. Superscalar

Most modern processors support some degree of superscalar operation, including both the IA-32 (x86) and the PowerPC970. Consistent with superscalar processor limitations, these top out at five or six execution units[11].

B. VLIW

1) Historical: VLIW is not a particularly new idea; the basic idea can trace its origins to work performed in the 1940s and 50s by Alan Turing and Maurice Wilkes [9]. The first computers to emerge with true VLIW processors were in the early 1980s, led by Joseph Fisher. The initiative came not from hardware thoughts but initially from compiler revolutions – his invention of Trace Scheduling. This made VLIW practical, and he left his teaching position at Yale to start MultiFlow in 1984. The first MultiFlow computers were highly-specialized, small supercomputers[10]; the specialization allowed them to sidestep the challenges that arise from having little existing code compiled for a new architecture. The processors were large and could issue up to 28 simultaneous instructions. They were also made out of a number of discrete, rather than integrated, components, which decreased their reliability while raising costs. This ultimately led to the company's failure in 1990; had today's capabilities to integrate many components on a single chip been available, the situation may have turned out differently.

In the 1990s, a number of companies licensed the technology to produce VLIW chips. Intel's i860 RISC processor could operate both in scalar and VLIW modes, but failed because compiler technology was not sufficiently advanced to be able to take advantage of the i860's architecture [**?**]. As of 1996, VLIW compilers were still very much considered to exist primarily in the research domain [9].

2) Present: The Intel-HP joint venture to produce the IA-64 Itanium is an example of VLIW processing, first released in 1999, also under the leadership of Joseph Fisher. This particular implementation takes instructions in batches of three, and has 22 execution units divided among six integer units, six multimedia units, two load units, two store units, three branch units, two extended-precision floating point units, and a single-precision floating point unit. It deals with the challenges of branch prediction by speculatively executing both possible outcomes, so long as execution units and registers are available [5].

However, because relatively few applications have been recompiled for the IA-64 – essentially none at its launch – Intel chose to also include superscalar capabilities to maintain high performance on existing code. IA-32-compatible code first enters an IA-32 decode, sort, and dispatch module which transforms the statements into VLIW equivalents. When the processor encounters pre-optimized VLIW code, it simply bypasses the dispatch unit. Intel calls this synergy "Explicit Parallel Computing" (EPIC), partly as a marketing decision to avoid previous negative connotations of VLIW and also because it is not pure VLIW. Because Intel had to maintain back-compatibility with IA-32 code, the IA-64 achieves only the performance enhancements associated with VLIW and not the cost-savings.[6]

Transmeta has also implemented VLIW in its low-power Crusoe processors. Increasingly complex superscalar processing requires massive amounts of control logic to properly schedule instructions; this effect does not exist in VLIW processors since the scheduling was completed at run time; with the removal of this complicated hardware, the power requirements decrease. In order to achieve the desired effects, though, Transmeta required that their processor be able to treat all code as VLIW, and to do so, they developed a software translator that converts, in real time, IA-32 binaries into VLIW bytecode. Crusoe chips are able to execute four 32-bit instructions each cycle. [7]

IBM Research is also developing its own software translator for existing code into VLIW format, the Dynamically Architected Instruction Set from Yorktown (DAISY). An early version of the software is available for download directly from their website. DAISY is able to convert binaries compiled for PowerPC, x86, S/390 architectures as well as the Java Virtual Machine into optimized VLIW code, bringing the added benefit of cross-architecture performance [8]. Of course, the trade off of these software translators is that some processor cycles are spent modifying the code, and so the gain in other goals – performance boosts and power reduction through VLIW or cross-architecture compatibility – must be sufficient to justify this performance drop.

VLIW has also appeared in some lower-end processors. STMicro launched a generalized VLIW processor to compete with specialized DSP chips. This offered a mix of advantages: low cost processors and easier coding. DSP coding has been described as particularly painful, the VLIW compiler used is able to produce bytecode that rivals its DSP competitors from languages as comfortable to programmers as C [12].

V. Conclusion

VLIW processing offers some significant advantages in chip size and cost compared to superscalar instruction-level parallelism, but these benefits come with the cost of much more complex and processor-specific compilers. Current VLIW processors, such as the IA-64, maintain backwards compatibility by offering a legacy mode, though this results in performance slowdowns when running the outdated code and eliminates some of the advantages offered by VLIW processors. Advances made in binary-to-binary code translation by Transmeta and IBM may eventually lead to pure-VLIW processors if they are able to rapidly and reliably translate binaries for one architecture into binaries optimized for any specific VLIW chip.

Acknowledgment

The authors would like to thank Mark Chang for providing the Computer Architecture foundation we needed to stand on in order complete this project. Additionally, the authors simply must thank Kathy King and Sarah Zwicker for their company during long hours in the library.

References

[1] Philips Semiconductors, *An Introduction to Very Long Instruction Word Computer Architecture.* Publication # 9397-750-01759.

[2] Fisher, J.A.. “Global code generation for instruction-level parallelism: Trace scheduling-2,” Technical Report HPL-93-43, Hewlett Packard Laboratories, June 1993.

[3] Wahlen, Oliver et al. “Codesign of Hardware, Software, and Algorithms.” *Proceedings of the Joint conference on Languages, Compilers and Tools for Embedded Systems*, 2002.

[4] Wilberg, J. et al. “Application specific compiler/architecture codesign: a case study,” *IEEE Circuits and Systems, ISCAS ’96., “Connecting the World”*, 1996.

[5] Intel Corporation. *Intel Itanium 2 Hardware Developer’s Manual*, July 2002. http://www.intel.com/design/itanium2/manuals/25110901.pdf.

[6] Sharangpani, Harsh. “Intel Itanium Microarchitecture Overview,” *Intel Microprocessor Forum*, October 1999. http://www.dig64.org/More_on_DIG64/microarch_ovw.pdf

[7] Transmeta Corporation. “Crusoe Architecture.” http://www.transmeta.com/crusoe/vliw.html

[8] Altman, Erik and Kemal Ebcioglu. “DAISY: Dynamic Compilation for 100% Architectural Compatibility.” IBM Labs *RC20538*, 1996. http://www.transmeta.com/crusoe/vliw.html

[9] Pountain, Dick. “The Word on VLIW,” *BYTE Talk*, 1996.

[10] Len, Maksim and Ilya Vaitsman. “Old Architecture of the New Generation, ” *Digit-Life,* 2002.

[11] Apple Computer Corporation “Apple PowerPC Execution Core,” 2004. http://www.apple.com/g5processor/executioncore.html.

[12] Cataldo, “HP Helps put VLIW into STMicro’s System-Chip,” *Design and Resuse,* 22 January 2002.