Microprocessor Design

Contents

Articles

Article Licenses

Wikibooks:Collections Preface

This book was created by volunteers at Wikibooks (http://en.wikibooks.org).

What is Wikibooks?

Started in 2003 as an offshoot of the popular Wikipedia project, Wikibooks is a free, collaborative wiki website dedicated to creating high-quality textbooks and other educational books for students around the world. In addition to English, Wikibooks is available in over 130 languages, a complete listing of which can be found at http://www.wikibooks.org. Wikibooks is a "wiki", which means anybody can edit the content there at any time. If you find an error or omission in this book, you can log on to Wikibooks to make corrections and additions as necessary. All of your changes go live on the website immediately, so your effort can be enjoyed and utilized by other readers and editors without delay.

Books at Wikibooks are written by volunteers, and can be accessed and printed for free from the website. Wikibooks is operated entirely by donations, and a certain portion of proceeds from sales is returned to the Wikimedia Foundation to help keep Wikibooks running smoothly. Because of the low overhead, we are able to produce and sell books for much cheaper then proprietary textbook publishers can. **This book can be edited by anybody at any time, including you.** We don't make you wait two years to get a new edition, and we don't stop selling old versions when a new one comes out.

What is this book?

This book was generated by the volunteers at Wikibooks, a team of people from around the world with varying backgrounds. The people who wrote this book may not be experts in the field. Some may not even have a passing familiarity with it. The result of this is that some information in this book may be incorrect, out of place, or misleading. For this reason, you should never rely on a community-edited Wikibook when dealing in matters of medical, legal, financial, or other importance. Please see our disclaimer for more details on this.

Despite the warning of the last paragraph, however, books at Wikibooks are continuously edited and improved. If errors are found they can be corrected immediately. If you find a problem in one of our books, we ask that you **be bold** in fixing it. You don't need anybody's permission to help or to make our books better.

Wikibooks runs off the assumption that many eyes can find many errors, and many able hands can fix them. Over time, with enough community involvement, the books at Wikibooks will become very high-quality indeed. **You are invited to participate at Wikibooks to help make our books better**. As you find problems in your book don't just complain about them: Log on and fix them! This is a kind of proactive and interactive reading experience that you probably aren't familiar with yet, so log on to http://en.wikibooks.org and take a look around at all the possibilities. We promise that we won't bite!

Who are the authors?

The volunteers at Wikibooks come from around the world and have a wide range of educational and professional backgrounds. They come to Wikibooks for different reasons, and perform different tasks. Some Wikibookians are prolific authors, some are perceptive editors, some fancy illustrators, others diligent organizers. Some Wikibookians find and remove spam, vandalism, and other nonsense as it appears. Most wikibookians perform a combination of these jobs.

It's difficult to say who are the authors for any particular book, because so many hands have touched it and so many changes have been made over time. It's not unheard of for a book to have been edited thousands of times by hundreds of authors and editors. *You could be one of them too*, if you're interested in helping out. At the time this book was prepared for print, there have been over ***' edits made by over*** *0*' registered users. These numbers are growing every day.

Wikibooks in Class

Books at Wikibooks are free, and with the proper editing and preparation they can be used as cost-effective textbooks in the classroom or for independent learners. In addition to using a Wikibook as a traditional read-only learning aide, it can also become an interactive class project. Several classes have come to Wikibooks to write new books and improve old books as part of their normal course work. In some cases, the books written by students one year are used to teach students in the same class next year. Books written can also be used in classes around the world by students who might not be able to afford traditional textbooks.

Happy Reading!

We at Wikibooks have put a lot of effort into these books, and we hope that you enjoy reading and learning from them. We want you to keep in mind that what you are holding is not a finished product but instead a work in progress. These books are never "finished" in the traditional sense, but they are ever-changing and evolving to meet the needs of readers and learners everywhere. Despite this constant change, we feel our books can be reliable and high-quality learning tools at a great price, and we hope you agree. Never hesitate to stop in at Wikibooks and make some edits of your own. We hope to see you there one day. **Happy reading!**

Microprocessor Design/Introduction

Microprocessor Design

About This Book

Computers and computer systems are a pervasive part of the modern world. Aside from just the common desktop PC, there are a number of other types of specialized computer systems that pop up in many different places. The central component of these computers and computer systems is the microprocessor, or the CPU. The CPU (short for "Central Processing Unit") is essentially the brains behind the computer system, it is the component that "computes". This book is going to discuss what microprocessor units do, how they do it, and how they are designed.

This book is going to discuss the design of microprocessor units, but it will not discuss the design of complete computer systems nor the design of other computer components or peripherals. Some microprocessor designs will be implemented and synthesized in Hardware Description Languages, such as Verilog or VHDL. The book will be organized to discuss simple designs and concepts first, and expand the initial designs to include more complicated concepts as the book progresses.

This book will attempt to discuss the basic concepts and theory of microprocessor design from an abstract level, and give real-world examples as necessary. This book will not focus on studying any particular processor architecture, although several of the most common architectures will appear frequently in examples and notes.

How Will This Book Be Organized?

The first section of the book will review computer architecture, and will give a brief overview of the components of a computer, the components of a microprocessor, and some of the basic architectures of modern microprocessors.

The second section will discuss in some detail the individual components of a microcontroller, what they do, and how they are designed.

The third section will focus in on the ALU and FPU, and will discuss implementation of particular mathematical operations.

The fourth section will discuss the various design paradigms, starting with the most simple single cycle machine to more complicated exotic architectures such as vector and VLIW machines.

Additional chapters will serve as extensions and support chapters for concepts discussed in the first four sections.

Prerequisites

This book will rely on some important background information that is currently covered in a number of other local wikibooks. Readers of this book will find the following prerequisites important to understand the material in this book:

- Digital Circuits
- Programmable Logic
- Embedded Systems
- Assembly Language

All readers **must be familiar with binary numbers** and also hexadecimal numbers. These notations will be used throughout the book without any prior explanation. Readers of this book should be familiar with at least one assembly language, and should also be familiar with a hardware description language. This book will use both types of languages in the main narrative of the text without offering explanation beforehand. Appendices might be included that contain primers on this material.

Readers of this book will also find some pieces of software helpful in examples. Specifically, assemblers and assembly language simulators will help with many of the examples. Likewise, HDL compilers and simulators will be useful in the design examples. If free versions of these software programs can be found, links will be added in an appendix.

Who Is This Book For?

This book is designed to accompany an advanced undergraduate or graduate study in the field of microprocessor design. Students in the areas of Electrical Engineering, Computer Engineering, or Computer Science will likely find this book to be the most useful. The basic subjects in this field will be covered, and more advanced topics will be included depending on the proficiencies of the authors. Many of the topics considered in this book will apply to the design of many different types of digital hardware, including ASICs. However, the main narrative of the book, and the ultimate goals of the book will be focused on microcontrollers and microprocessors, not other ASICs.

What This Book Will Not Cover

This book is about the design of micro-controllers and microprocessors only. This book will not cover the following topics in any detail, although some mention might be made of them as a matter of interest:

- Transistor mechanics, semiconductors, or integrated circuit fabrication (Microtechnology)
- Digital Circuit Logic, Design or Layout (Programmable Logic)
- Design or interfacing with other computer components or peripherals (Embedded Systems)
- Design or implementation of communication protocols used to communicate between computer components (Serial Programming)
- Design or creation of computer software (Computer Programming)
- Design of System-on-a-Chip hardware or any device with an integrated micro-controller

Terminology

Throughout the book, the words "Microprocessor", "Microcontroller", "Processor", and "CPU" will all generally be used interchangeably to denote a digital processing element capable of performing arithmetic and quantitative comparisons. We may differentiate between these terms in individual sections, but an explanation of the differences will always be provided.

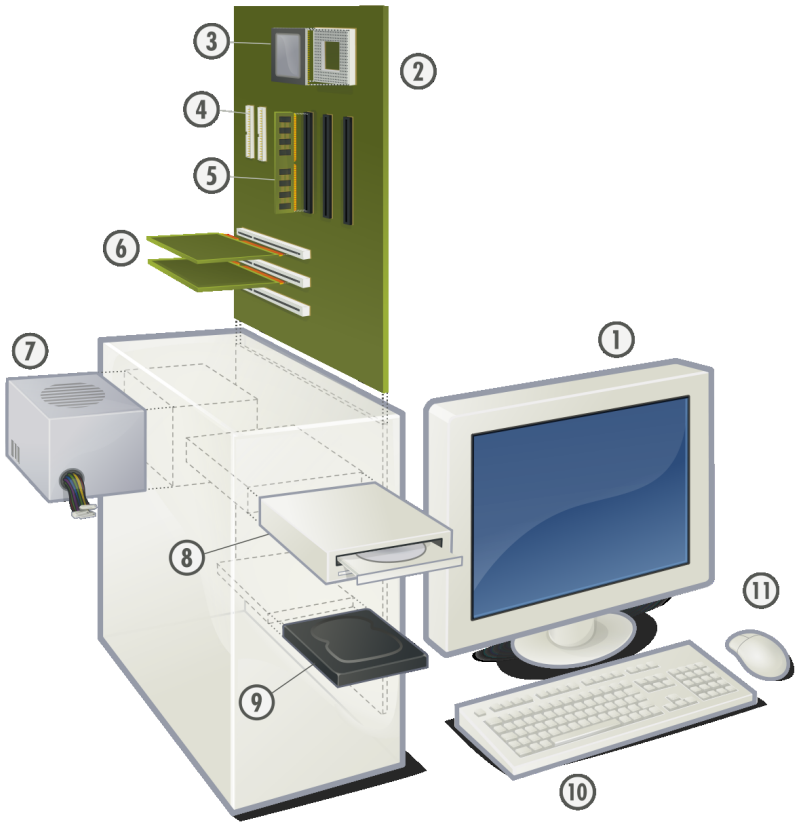
Microprocessor Basics

Microprocessor Design/Microprocessors

Microprocessor Design

Microprocessors

Microprocessors are the devices in a computer which make things happen. Microprocessors are capable of performing basic arithmetic operations, moving data from place to place, and making basic decisions based on the quantity of certain values.



The components of a PC computer. Part number 3 is the CPU.

Types of Processors

The vast majority of microprocessors are embedded microcontrollers. The second most common type of processors are common desktop processors, such as Intel's Pentium or AMD's Athlon. Less common are the extremely powerful processors used in high-end servers, such as Sun's SPARC, IBM's Power, or Intel's Itanium.

Historically, microprocessors and microcontrollers have come in "standard sizes" of 8 bits, 16 bits, 32 bits, and 64 bits. These sizes are common, but that does not mean that other sizes are not available. Some microcontrollers (usually specially designed embedded chips) can come in other "non-standard" sizes such as 4 bits, 12 bits, 18 bits, or 24 bits. The number of bits represent how much physical memory can be directly addressed by the CPU. It also represents the amount of bits that can be read by one read/write operation. In some circumstances, these are different; for instance, many 8 bit microprocessors have an 8 bit data bus and a 16 bit address bus.

- 8 bit processors can read/write 1 byte at a time and can directly address 256 bytes
- 16 bit processors can read/write 2 bytes at a time, and can address 65,536 bytes (64 Kilobytes)
- 32 bit processors can read/write 4 bytes at a time, and can address 4,294,967,295 bytes (4 Gigabytes)
- 64 bit processors can read/write 8 bytes at a time, and can address 18,446,744,073,709,551,616 bytes (16 Exabytes)

General Purpose Versus Specific Use

Microprocessors that are capable of performing a wide range of tasks are called **general purpose microprocessors**. General purpose microprocessors are typically the kind of CPUs found in desktop computer systems. These chips typically are capable of a wide range of tasks (integer and floating point arithmetic, external memory interface, general I/O, etc). We will discuss some of the other types of processor units available:

General Purpose

> A general purpose processing unit, typically referred to as a "microprocessor" is a chip that is designed to be integrated into a larger system with peripherals and external RAM. These chips can typically be used with a very wide array of software.

DSP

> A Digital Signal Processor, or DSP for short, is a chip that is specifically designed for fast arithmetic operations, especially addition and multiplication. These chips are designed with processing speed in mind, and don't typically have the same flexibility as general purpose microprocessors. DSPs also have special address generation units that can manage circular buffers, perform bit-reversed addressing, and simultaneously access multiple memory spaces with little to no overhead. They also support zero-overhead looping, and a single-cycle multiply-accumulate instruction. They are not typically more powerful than general purpose microprocessors, but can perform signal processing tasks using far less power (as in watts).

Embedded Controller

> Embedded controllers, or "microcontrollers" are microprocessors with additional hardware integrated into a single chip. Many microcontrollers have RAM, ROM, A/D and D/A converters, interrupt controllers, timers, and even oscillators built into the chip itself. These controllers are designed to be used in situations where a whole computer system isn't available, and only a small amount of simple processing needs to be performed.

Programmable State Machines

> The most simplistic of processors, programmable state machines are a minimalist microprocessor that is designed for very small and simple operations. PSMs typically have very small amount of program ROM available, limited scratch-pad RAM, and they are also typically limited in the type and number of instructions that they can perform. PSMs can either be used stand-alone, or (more frequently) they are embedded directly into the design of a larger chip.

Graphics Processing Units

Computer graphics are so complicated that functions to process the visuals of video and game applications have been offloaded to a special type of processor known as a GPU. GPUs typically require specialized hardware to implement matrix multiplications and vector arithmetic. GPUs are typically also highly parallelized, performing shading calculations on multiple pixels and surfaces simultaneously.

Types of Use

Microcontrollers and Microprocessors are used for a number of different types of applications. People may be the most familiar with the desktop PC, but the fact is that desktop PCs make up only a small fraction of all microprocessors in use today. We will list here some of the basic uses for microprocessors:

Signal Processing

Signal processing is an area that demands high performance from microcontroller chips to perform complex mathematical tasks. Signal processing systems typically need to have low latency, and are very deadline driven. An example of a signal processing application is the decoding of digital television and radio signals.

Real Time Applications

Some tasks need to be performed so quickly that even the slightest delay or inefficiency can be detrimental. These applications are known as "real time systems", and timing is of the upmost importance. An example of a real-time system is the anti-lock braking system (ABS) controller in modern automobiles.

Throughput and Routing

Throughput and routing is the use of a processor where data is moved from one particular input to an output, without necessarily requiring any processing. An example is an internet router, that reads in data packets and sends them out on a different port.

Sensor monitoring

Many processors, especially small embedded processors are used to monitor sensors. The microprocessor will either digitize and filter the sensor signals, or it will read the signals and produce status outputs (the sensor is good, the sensor is bad). An example of a sensor monitoring processor is the processor inside an antilock brake system: This processor reads the brake sensor to determine when the brakes have locked up, and then outputs a control signal to activate the rest of the system.

General Computing

A general purpose processor is like the kind of processor that is typically found inside a desktop PC. Names such as Intel and AMD are typically associated with this type of processor, and this is also the kind of processor that the public is most familiar with.

Graphics

Processing of digital graphics is an area where specialized processor units are frequently employed. With the advent of digital television, graphics processors are becoming more common. Graphics processors need to be able to perform multiple simultaneous operations. In digital video, for instance, a million pixels or more will need to be processed for every single frame, and a particular signal may have 60 frames per second! To the benefit of graphics processors, the color value of a pixel is typically not dependent on the values of surrounding pixels, and therefore many pixels can typically be computed in parallel.

Abstraction Layers

Computer systems are developed in layers known as layers of abstraction. Layers of abstraction allow people to develop computer components (hardware and software) without having to worry about the internal design of the other layers in the system. At the highest level are the user-interface programs that people use on their computers. At the lowest level are the transistor layouts of the individual computer components. Some of the layers in a computer system are (listed from highest to lowest):

1. Application
2. Operating System
3. Firmware
4. Instruction Set Architecture
5. Microprocessor Control Logic
6. Physical Circuit Layout

This book will be mostly concerned with the Instruction Set Architecture (ISA), and the Microprocessor Control Logic. Topics above these are typically the realm of computer programmers. The bottom layer, the Physical Circuit Layout is the job of hardware and VLSI engineers.

ISA

The **Instruction Set Architecture** is a long name for the assembly language of a particular machine, and the associated machine code for that assembly language. We will discuss this below.

Assembly Language

An assembly language is a small language that contains a short word or "mnemonic" for each individual command that a microcontroller can follow. Each command gets a single mnemonic, and each mnemonic corresponds to a single machine command. Assembly language gets converted (by a program called an "assembler") into the binary machine code. The machine code is specific to each different type of machine.

Common ISAs

> Wikibooks contains books about programming in multiple different types of assembly language. For more information about Assembly language, or for books on a particular ISA, see Assembly Language.

Some of the most common ISAs, listed in order of popularity (most popular first) are:

- ARM
- IA-32 (Intel x86)
- MIPS
- Motorola 68K
- PowerPC
- Hitachi SH
- SPARC

Moore's Law

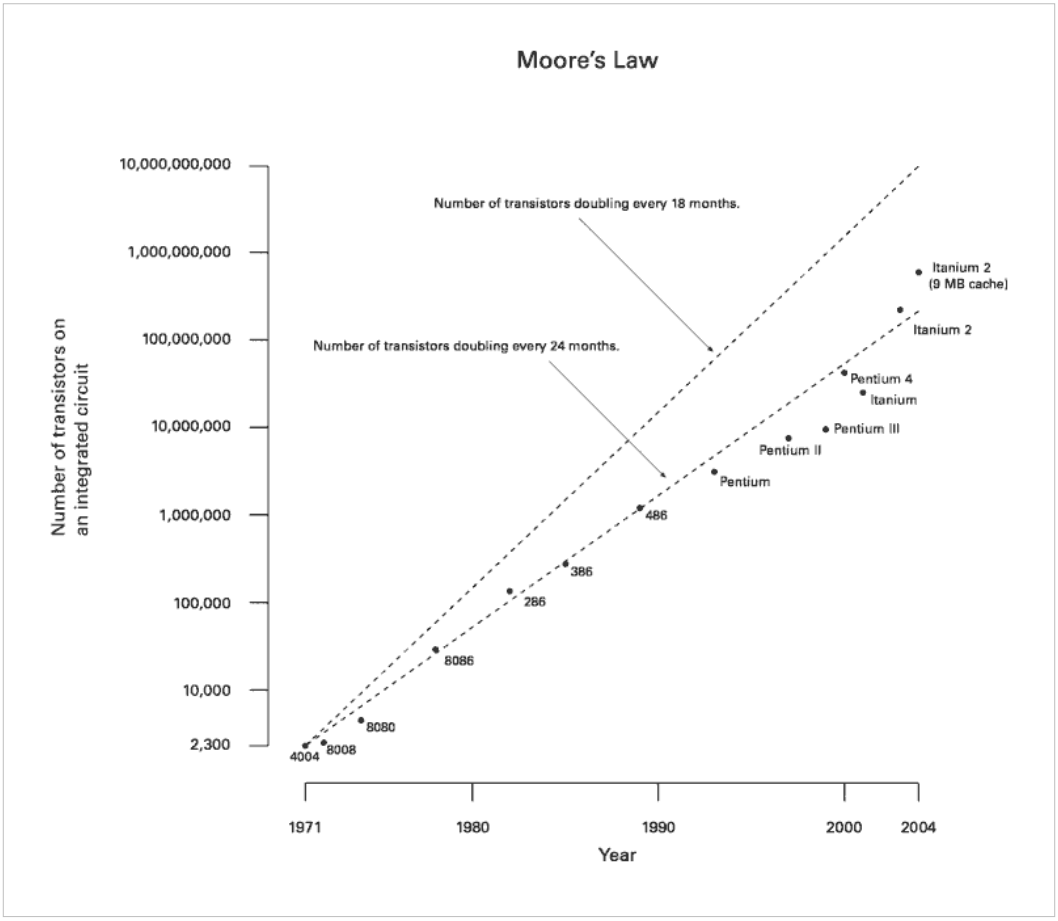
A common law that governs the world of microprocessors is **Moore's Law**. Moore's Law, originally by Dr. Carver Mead at Caltech, and summarized famously by Intel Founder Gordon Moore. Moore's Law states that the number of transistors on a single chip at the same price will double every 18 to 24 months. This law has held without fail since it was originally stated in 1965. Current microprocessor chips contain millions of transistors and the number is growing rapidly. Here is Moore's summarization of the law from Electronics Magazine in 1965:

> *The complexity for minimum component costs has increased at a rate of roughly a factor of two per year...Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years. That means by 1975, the number of components per integrated circuit for minimum cost will be 65,000. I believe that such a large circuit can be built on a single wafer.*
>
> —Gordon Moore

Moore's Law has been used incorrectly to calculate the speed of an integrated circuit, or even to calculate it's power consumption, but neither of these interpretations are true. Also, Moore's law is talking about the number of transistors on a chip for a "minimum component cost", which means that the number of transistors on a chip, for the same price, will double. This goes to show that chips for less price can have fewer transistors, and that chips at a higher price can have more transistors. On an economic note, a consequence of Moore's Law is that companies need to continue to innovate and integrate more transistors onto a single chip, without being able to increase prices.

Moore's Law does not require that the speed of the chip increase along with the number of transistors on the chip. However, the two measurements are typically related. Some points to keep in mind about transistors and Moore's Law are:

1. Smaller Transistors typically switch faster then larger transistors.
2. To get more transistors on a single chip, the chip needs to be made larger, or the transistors need to be made smaller. Typically, the transistors get smaller.
3. Transistors tend to leak electrical current as they get smaller. This means that smaller transistors require more power to operate, and they generate more heat.
4. Transistors tend to generate heat as a function of frequencies. Higher clock rates tend to generate more heat.



Moore's law is occasionally misinterpreted to mean that the speed of processors, in hertz will double every 18 months. This is not strictly true, although the speed of processors does tend to increase as transistors are made

smaller and more compact. With the advent of multi-core processors, some people have used Moore's law to mean that processor throughput increases with time, which is not strictly the case either (although it is a likely side effect of Moore's law).

Clock Rates

Microprocessors are typically discussed in terms of their clock speed. The clock speed is measured in **hertz** (or megahertz, or gigahertz). A hertz is a "cycle per second". Each cycle, a microprocessor will perform certain tasks, although the amount of work performed in a single cycle will be different for different types of processors. The amount of work that a processor can complete in a single cycle is measured in "cycles per instruction". For some systems, such as MIPS, there is 1 cycle per instruction. For other systems, such as modern x86 chips, there are typically very many cycles per instruction.

The clock rate is equated as such:

$$\text{Clock Time} = \frac{1}{\text{Clock Rate}}$$

This means that the amount of time for a cycle is inversely proportional to the clock rate. A computer with a 1MHz clock rate will have a clock time of 1 microsecond. A modern desktop computer with a 3.2 GHz processor will have a clock time of approximately 3×10^{-10} seconds, or 300 picoseconds. 300 picoseconds is an incredibly small amount of time, and there is a lot that needs to happen inside the processor in each clock cycle.

Basic Elements of a Computer

There are a few basic elements that are common to all computers. These elements are:

- CPU
- Memory
- Input Devices
- Output Devices

Depending on the particular computer architecture, these elements may be available in various sizes, and they may be accompanied by additional elements.

Microprocessor Design/Computer Architecture

Von Neumann Architecture

Early computer programs were hard wired. To reprogram a computer meant changing the hardware switches manually, that took a long time with potential errors. Computer memory was only used for storing data.

John von Neumann suggested that data and programs should be stored together in memory, it is now called Von Neumann architecture. Programs are fetched from memory for executions by a central unit that is what we call the CPU. Basically programs and data are represented on memory in the same way. The program is just data coded for special meaning.

A Von Neumann microprocessor is a processor that follows this pattern:

Fetch

: An instruction and the necessary data are obtained from memory.

Decode

: The instruction and data are separated, and the components and pathways required to execute the instruction are activated.

Execute

: The instruction is performed, the data is manipulated, and the results are stored.

This pattern is typically implemented by separating the task into two components, the **control**, and the **datapath**.

Control

The control unit reads the instruction, and activates the appropriate parts of the datapath.

Datapath

The datapath is the pathway that the data takes through the microprocessor. As the data travels to different parts of the datapath, the command signals from the control unit cause the data to be manipulated in specific ways, according to the instruction. The datapath consists of the circuitry for transforming data and for storing temporary data. It contains ALUs capable of transforming data through operations such as addition, subtraction, logical AND, OR, inverting, and shifting.

Harvard Architecture

In a **Harvard Architecture** machine, the computer system's memory is separated into two discrete parts: data and instructions. In a pure Harvard system, the two different memories occupy separate memory modules, and instructions can only be executed from the instruction memory.

In a "Princeton Architecture" machine, the computer system's memory comprises one uniform address space: data and instructions may be placed anywhere in this single memory.

Many DSPs are modified Harvard architectures, designed to simultaneously access three distinct memory areas: the program instructions, the signal data samples, and the filter coefficients (often called the P, X, and Y memories).

In theory, such three-way Harvard architectures can be three times as fast as a Princeton architecture that is forced to read the instruction, the data sample, and the filter coefficient, one at a time.

In Princeton architecture systems, there is only one memory area. Any particular memory location that can be written to and read as data at any one time can also be executed as an instruction at other times. Several common computer security problems arise because modern processors are not pure Harvard systems, and manipulate instructions as if

they were data, and vice-versa.

RISC and CISC and DSP

Historically, the first type of ISA was the **complex instruction set computers** (CISC), and the second type was the **reduced instruction set computers** (RISC). It is a common misunderstanding that RISC systems typically have a small ISA (fewer instructions) but make up for it with faster hardware. RISC system actually have "reduced instructions", in the sense that each instruction does so little that it takes very little time to execute it. It is a common misunderstanding that CISC systems have more instructions, but typically pay a steep performance penalty for the added versatility. CISC systems actually have "complex instructions", in the sense that at least one instruction takes a long time to execute -- for example, the "double indirect" addressing mode inherently requires two memory cycles to execute, and a few CPUs have a "string copy" instruction that may require hundreds of memory cycles to execute. MIPS and SPARC are examples of RISC computers. Intel x86 is an example of a CISC computer.

Some people group stack machines with the RISC machines; others[1] group stack machines with the CISC machines; some people [2] describe stack machines as neither RISC nor CISC.

Other ISA types include DSPs, stack machines, VLIW machines, MISC machines, TTA architectures, massively parallel processor arrays, etc.

We will discuss these terms and concepts in more detail later.

Microprocessor Components

Some of the common components of a microprocessor are:

- Control Unit
- I/O Units
- Arithmetic Logic Unit (ALU)
- Registers
- Cache

We will give a brief introduction to these components below.

Control Unit

The control unit, as described above, reads the instructions, and generates the necessary digital signals to operate the other components. An instruction to add two numbers together would cause the Control Unit to activate the addition module, for instance.

I/O Units

A processor needs to be able to communicate with the rest of the computer system. This communication occurs through the I/O ports. The I/O ports will interface with the system memory (RAM), and also the other peripherals of a computer.

Arithmetic Logic Unit

The **Arithmetic Logic Unit**, or ALU is the part of the microprocessor that performs arithmetic operations. ALUs can typically add, subtract, divide, multiply, and perform logical operations of two numbers (and, or, nor, not, etc).

Registers

In this book, we talk about lots of different kinds of registers. Hopefully it will be obvious which kind of register we are talking about from the context.

The most general meaning is a "hardware register": anything that can be used to store bits of information, in a way that all the bits of the register can be written to or read out simultaneously. Since registers outside of a CPU are also outside the scope of the book, this book will only discuss processor registers, which are hardware registers that happen to be inside a CPU. But usually we will refer to a more specific kind of register.

programmer-visible registers

The programmer-visible registers, also called the user-accessible registers, also called the architectural registers, often simply called "the registers", are the registers that are directly encoded as part of at least one instruction in the instruction set.

The registers are the fastest accessible memory locations, and because they are so fast, there are typically very few of them. In most processors, there are fewer than 32 registers. The size of the registers defines the size of the computer. For instance, a "32 bit computer" has registers that are 32 bits long. The length of a register is known as the **word length** of the computer.

There are several factors limiting the number of registers, including:

- It is very convenient for a new CPU to be software-compatible with an old CPU. This requires the new chip to have exactly the same number of programmer-visible registers as the old chip.
- Doubling the number general-purpose registers requires adding another bit to each instruction that selects a particular register. Each 3-operand instruction (that specify 2 source operands and a destination operand) would expand by 3 bits. Modern chip manufacturing processes could put a million registers on a chip; that would make each and every 3-operand instruction require 60 bits just to select the registers, not counting the bits required to specify what to do with those operands.
- Adding more registers adds more wires to the critical path, adding capacitance, which reduces the maximum clock speed of the CPU.
- Historically, CPUs were designed with few registers, because each additional register increased the cost of the CPU significantly. But now that modern chip manufacturing can put tens of millions of bits of storage on a single commodity CPU chip, this is less of an issue.

Microprocessors typically contain a large number of registers, but only a small number of them are accessible by the programmer. The registers that can be used by the programmer to store arbitrary data, as needed, are called **general purpose registers**. Registers that cannot be accessed by the programmer directly are known as **reserved registers**[*citation needed*].

Some computers have highly specialized registers -- memory addresses always came from the program counter or "the" index register or "the" stack pointer; one ALU input was always hooked to data coming from memory, the other ALU input was always hooked to "the" accumulator; etc.

Other computers have more general-purpose registers -- any instruction that access memory can use any address register as a index register or as a stack pointer; any instruction that uses the ALU can use any data register.

Other computers have completely general-purpose registers -- any register can be used as data or an address in any instruction, without restriction.

microarchitectural registers

Besides the programmer-visible registers, all CPUs have other registers that are not programmer-visible, called "microarchitectural registers" or "physical registers".

These registers include:

- memory address register
- memory data register
- instruction register
- microinstruction register
- microprogram counter
- pipeline registers
- extra physical registers to support register renaming
- the prefetch input queue
- writable control stores (We will discuss the control store in the Microprocessor Design/Control Unit and Microprocessor Design/Microcode)
- Some people consider on-chip cache to be part of the microarchitectural registers; others consider it "outside" the CPU.

There are a wide variety of ways to implement any one instruction set. The vast majority of these microarchitectural registers are technically not "necessary". A designer could choose to design a CPU that had almost no physical registers other than the programmer-visible registers. However, many designers choose to design a CPU with lots of physical registers, using them in ways that make the CPU execute the same given instruction set much faster than a CPU that lacks those registers.

Cache

Most CPUs manufactured do not have any cache.

Cache is memory that is located on the chip, but that is not considered registers. The cache is used because reading external memory is very slow (compared to the speed of the processor), and reading a local cache is much faster. In modern processors, the cache can take up as much as 50% or more of the total area of the chip. The following table shows the relationship between different types of memory:

smallest		largest
Registers	cache	RAM
fastest		slowest

Cache typically comes in 2 or 3 "levels", depending on the chip. Level 1 (L1) cache is smaller and faster than Level 2 (L2) cache, which is larger and slower. Some chips have Level 3 (L3) cache as well, which is larger still than the L2 cache (although L3 cache is still much faster than external RAM).

Endian

Different computers order their multi-byte data words (i.e., 16-, 32-, or 64-bit words) in different ways in RAM. Each individual byte in a multi-byte word is still separately addressable. Some computers order their data with the most significant byte of a word in the lowest address, while others order their data with the most significant byte of a word in the highest address. There is logic behind both approaches, and this was formerly a topic of heated debate.

This distinction is known as **endianness**. Computers that order data with the least significant byte in the lowest address are known as "Little Endian", and computers that order the data with the most significant byte in the lowest address are known as "Big Endian". It is easier for a human (typically a programmer) to view multi-word data dumped to a screen one byte at a time if it is ordered as Big Endian. However it makes more sense to others to store the LS data at the LS address.

When using a computer this distinction is typically transparent; that is that the user cannot tell the difference between computers that use the different formats. However, difficulty arises when different types of computers attempt to communicate with one another over a network.

With a big-endian 68K sort of machine,

```
address increases > ------ >
data   : 74 65 73 74 00 00 00 05
```

is the string "test" followed by the 32-bit integer 5. The little-endian x86 sort of machine would interpret the last part as the integer 0x0500_0000.

When communicating over a network composed of both big-endian and little-endian machines, the network hardware (should) apply the Address Invariance principle, to avoid scrambling text (avoiding the NUXI problem). High-level software (should) format packets of data to be transmitted over the network in Network Byte Order. High-level software (should) be written as "endian clean" -- always reading and writing 16 bit integers as whole 16 bit integers, 32 bit integers as whole 32 bit integers, etc. -- so no changes are needed to re-compile it for big-endian or little-endian machines. Software that is not "endian clean" -- software that writes integers, but then reads them out as 8 bit octets or integers of some other length -- usually fails when re-compiled for another computer.

A few computers -- including nearly all DSPs -- are "neither-endian". They always read and write complete aligned words, and don't have any hardware for dealing with individual bytes. Systems build on top of such computers often *do* have a particular endianness -- but that endianness is written into the software, and can be switched by re-compiling for the opposite endianness.

Stack

A stack is a block of memory that is used as a scratchpad area. The stack is a sequential set of memory locations that is set to act like a LIFO (last in, first out) buffer. Data is added to the top of the stack in a "push" operation, and the top data item is removed from the stack during a "pop" operation. Most computer architectures include at least a register that is usually reserved for the stack pointer.

Some microprocessors include a small hardware stack built into the CPU, independent from the rest of the RAM.

Modern Computers

Modern desktop computers, especially computers based on the Intel x86 ISA are not Harvard computers, although the newer variants have features that are "Harvard-Like". All information, program instructions, and data are stored in the same RAM areas. However, a modern feature called "paging" allows the physical memory to be segmented into large blocks of memory called "pages". Each page of memory can either be instructions or data, but not both.

Modern embedded computers, however, are typically based on a Harvard architecture. Instructions are stored in a different addressable memory block than the data is, and there is no way for the microprocessor to interchange data and instructions.

further reading

- Wikipedia: writable control stores

References

[1] http://www.cs.uiowa.edu/~jones/arch/cisc/
[2] http://www.ultratechnology.com/ml0.htm

Microprocessor Design/Instruction Set Architectures

Microprocessor Design

ISAs

The **instruction set** or the **instruction set architecture** (**ISA**) is the set of basic instructions that a processor understands. The instruction set is a portion of what makes up an architecture.

Historically, the first two philosophies to instruction sets were: reduced (RISC) and complex (CISC). The merits and argued performance gains by each philosophy are and have been thoroughly debated.

CISC

Complex Instruction Set Computer (**CISC**) is rooted in the history of computing. Originally there were no compilers and programs had to be coded by hand one instruction at a time. To ease programming more and more instructions were added. Many of these instructions are complicated combination instructions such as loops. In general, more complicated or specialized instructions are inefficient in hardware, and in a typically CISC architecture the best performance can be obtained by using only the most simple instructions from the ISA.

The most well known/commoditized CISC ISAs are the Motorola 68k and Intel x86 architectures.

RISC

Reduced Instruction Set Computer (**RISC**) was realized in the late 1970s by IBM. Researchers discovered that most programs did not take advantage of all the various address modes that could be used with the instructions. By reducing the number of address modes and breaking down multi-cycle instructions into multiple single-cycle instructions several advantages were realized:

- compilers were easier to write (easier to optimize)
- performance is increased for programs that did simple operations
- the clock rate can be increased since the minimum cycle time was determined by the longest running instruction

The most well known/commoditized RISC ISAs are the PowerPC, ARM, MIPS and SPARC architectures.

VLIW

We will discuss VLIW Processors in a later section.

Vector processors

We will discuss Vector Processors in a later section.

Memory Arrangement

Instructions are typically arranged sequentially in memory. Each instruction occupies 1 or more computer words. The **Program Counter** (PC) is a register inside the microprocessor that contains the address of the current instruction.[1] During the fetch cycle, the instruction from the address indicated by the program counter is read from memory into the instruction register (IR), and the program counter is incremented by *n*, where *n* is the word length of the machine (in bytes).

In addition to fetches of the executable instructions, many (but not all) instructions also fetch data values from memory ("load") into a data register, or write data values from a data register to memory ("store"). The address of the particular memory word accessed in such a load or store instruction is called the "effective address". In the simplest instruction sets, the effective address always contained in some address register. Other instruction sets have more complex "effective address" calculations — we will discuss such "addressing modes" later.

Common Instructions

Move, Load, Store

Move instructions cause data from one register to be moved or copied to another register. Load instructions put data from an external source, such as memory, into a register. Store instructions move data from a register to an external destination.

Instructions that move (or copy) data from one place to another are the #1 most-frequently-used instructions in most programs.[2]

Branch and Jump

Branching and Jumping is the ability to load the PC register with a new address that is not the next sequential address. In general, a "jump" or "call" occurs unconditionally, and a "branch" occurs on a given condition. In this book we will generally refer to both as being branches, with a "jump" being an unconditional branch.

A "call" instruction is a branch instruction with the additional effect of storing the current address in a specific location, e.g. pushing it on the stack, to allow for easy return to continue execution. A "call" instruction is generally matches with a "return" instruction which retrieves the stored address and resumes execution where it left off.

An "interrupt" instruction is a call to a preset location, generally one encoded somehow in the instruction itself. This is often used to reach commonly-used resources such as the operating system. Generally, a routine entered via an interrupt instruction is left via an interrupt return instruction, which, similarly to the return instruction, retrieves the stored address and resumes execution.

In many programs, "call" is the second-most-frequently used instruction (after "move").[2]

Arithmetic Instructions

The Arithmetic Logic Unit (ALU) is used to perform arithmetic and logical instructions. The capability of the ALU typically is greater with more advanced central processors, but RISC machines' ALUs are deliberately kept simple and so have only some of these functions. An ALU will, at minimum, perform addition, subtraction, NOT, AND, OR, and XOR, and usually also single-bit rotates and shifts. Many CISC machine ALUs can also perform multi-bit rotates and shifts (with a barrel shifter) and integer multiplication and division. While many modern CPUs can also do floating point mathematical operations, these are usually handled by the FPU, a different part of the machine. We describe the ALU in more detail in the ALU design chapter.

Input / Output

Input instructions fetch data from a specified input port, while output instructions send data to a specified output port. There is very little distinction between input/output space and memory space, the microprocessor presents an address and then either accepts data from, or sends data to, the data bus, but the sort of operations available in the input/output space are typically more limited than those available in memory space.

NOP

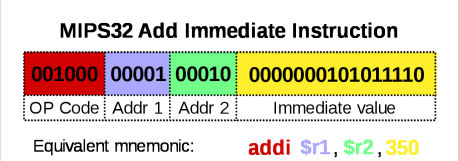
NOP, short for "no operation" is an instruction that produces no result and causes no side effects. NOPs are useful for timing and preventing **hazards**.

instruction length

There are several different ways people balance the various advantages and disadvantages of various instruction lengths.

Fixed-length instructions are less complicated for a CPU to handle than variable-width instructions for several reasons, and are therefore somewhat easier to optimize for speed. Such reasons include: CPUs with variable-length instructions have to check check whether each instruction straddles a cache line or virtual memory page boundary; CPUs with fixed-length instructions can skip all that.

There simply are not enough bits in a 16 bit instruction to accommodate 32 general-purpose registers, and also do "Ra = Rb (op)



A MIPS "add immediate" instruction includes the opcode (logical operation), the destination register specifier, the source register specifier, and a constant value, all in the same 32 bits that are used for every MIPS instruction.

Rc" -- i.e., independently select 2 source and 1 destination register out of a general purpose register bank of 32 registers, and also independently select one of several ALU operations.

And so people who design instruction sets must make one or more of the following compromises:

- sacrifice code density and use longer fixed-width instructions, typically 32 bit, such as the MIPS and DLX and ARM.
- sacrifice fixed-width instructions, requiring a more complicated decoder to handle both short 16 bit instructions and longer 3-operand instructions, such as ARM Thumb
- sacrifice 3-operands, using no more than 2 operands in all instructions for everything, such as the Atmel AVR. Without 3-operand instructions, programs occasionally require extra copy instructions when both variable input operands to some ALU operation need to be preserved for some later instruction(s).
- sacrifice registers, so only 16 or 8 programmer-visible registers.
- sacrifice the concept of general purpose register -- perhaps only 16 or 8 "data registers" are visible to 3-operand ALU instructions, as in the 68000, or the destination is restricted to one or two "accumulators", but other registers (such as "address registers") are visible to other instructions.

Further reading

[1] Practically all modern CPUs maintain the illusion of a program counter sequentially walking through code one instruction at a time. However, a few complex modern CPUs internally execute several instructions simultaneously (superscalar), or execute instructions out-of-order, or even speculatively pre-execute instructions down the "wrong" path, then back up and take the right path. When designing and testing such internal structures, the concept of "the" PC is a bit fuzzy.

Some processor architectures, for instance the CDP1802, do not have a single Program Counter; instead, one of the general purpose registers is used as a program counter, and which register that is can be changed under program control.

[2] Peter Kankowski. "x86 Machine Code Statistics" (http://www.strchr.com/x86_machine_code_statistics)

Microprocessor Design/Memory

Memory is a fundamental aspect of microcontroller design, and a good understanding of memory is necessary to discuss and processor system.

Memory Hierarchy

Memory suffers from the dichotomy that it can be either large or it can be fast. As memory becomes more large, it becomes less fast, and vice-versa. Because of this trade-off, computer systems typically have a hierarchy of memory types, where faster (and smaller) memories are closer to the processor, and slower (but larger) memories are further from the processor.

Hard Disk Drives

Hard Disk Drives (HDD) are occasionally known as **secondary memory** or *nonvolatile memory*. HDD typically stores data magnetically (although some newer models use FLASH), and data is maintained even when the computer is turned off or removed from power. HDD is several orders of magnitude slower then all other memory devices, and a computer system will be more efficient when the number of interactions with the HDD are minimized.

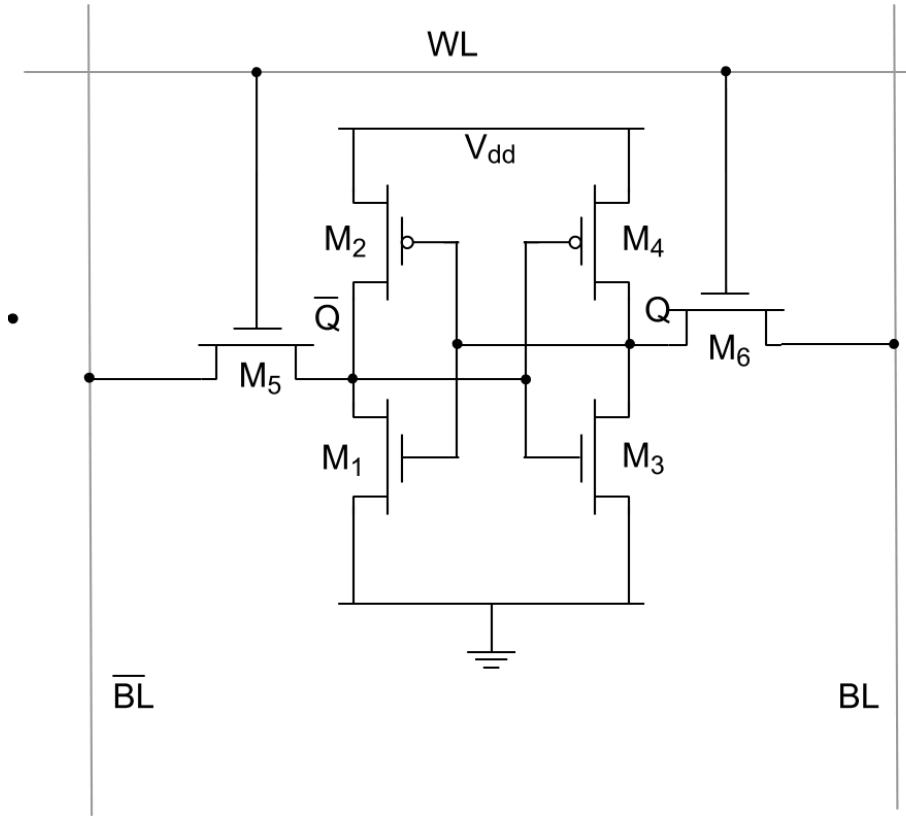
Because most HDDs are mechanical and have moving parts, they tend to wear out and fail after time.

RAM

Random Access Memory (RAM), also known as **main memory**, is a volatile storage that holds data for the processor. Unlike HDD storage, RAM typically only has a capacity of a few megabytes to a few gigabytes. There are two primary forms of RAM, and many variants on these.

SRAM

Static RAM (SRAM) is a type of memory storage that uses 6 transistors to store data. These transistors store data so long as power is supplied to the RAM and do not need to be refreshed.



A single bit of storage in SRAM, showing 6 transistors.

SRAM is typically used in processor cache, not main memory because it has a faster speed despite it's larger area.

DRAM

Dynamic RAM (DRAM) is a type of RAM that contains a single transistor and a capacitor. DRAM is smaller than SRAM, and therefore can store more data in a smaller area. Because of the charge and discharge times of the capacitor, however, DRAM tends to be slower than SRAM. Many modern types of Main Memory are based on DRAM design because of the high memory densities. Because DRAM is simpler than SRAM, it is typically cheaper to produce.

A popular type of RAM, SDRAM, is a variant of DRAM and is not related to SRAM.

As digital circuits continue to grow smaller and faster as per Moore's Law, the speed of DRAM is not increasing as rapidly. This means that as time goes on, the speed difference between the processor and the RAM units (so long as the RAM is based on DRAM or variants) will continue to increase, and communications between the two units becomes more inefficient.

Cache

Cache is memory that is smaller and faster than main memory and resides closer to the processor. RAM runs on the system bus clock, but Cache typically runs on the processor speed which can be 10 times faster or more. Cache is frequently divided into multiple levels: L1, L2, and L3, with L1 being the smallest and fastest, and L3 being the largest and slowest. We will discuss cache in more detail in a later chapter.

A computer system may have between several kilobytes to a few megabytes of available cache.

Registers

Registers are the smallest and fastest memory storage elements. A modern processor may have anywhere from 4 to 256 registers. We will discuss registers in much more detail in a later chapter, Microprocessor Design/Register File.

Microprocessor Design/Control and Datapath

Microprocessor Design

Most processors and other complicated hardware circuits are typically divided into two components: a **datapath** and a **control unit**. The datapath contains all the hardware necessary to perform all the necessary operations. In many cases, these hardware modules are parallel to one another, and the final result is determined by multiplexing all the partial results.

The control unit determines the operation of the datapath, by activating switches and passing control signals to the various multiplexers. In this way, the control unit can specify how the data flows through the datapath.

The width of the data path ...

> *"There is only one mistake that can be made in a computer design that is difficult to recover from: not providing enough address bits for memory addressing and memory management." -- Gordon Bell and Bill Strecker, 1975*[1]

For good code density, you want the ALU datapath width to be at least as wide as the address bus width. Then every time you need to increment an address, you can do it in a single instruction, rather than requiring multiple instructions to manipulate an address one piece at a time.[2] [3]

[1] Engineering Education "Today in History" PDP-11 minicomputer introduced (http://www.k-grayengineeringeducation.com/blog/index.php/2009/03/13/engineering-education-today-in-history-pdp-11-minicomputer-introduced-2/) by Gordon Bell 2009; referring to "What we learned from the PDP-11" (http://research.microsoft.com/~gbell/Digital/Bell_Strecker_What_we _learned_fm_PDP-11c 7511.pdf) by Gordon Bell and Bill Strecker, 1975. Early version of the PDP-11 had a 16-bit address space. ... also quoted in the book "Electronics" (http://books.google.com/books?id=6amzAAAAIAAJ&q=insufficient+address+computer+architecture&dq=insufficient+address+computer+architecture&pgis=1)

[2] "It seems that the 16-bit ISA hits somehow the "sweet spot" for the best code density, perhaps because the addresses are also 16-bit wide and are handled in a single instruction. In contrast, 8-bitters need multiple instructions to handle 16-bit addresses." -- "Insects of the computer world" (http://embeddedgurus.com/state-space/2009/03/insects-of-the-computer-world/) by Miro Samek 2009.

[3] "it just really sucks if the largest datum you can manipulate is smaller than your address size. This means that the accumulator needs to be the same size as the PC -- 16-bits." -- Allen "Opcode considerations" (http://david.carybros.com/html/computer_architecture.html#considerations)

Microprocessor Design/Performance

Microprocessor Design

Clock Cycles

The clock signal is a 1-bit signal that oscillates between a "1" and a "0" with a certain frequency. When the clock transitions from a "0" to a "1" it is called the **positive edge**, and when the clock transitions from a "1" to a "0" it is called the **negative edge**.

The time it takes to go from one positive edge to the next positive edge is known as the **clock period**, and represents one **clock cycle**.

The number of clock cycles that can fit in 1 second is called the **clock frequency**. To get the clock frequency, we can use the following formula:

$$\text{Clock Frequency} = \frac{1}{\text{Clock Period}}$$

Clock frequency is measured in units of *cycles per second*.

Cycles per Instruction

In many microprocessor designs, it is common for multiple clock cycles to transpire while performing a single instruction. For this reason, it is frequently useful to keep a count of how many cycles are required to perform a single instruction. This number is known as the **cycles per instruction**, or CPI of the processor.

Because all processors may operate using a different CPI, it is not possible to accurately compare multiple processors simply by comparing the clock frequencies. It is more useful to compare the number of **instructions per second**, which can be calculated as such:

$$\text{Instructions per Second} = \frac{\text{Clock Frequency}}{CPI}$$

One of the most common units of measure in modern processors is the "MIPS", which stands for *millions of instructions per second*. A processor with 5 MIPS can perform 5 million instructions every second. Another common metric is "FLOPS", which stands for *floating point operations per second*. MFLOPS is a million FLOPS, GFLOPS is a billion FLOPS, and TFLOPS is a trillion FLOPS.

Instruction count

The "instruction count" in microprocessor performance measurement is the number of instructions executed during the run of a program. Typical benchmark programs have instruction counts in the millions or billions -- even though the program itself may be very short, those benchmarks have inner loops that are repeated millions of times.

Some microprocessor designers have the freedom to add instructions to or remove instructions from the instruction set. Typically the only way to reduce the instruction count is to add instructions such that those inner loops can be re-written in a way that does the necessary work using fewer instructions -- those instructions do "more work" per instruction.

Sometimes, counter-intuitively, we can improve overall CPU performance (i.e., reduce CPU time) in a way that increases the instruction count, by using instructions in that inner loop that may do "less work" per instruction, but those instructions finish in less time.

CPU Time

CPU Time is the amount of time it takes the CPU to complete a particular program. CPU time is a function of the amount of time it takes to complete instructions, and the number of instructions in the program:

$$\text{CPU time} = \text{Instruction Count} \times CPI \times \text{Clock Cycle Time}$$

Sometimes we can improve one of the 3 components alone, reducing CPU time. But quite often we find a tradeoff -- say, a technique that increases instruction count, but reduces the clock cycle time -- and we have to measure the total CPU time to see if that technique makes the overall performance better or worse.

Amdahls Law

Amdahl's Law is a law concerned with computer performance and optimization. Amdahl's law states that an improvement in the speed of a single processor component will have a comparatively small effect on the performance of the overall processor unit.

In the most general sense, Amdahl's Law can be stated mathematically as follows:

$$\Delta = \frac{1}{\sum_{k=0}^{n} \left(\frac{P_k}{S_k}\right)}$$

where:

- Δ is the factor by which the program is sped up or slowed down,
- P_k is a percentage of the instructions that can be improved (or slowed),
- S_k is the speed-up multiplier (where 1 is no speed-up and no slowing),
- k represents a label for each different percentage and speed-up, and
- n is the number of different speed-up/slow-downs resulting from the system change.

For instance, if we make a speed improvement in the memory module, only the instructions that deal directly with the memory module will experience a speedup. In this case, the percentage of load and store instructions in our program will be P_0, and the factor by which those instructions are sped up will be S_0. All other instructions, which are not affected by the memory unit will be P_1, and the speed up will be S_1 Where:

$$P_1 = 1 - P_0$$

$$S_1 = 1$$

We set S_1 to 1 because those instructions are not sped up or slowed down by the change to the memory unit.

benchmarking

- SpecInt
- SpecFP
- "Maxim/Dallas APPLICATION NOTE 3593" [1] benchmarking
- "Mod51 Benchmarks" [2]
- EEMBC, the Embedded Microprocessor Benchmark Consortium [3]

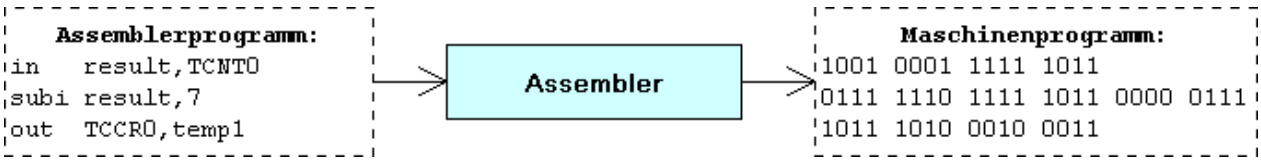
References

[1] http://www.maxim-ic.com/appnotes.cfm/appnote_number/3593
[2] http://www.designtools.co.nz/modbench.htm
[3] http://www.eembc.org/

Microprocessor Design/Assembly Language

Microprocessor Design

Assemblers



Assembly Language Constructs

There are a number of different assembly languages in existance, but all of them have a few things in common. They all map directly to the underlying hardware CPU instruction sets.

CPU instuction set

> is a set of binary code/instruction that the CPU understands. Based on the CPU, the instruction can be one byte, two bytes or longer. The instruction code is usually followed by one or two operands.

Instruction Code	operand 1	operand 2

How many instructions there are depends on the CPU.

Because binary code is difficult to remember, each instruction has as its name a so-called mnemonic. For example 'MOV' can be used for moving instructions.

```
MOV A, 0x0020
```

The above instruction moves the value of register A to the specified address.

A simple assembler will translate the 'MOV A' to its CPU's instruction code.

Assembly languages cannot be assumed to be directly portable to other CPU's. Each CPU has its own assembly language, though CPU's within the same family may support limited portability

Load and Store

These instructions tell the CPU to move data from memory to a CPU's register, or move date from one of the CPU's register to memory.

register

> is a special memory located inside the CPU, where arithmetic operations can be performed.

Arithmetic

Arithmetic operations can be performed using the CPU's registers:

- Increment the value of one of the CPU's registers
- Decrement the value of one of the CPU's registers

- Add a value to the register
- Subtract value from the register
- Multiply the register value
- Divide the register value
- Shift the register value
- Rotate the register value

Jumping

During a jump instruction, the program counter is loaded with a new address that is not necessarily the address of the next sequential instruction. After a jump, the program execution continues from the new location in memory.

Relative jump

: the instruction's operand tells how many bytes the program counter should be increased or decreased.

Absolute jump

: the instruction's operand is copied to the program counter; the operand is an absolute memory address where the execution should continue.

Branching

During a branch, the program counter is loaded with one of multiple new values, depending on some specified condition. A branch is a series of conditional jumps.

Some CPUs have skipping instructions. If a register is zero, the following instruction is skipped, if not then the following instruction is executed, which can be a jumping instruction. So Branching can be done by using skipping and jumping instructions together.

Further reading

- Assembly Language

Microprocessor Design/Design Steps

Microprocessor Design

When designing a new microprocessor or microcontroller unit, there are a few general steps that can be followed to make the process flow more logically. These few steps can be further sub-divided into smaller tasks that can be tackled more easily. The general steps to designing a new microprocessor are:

1. Determine the capabilities the new processor should have.
2. Lay out the datapath to handle the necessary capabilities.
3. Define the machine code instruction format (ISA).
4. Construct the necessary logic to control the datapath.

We will discuss each of these steps below:

Determine Machine Capabilities

Before you start to design a new processor element, it is important to first ask why you are designing it at all. What new thing will your processor do that existing processors cannot? Keep in mind that it is always less expensive to utilize an existing chip than to design and manufacture a new one.

Some questions to start:

1. Is this chip an embedded chip, a general-purpose chip, or a different type entirely?
2. What, if any, are the limitations in terms of resources, price, power, or speed?

With that in mind, we need to ask what our chip will do:

1. Does it have integer, floating-point, or fixed point arithmetic, or a combination of all three?
2. Does it have scalar or vector operation abilities?
3. Is it self-contained, or must it interface with a number of external peripherals?
4. Will it support interrupts? If so, How much interrupt latency is tolerable? How much interrupt-response jitter is tolerable?

We also need to ask ourselves whether the machine will support a wide array of instructions, or if it will have a limited set of instructions. More instructions make the design more difficult, but make programming and using the chip easier. On the other hand, having fewer instructions is easier to design, but can be harder and more costly to program.

Lay out the basic arithmetic operations you want your chip to have:

- Addition/Subtraction
- Multiplication
- Division
- Shifting and Rotating
- Logical Operations: AND, OR, XOR, NOR, NOT, etc.

List other capabilities that your machine has:

- Unconditional jumps
- Conditional Jumps (and what conditions?)
- Stack operations (Push, pop)

Once we know what our chip is supposed to do, it is easer to lay out the framework for our datapath

Design the Datapath

Right off the bat we need to determine what ALU architecture that our processor will use:

- Accumulator
- Stack
- Register
- A combination of the above 3

This decision, more than any other, is going to have the largest effect on your final design. Do not proceed in the design process until you have made this decision. Once you have your ALU architecture, you create your memory element (stack or register file), and you can lay out your ALU.

Create ISA

Once we have our basic datapath, we can start to design our ISA. There are a few things that we need to consider:

1. Is this processor RISC, CISC, or VLIW?
2. How long is a machine word?
3. How do you deal with immediate values? What kinds of instructions can accept immediate values?

Once we have our machine code basics, we frequently need to determine whether our processor will be compatible with higher-level languages. Specifically, are there any instructions that can be used for function call and return?

Determining the length of the instruction word in a RISC is a very important matter, and one that is worth a considerable amount of thought. For additional flexibility you can utilize a variable-length instruction set instead — like most CISC machines — at the expense of additional—and more complicated—instruction decode logic. If the instruction word is too long, programmers will be able to fit fewer instructions into memory. If the instruction word is too small, there will not be enough room for all the necessary information. On a desktop PC with several megabytes or even gigabytes of RAM, large instruction words are not a big problem. On an embedded system however, with limited program ROM, the length of the instruction word will have a direct effect on the size of potential programs, and the usefulness of the chips.

Each instruction should have an associated opcode, and typically the length of the opcode field should be constant for all instructions, to reduce complexity of the decoder. The length of the opcode field will directly impact the number of distinct instructions that can be implemented. if the opcode field is too small, you won't have enough room to designate all your instructions. If your opcode is too large, you will be wasting precious bits in your instruction word.

Some instructions will need to be larger than others. For instance, instructions that deal with an immediate value, a memory location, or a jump address are typically larger than instructions that only deal with registers. Instructions that deal only with registers, therefore, will have additional space left over that can be used as an extension to the opcode field.

Example: MIPS R-Type

In the MIPS architecture, instructions that only deal with registers are called **R type** instructions. With 32 registers, a register address is only 5 bits wide. The MIPS opcode is 6 bits wide. With the opcode and the three register addresses (two source and 1 destination register), an R-type instruction only uses 21 out of the 32 bits available.

The additional 11 bits are broken into two additional fields: **Shamt**, a 5 bit immediate value that controls the amount of places shifted by a shift or rotate instruction, and **Func**. Func is a 6 bit field that contains additional information about R-Type instructions. Because of the availability of the Func field, all R-Type instructions share an opcode of 0.

Instruction Set Design

Picking a particular set of instructions is often more an art than a science.

Historically there have been different perspectives on what makes a "good" instruction set.

- The early CISC years focused on making instruction sets that expert assembly language programmers enjoyed programming -- "../code density" was a common metric.
- the early RISC years focused on making instruction sets that ran a few benchmark programs in C, when compiled with relatively primitive compilers, really, really fast -- "cycles per instruction", and later "instructions per cycle" was recognized as an important part of achieving low "time to run the benchmark".
- The rise of multitasking operating systems (and shared-memory parallel processors) lead to the discovery of non-blocking synchronization and the instructions necessary to support it.
- CPUs dedicated to a single application (ASICs or FPGAs) led to the idea of customizing the CPU for one particular application[1]
- The rise of viruses and other malware led to the recognition of the Popek and Goldberg virtualization requirements.

Build Control Logic

Once we have our datapath and our ISA, we can start to construct the logic of our primary control unit. These units are typically implemented as a finite state machine, and we can try to map the ISA to the control unit in a logical way.

design the address path

If a simple virtual==physical address path is adequate for your CPU, you can skip this section.

Most processors have a very simple address path -- address bits come from the PC or some other programmer-visible register, or directly from some instruction, and they are directly applied to the address bus.

Many general-purpose processors have a more complex address path: user-level programs run as if they have a simple address path, but the physical address applied to the address bus is significantly different than the programmer-visible address. If your CPU needs to do this, then you need something to translate user-visible addresses to physical address -- either design the CPU to connect to some off-chip bank register or MMU (such as the 8722 MMU or the 68851 MMU) or design in an on-chip bank register or MMU.

You may want to do this in order to:

- support various debug tools that trap on reads or writes to selected addresses.
- allow access to more RAM (wider physical address) than the user-level address seems to support (banking)
- support many different programs all in different physical RAM locations, even though they were all compiled to run at location 0x300.
- allow a program to successfully read and write a large block of data using normal LOAD and STORE instructions as if it were all in RAM, even though the machine doesn't have that much RAM (paging with virtual memory)
- support a "protected" supervisor-level system that can run buggy or malicious user-level code in an isolated sandbox at full speed without damaging other user-level programs or the supervisor system itself -- Popek and Goldberg virtualization, W xor X memory protection, etc.
- or some combination of the above.

Verify the design

People who design a CPU often spend more time on functional verification than all other steps combined.

Further reading

- Kong and Patterson. "Instruction set design". 1995.[2]

References

[1] "Generating instruction sets and microarchitectures from applications" (http://portal.acm.org/citation.cfm?id=191326.191501) by Ing-Jer Huang, and Alvin M. Despain

[2] http://www.cs.berkeley.edu/~pattrsn/152/lec3.ps

Microprocessor Components

Microprocessor Design/Basic Components

Microprocessor Design

Basic Components

There are a number of components in a common microprocessor that designers should be familiar with before attempting a design. For an overview of these components, see Digital Circuits.

Registers

A register is a storage element typically composed of an array of flip-flops. A 1-bit register can store 1 bit, and a 32-bit register can hold 32 bits, etc. Registers can be any length.

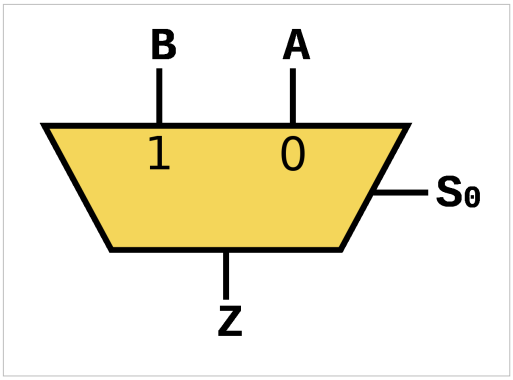
A register has two inputs, a data input and a clock input. The clock input is typically called the "enable". When the enable signal is high, the register stores the data input. When the clock signal is low, the register value stays the same.

Register File

A register file is a whole collection of registers, typically all of which are the same length. A register file takes three inputs, an index address value, a data value, and an enable signal. A signal **decoder** is used to pass the data value from the register file input to the particular register with the specified address.

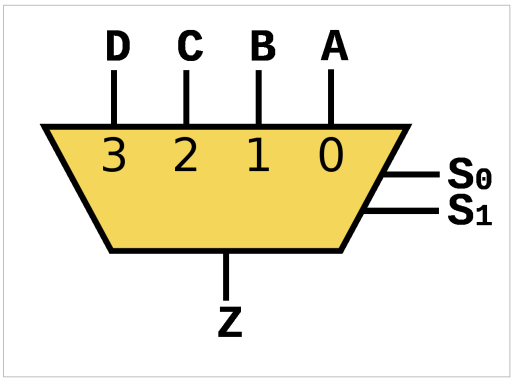
Multiplexers

A multiplexer is an input selector. A multiplexer has 1 output, a control input, and several data inputs. For ease, we number multiplexer inputs from zero, at the top. If the control signal is "0", the 0th input is moved to the output. If the control signal is "3", the 3rd input is moved to the output.

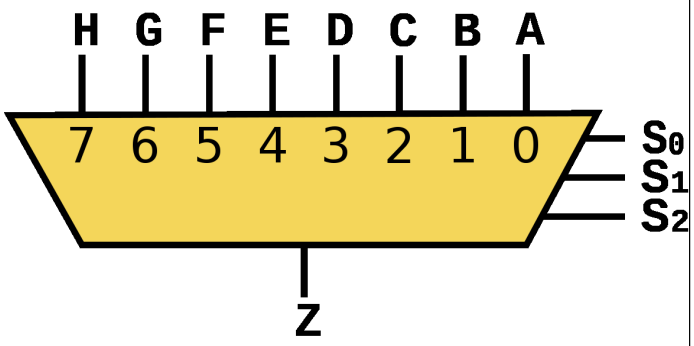


A multiplexer with N control signal bits can support 2^N inputs. For example, a multiplexer with 3 control signals can support $2^3 = 8$ inputs.

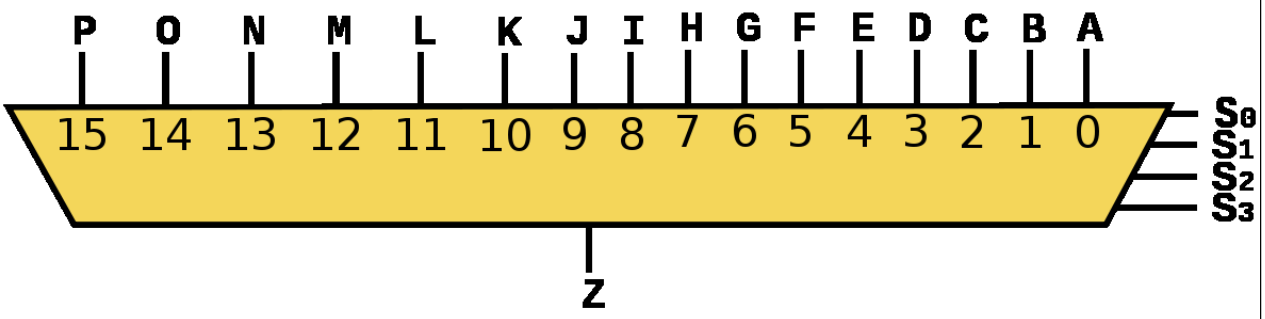
Multiplexers are typically abbreviated as "MUX", and will be abbreviated as such throughout the rest of this book.



A 4 input Multiplexer with 2 control signal wires



An 8 input Multiplexer with 3 control signal wires



A 16 input Multiplexer with 4 control wires

There can be decoders implemented in the components. Decoder

```
Decoder ( inverse functionality of Encoder) can have multiple inputs and depending upon the inputs one of the output signals can go high.
```

For a 2 input decoder there will be 4 output signals.

```
       /|- O0
 i0---|  |- O1
 i1---|  |- O2
       \|- O3
```

```
suppose input i is having value 00  then output signal O0 will go high and remaining other threee lines O1 to O3 will be low.

 In same fashion if i is having value 2 then output O2 will be high and remaining other three lines will be low.
```

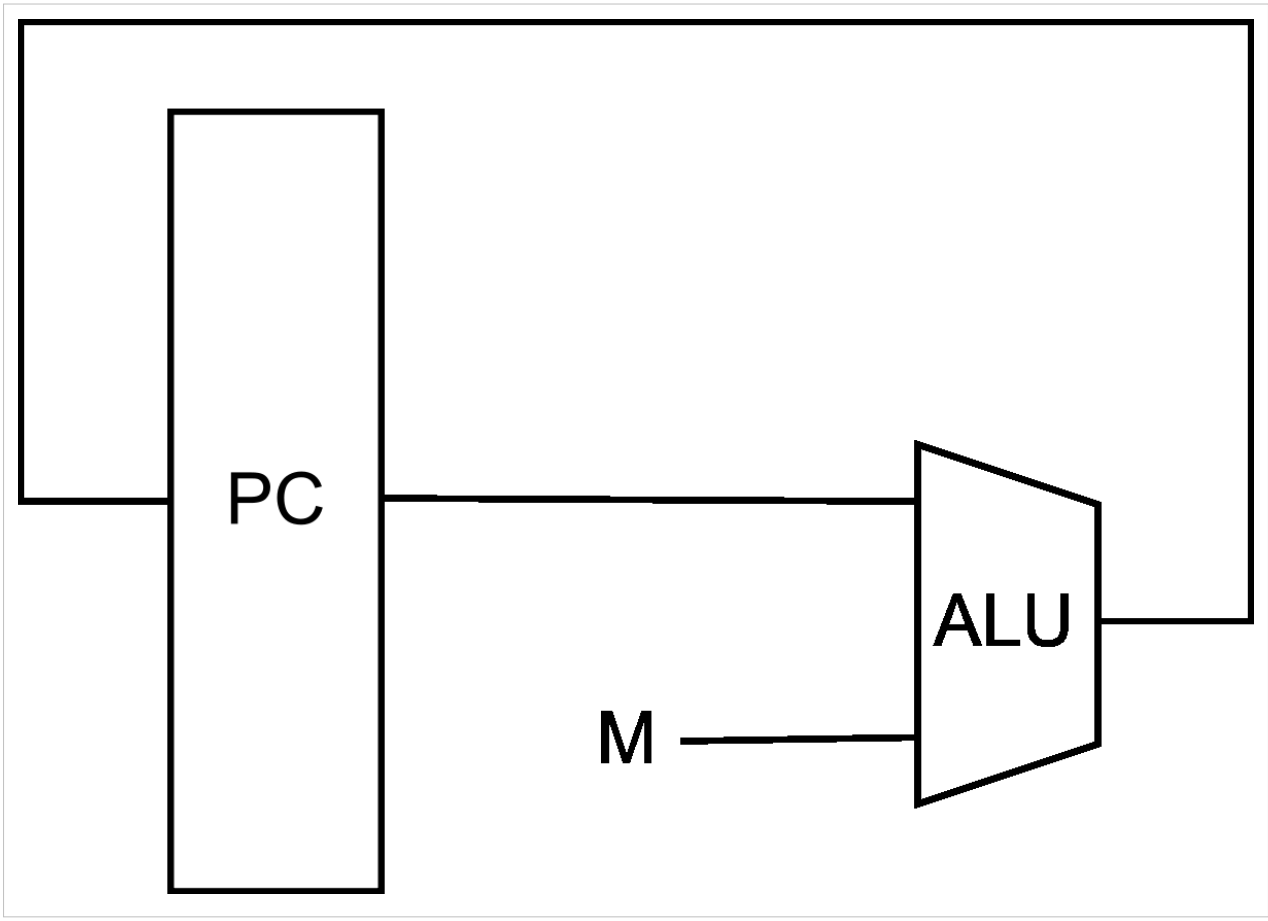
Microprocessor Design/Program Counter

The **Program Counter** (PC) is a register structure that contains the address pointer value of the current instruction. Each cycle, the value at the pointer is read into the instruction decoder and the program counter is updated to point to the next instruction. For RISC computers updating the PC register is as simple as adding the machine word length (in bytes) to the PC. In a CISC machine, however, the length of the current instruction needs to be calculated, and that length value needs to be added to the PC.

Updating the PC

The PC can be updated by making the enable signal high. Each instruction cycle the PC needs to be updated to point to the next instruction in memory. It is important to know how the memory is arranged before constructing your PC update circuit.

Harvard-based systems tend to store one machine word per memory location. This means that every cycle the PC needs to be incremented by 1. Computers that share data and instruction memory together typically are *byte addressable*, which is to say that each byte has its own address, as opposed to each machine word having its own address. In these situations, the PC needs to be incremented by the number of bytes in the machine word.



In this image, the letter *M* is being used as the amount by which to update the PC each cycle. This might be a variable in the case of a CISC machine.

Example: MIPS

The MIPS architecture uses a byte-addressable instruction memory unit. MIPS is a RISC computer, and that means that all the instructions are the same length: 32-bits. Every cycle, therefore, the PC needs to be incremented by 4 (32

bits = 4 bytes).

Example: Intel IA32

The Intel IA32 (better known by some as "x86") is a CISC architecture, which means that each instruction can be a different length. The Intel memory is byte-addressable. Each cycle the instruction decoder needs to determine the length of the instruction, in bytes, and it needs to output that value to the PC. The PC unit increments itself by the value received from the instruction decoder.

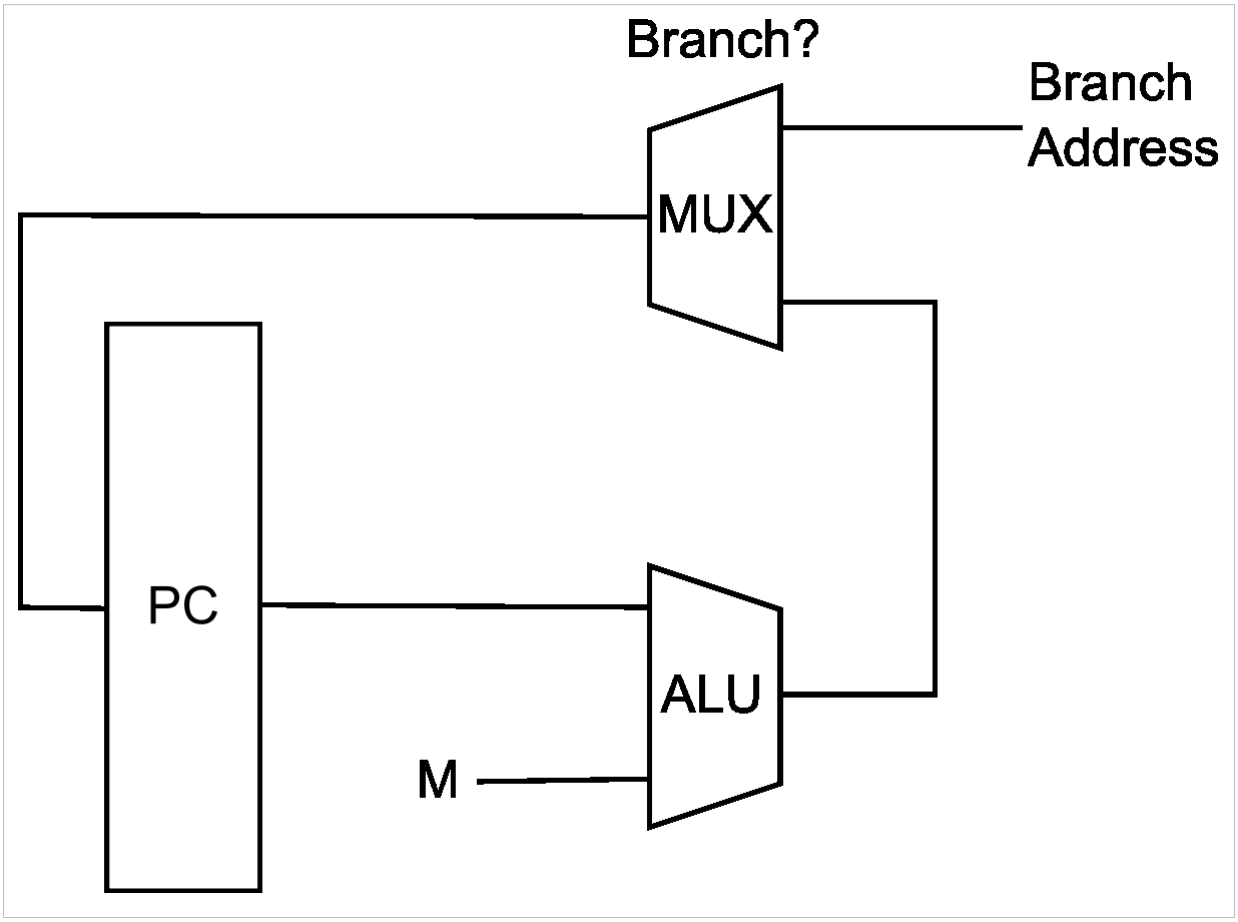
Branching

Branching occurs at one of a set of special instructions known collectively as "branch" or "jump" instructions. In a branch or a jump, control is moved to a different instruction at a different location in instruction memory.

During a branch, a new address for the PC is loaded, typically from the instruction or from a register. This new value is loaded into the PC, and future instructions are loaded from that location.

Non-Offset Branching

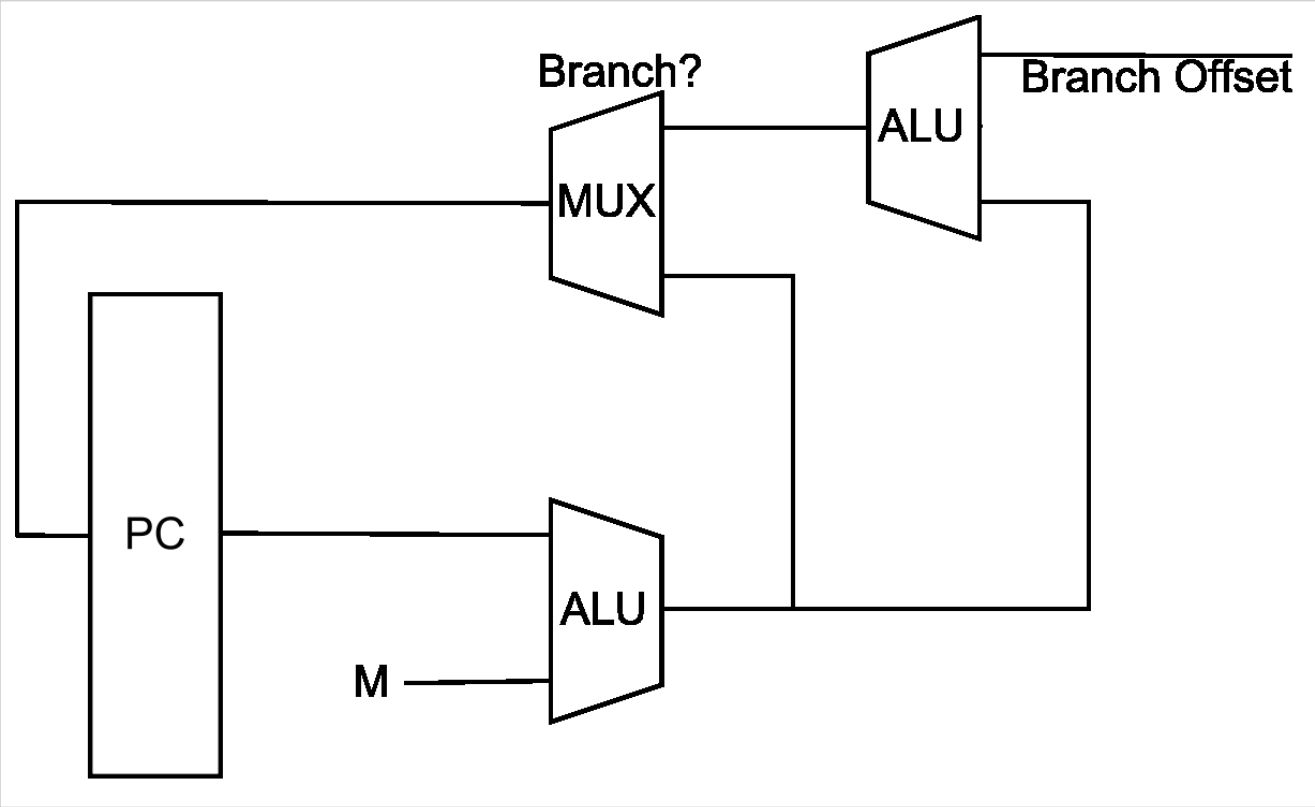
A non-offset branch, frequently referred to as a "jump" is a branch where the previous PC value is discarded and a new PC value is loaded from an external source.



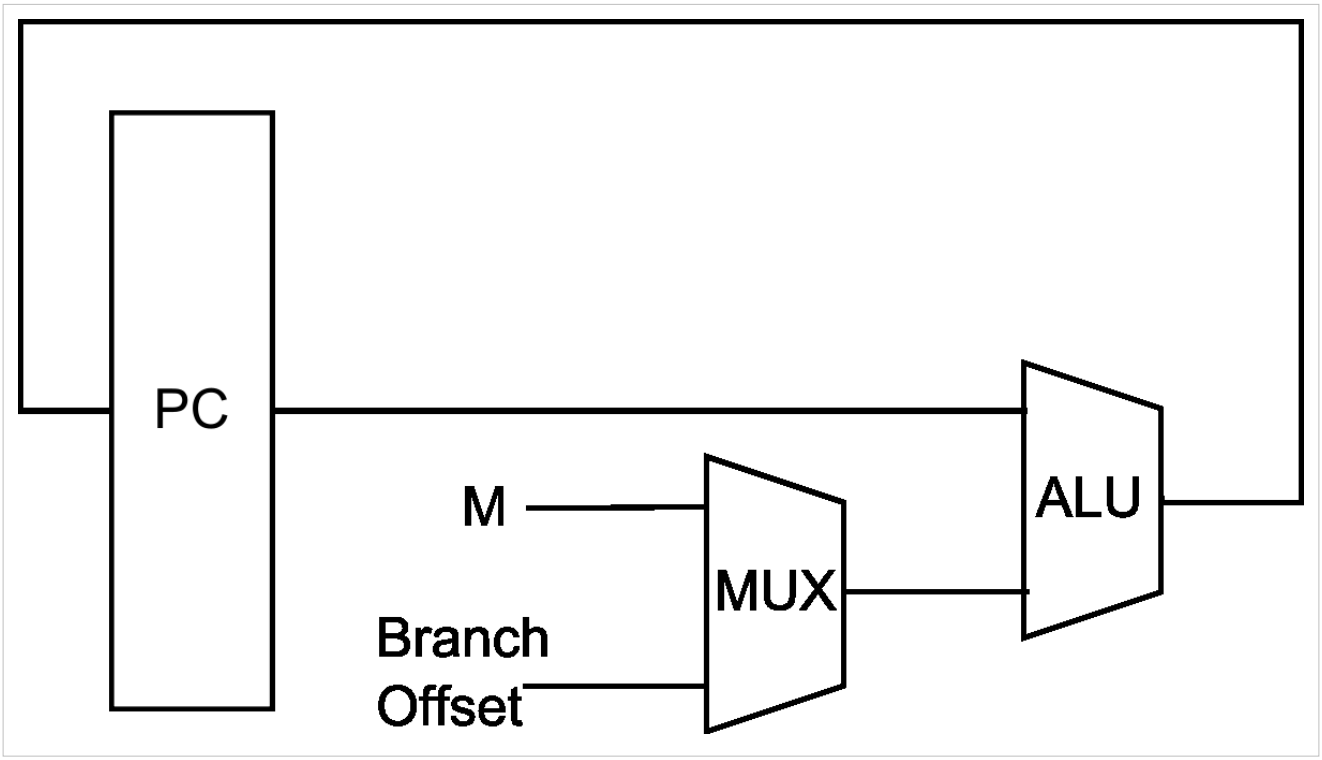
In this image, the PC value is either loaded with an updated version of itself, or else it is loaded with a new *Branch Address*. For simplification we do not show the control signals to the MUX.

Offset Branching

An offset branch is a branch where a value is added (or subtracted) to the current PC value to produce the new value. This is typically used in systems where the PC value is larger then a register value or an immediate value, and it is not possible to load a complete value into the PC. It is also commonly used to support relocatable binaries which may be loaded at an arbitrary base address.



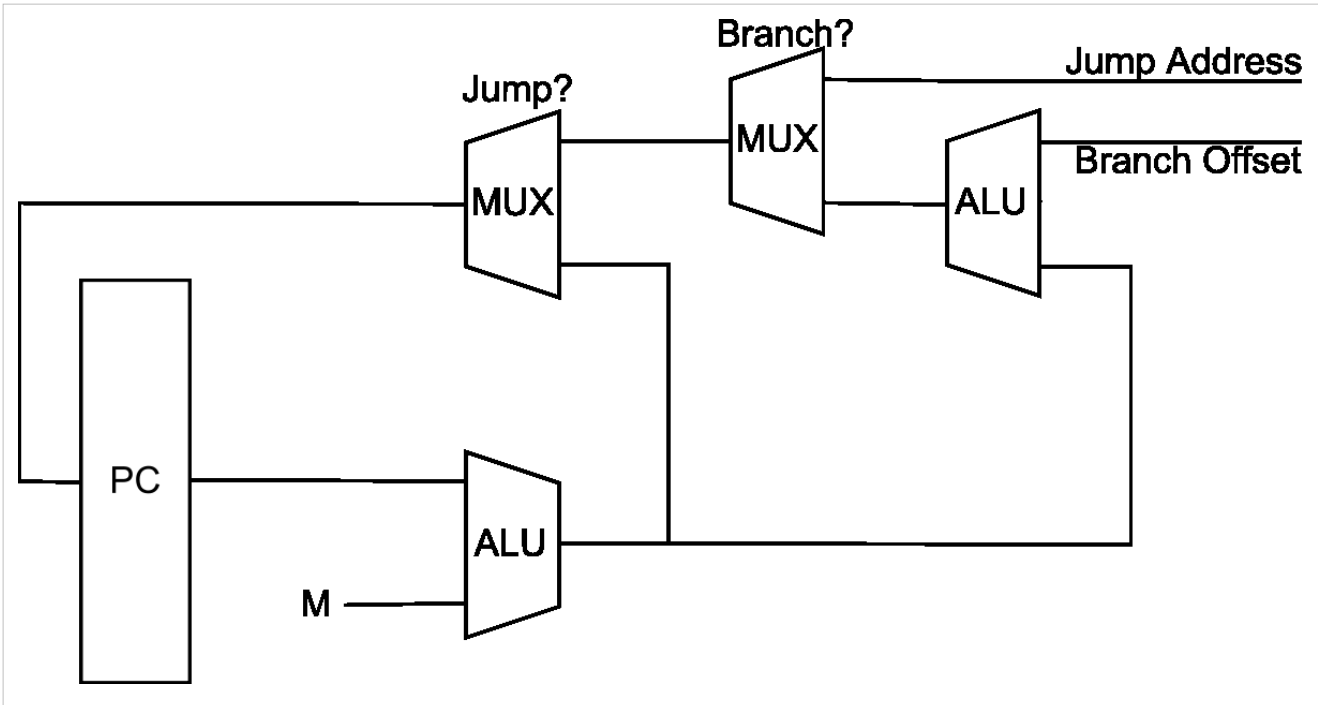
In this image there is a second ALU unit. Notice that we could simplify this circuit and remove the second ALU unit if we use the configuration below:



These are just two possible configurations for this circuit.

Offset and Non-Offset Branching

Many systems have capabilities to use both offset and non-offset branching. Some systems may differentiate between the two as "far jump" and "near jump", respectively, although this terminology is archaic.



Microprocessor Design/Instruction Decoder

Microprocessor Design

The **Instruction Decoder** reads the next instruction in from memory, and sends the component pieces of that instruction to the necessary destinations.

RISC Instruction Decoder

The RISC instruction decoder is typically a very simple device. Because RISC instruction words are a fixed length, the positions of the fields are fixed. We can decode an instruction, therefore, by simply separating the machine word into small parts using wire slices.

CISC Instruction Decoder

Decoding a CISC instruction word is much more difficult than the RISC case, and the increased complexity of the decoder is a common reason that people cite when they choose to use RISC over CISC in their designs.

A CISC decoder is typically set up as a state machine. The machine reads the opcode field to determine what type of instruction it is, and where the other data values are. The instruction word is read in piece by piece, and decisions are made at each stage as to how the remainder of the instruction word will be read.

Microprocessor Design/Register File

Microprocessor Design

Registers are temporary storage locations inside the CPU that hold data and addresses.

The register file is the component that contains all the general purpose registers of the microprocessor. A few CPUs also place special registers such as the PC and the status register in the register file. Other CPUs keep them separate.

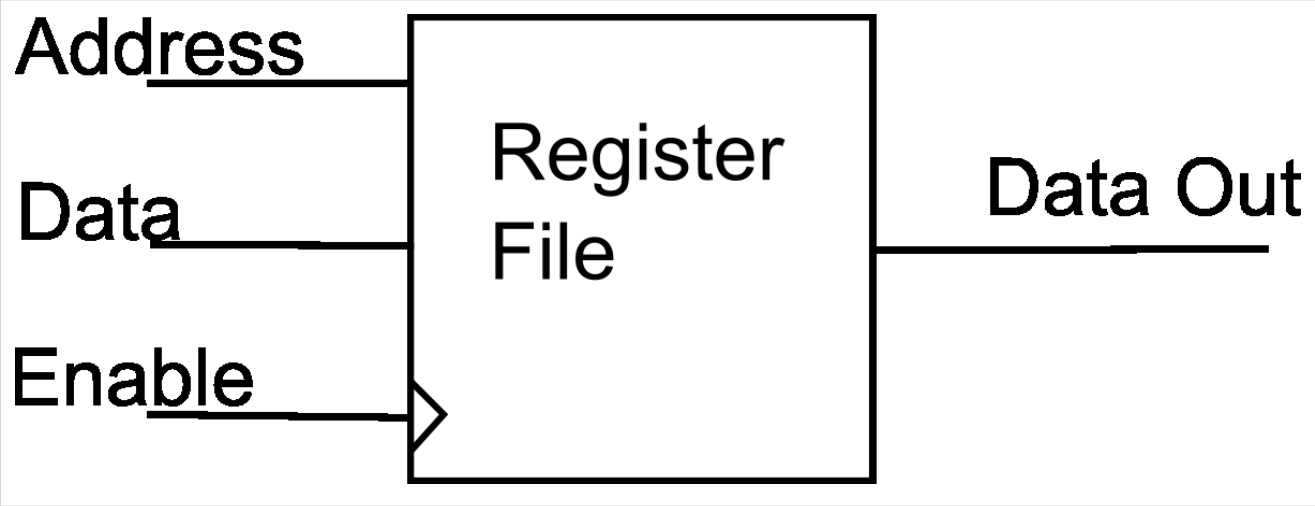
When designing a CPU, some people distinguish between "architectural features" and the "implementation details". The "architectural features" are the programmer visible parts; if someone makes a new system where any of these parts are different from the old CPU, then suddenly all the old software won't work on the new CPU. The "implementation details" are the parts that, although we put even more time and effort into getting them to work, one can make a new system that has a different way of implementing them, and still keep software compatibility -- some programs may run a little faster, other programs may run a little slower, but they all produce the same results as on the earlier machine.

The programmer-visible register set has the biggest impact on software compatibility of any other part of the datapath, and perhaps of any other part in the entire computer. The architectural features of the programmer-visible register set are the number of registers, the number of bits in each register, and the logical organization of the registers. Assembly language programmers like to have many registers. Early microprocessors had painfully few registers, limited by the area of the chip. Today, many chips have room for huge numbers of registers, so the number of programmer-registers is limited by other constraints: More programmer-visible registers requires bigger operand fields. More programmer-visible registers requires more time saving and restoring registers on an interrupt or context switch. Software compatibility requires keeping exactly the same number, size, and organization of programmer-visible registers. Assembly language programmers like a "flat" address space, where the full address of any location in (virtual) memory fits in a single address register. And so the amount of (virtual) memory desired by an architect sets a minimum width to each address register. [1]

The idea of "general registers" -- a group of registers, any one of which can, at different times, operate as a stack pointer, index register, accumulator, program counter, etc. was invented around 1971.[2]

Register File

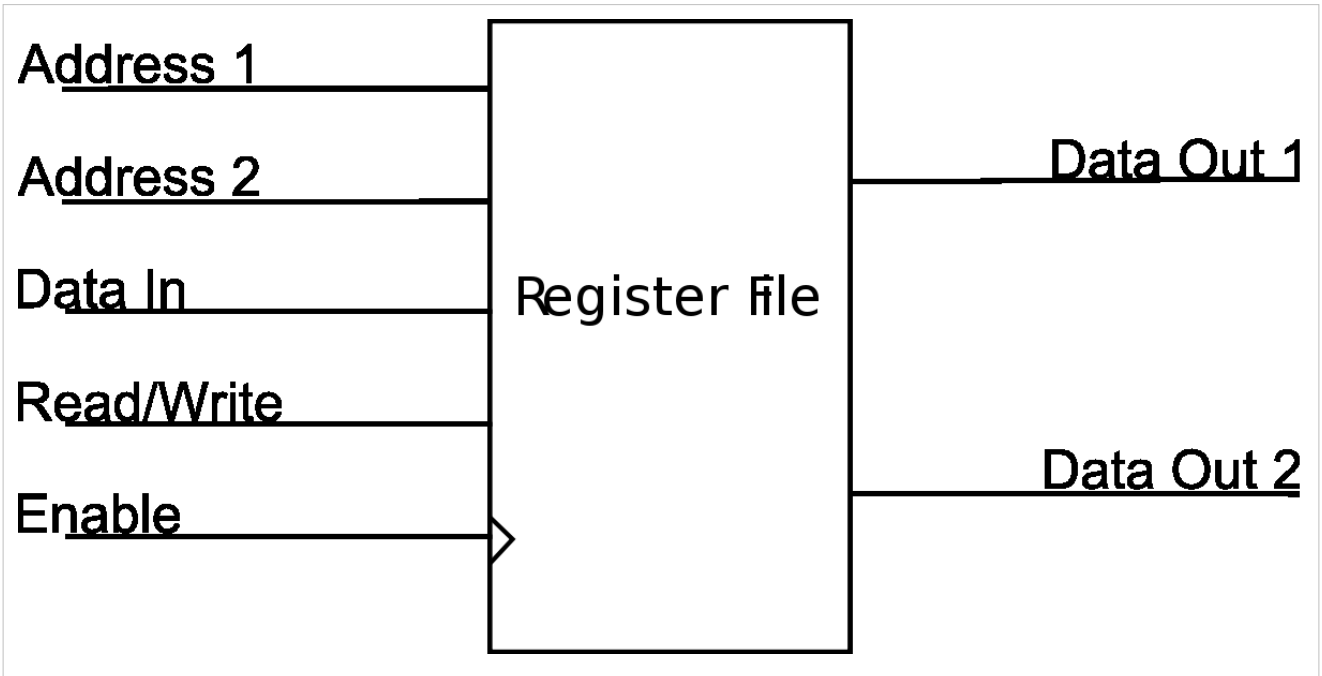
A simple register file is a set of registers and a decoder. The register file requires an address and a data input.



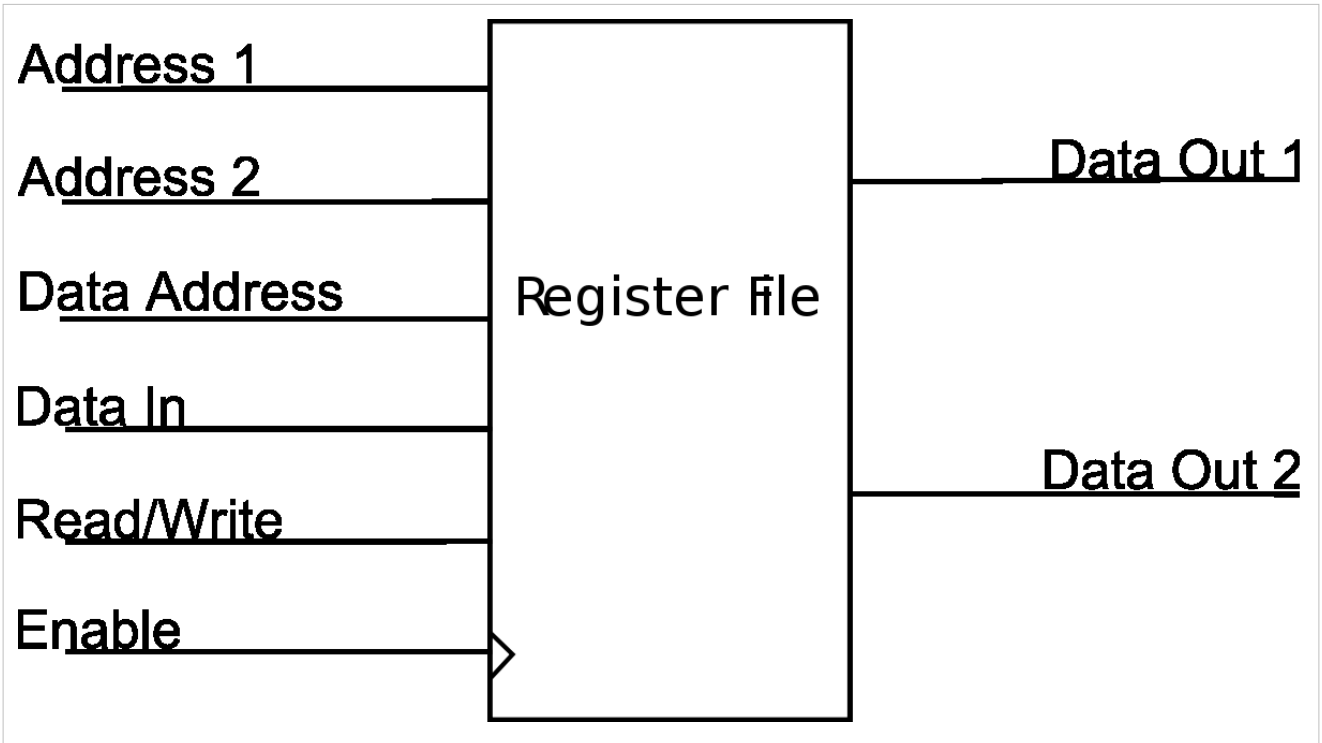
However, this simple register file isn't useful in a modern processor design, because there are some occasions when we don't want to write a new value to a register. Also, we typically want to read two values at once and write one value back in a single cycle. Consider the following equation:

$$C = A + B$$

To perform this operation, we want to read two values from the register file, *A* and *B*. We also have one result that we want to write back to the register file when the operation has completed. For cases where we do not want to write any value to the register file, we add a control signal called *Read/Write*. When the control signal is high, the data is written to a register, and when the control signal is low, no new values are written.



In this case, it is likely advantageous for us to specify a third address port for the write address:



More registers than you can shake a stick at

Consider a situation where the machine word is very small, and therefore the available address space for registers is very limited. If we have a machine word that can only accommodate 2 bits of register address, we can only address 4 registers. However, register files are small to implement, so we have enough space for 32 registers. There are several solutions to this dilemma -- several ways of increasing performance by using many registers, even though we don't quite have enough bits in the instruction word to directly address all of them.

Some of those solutions include:

- special-purpose registers that are always used for some specific instruction, and so that instruction doesn't need any bits to specify that register.
  - In almost every CPU, the program counter PC and the status register are treated differently than the other registers, with their own special set of instructions.
- separating registers into two groups, "address registers" and "data registers", so an instruction that uses an address needs enough bits to select one out of all the address registers, which is 1 less bit than one out of every register.
- register windowing as on SPARC

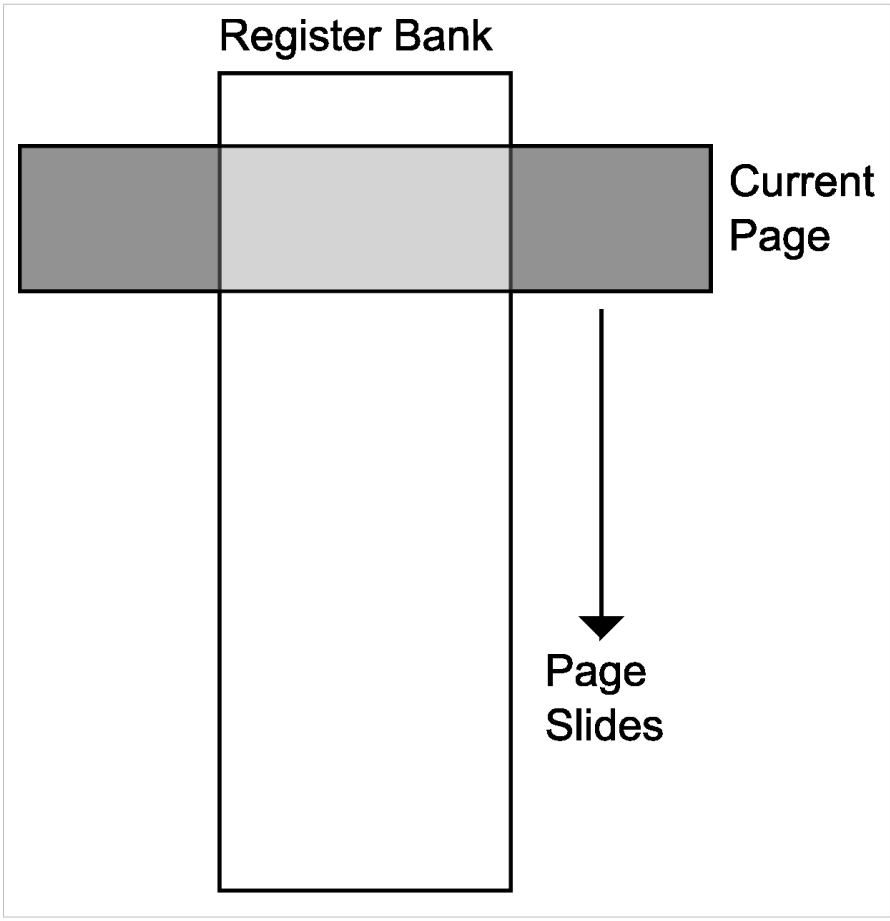
[1] and

- using a "register bank".

Register Bank

Consider a situation where the machine word is very small, and therefore the available address space for registers is very limited. If we have a machine word that can only accommodate 2 bits of register address, we can only address 4 registers. However, register files are small to implement, so we have enough space for 32 registers. The solution to this dilemma is to utilize a **register bank** which consists of a series of register files combined together.

A **register bank** contains a number of register files or *pages*. Only one page can be active at a time, and there are additional instructions added to the ISA to switch between the available register pages. Data values can only be written to and read from the currently active register page, but instructions can exist to move data from one page to another.

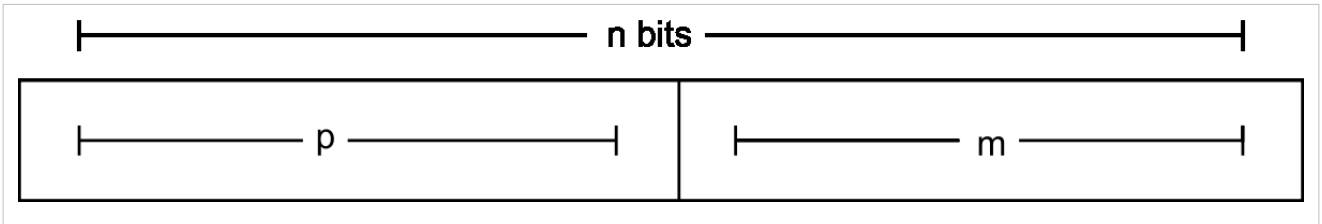


As can be seen in this image, the gray box represents the current page, and the page can be moved up and down on the register bank.

If the register bank has *N* registers, and a page can only show *M* registers (with $N > M$), we can address registers with two values, *n* and *m* respectively. We can define these values as:

$$n = \log_2(N)$$
$$m = \log_2(M)$$

In other words, *n* and *m* are the number of bits required to address *N* and *M* registers, respectively. We can break down the address into a single value as such:



Where *p* is the number of bits reserved to specify the current register page. As we can see from this graphic, the current register address is simply the concatenation of the page address and the register address.

References

[1] "Computer architecture: fundamentals and principles of computer design" (http://books.google.com/books?id=ZWaUurOwMPQC&pg=PA112&lpg=PA112&dq=insufficient+address+computer+architecture&source=bl&ots=Ak4ghlsMBy&sig=dqDtvlQA3fyPTSqQGfxwzz2lgio&hl=en&ei=N9n3SYO7BI3uMsPvyKkP&sa=X&oi=book_result&ct=result&resnum=3#v=onepage&q=&f=false) by Joseph D. Dumas 2006 page 111.

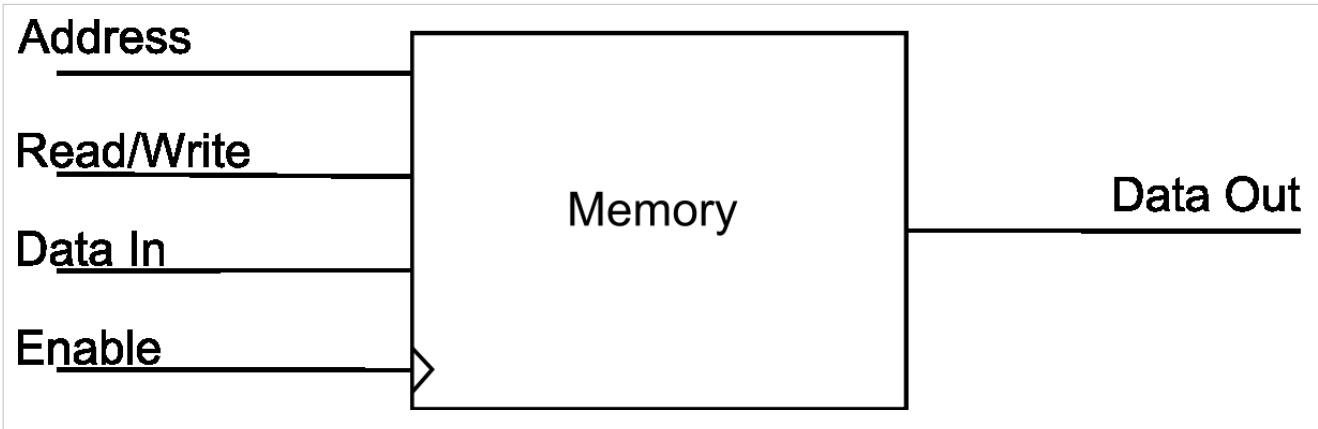
[2] "general registers" were invented by C. Gordon Bell and Allen Newell as they were working on their book, *Computer Structures: Readings and Examples* (1971). -- Frederik Nebeker. "More Treasured Texts" article. "IEEE Spectrum" 2003 July.

Microprocessor Design/Memory Unit

Microprocessor Design

Microprocessors rely on memory for storing the instructions and the data used by software programs. The memory unit is responsible for communicating with the system memory.

Memory Unit



Actions of the Memory Unit

All von Neumann CPUs store their instructions in memory.

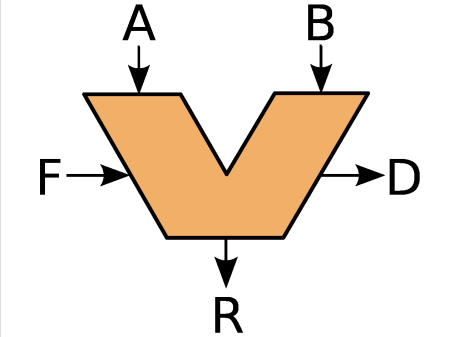
In a Harvard architecture, the data memory unit and the instruction memory unit are two different units. However, in a Princeton architecture the two memory units are combined into a single module. Most modern PC computer systems are Princeton, not Harvard, so the memory unit must handle all instruction and data transactions. This can serve as a bottleneck in the design.

Timing Issues

The memory unit is typically one of the slowest components of a microcontroller, because the external interface with RAM is typically much slower than the speed of the processor.

Microprocessor Design/ALU

Microprocessor Design



Microprocessors tend to have a single module that performs arithmetic operations on integer values. This is because many of the different arithmetic and logical operations can be performed using similar (if not identical) hardware. The component that performs the arithmetic and logical operations is known as the **Arithmetic Logic Unit**, or ALU.

The ALU is one of the most important components in a microprocessor, and is typically the part of the processor that is designed first. Once the ALU is designed, the rest of the microprocessor is implemented to feed operands and control codes to the ALU.

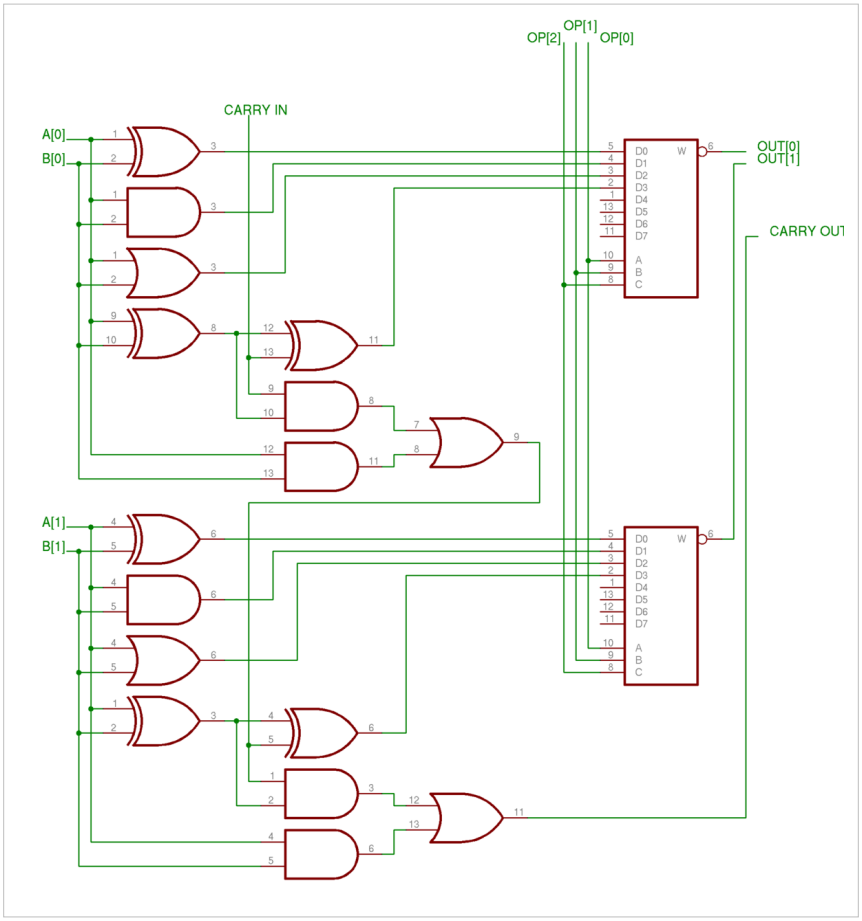
Tasks of an ALU

ALU units typically need to be able to perform the basic logical operations (AND, OR), including the addition operation. The inclusion of inverters on the inputs enables the same ALU hardware to perform the subtraction operation (adding an inverted operand), and the operations NAND and NOR.

A basic ALU design involves a collection of "ALU Slices", which each can perform the specified operation on a single bit. There is one ALU slice for every bit in the operand.

Example: 2-Bit ALU

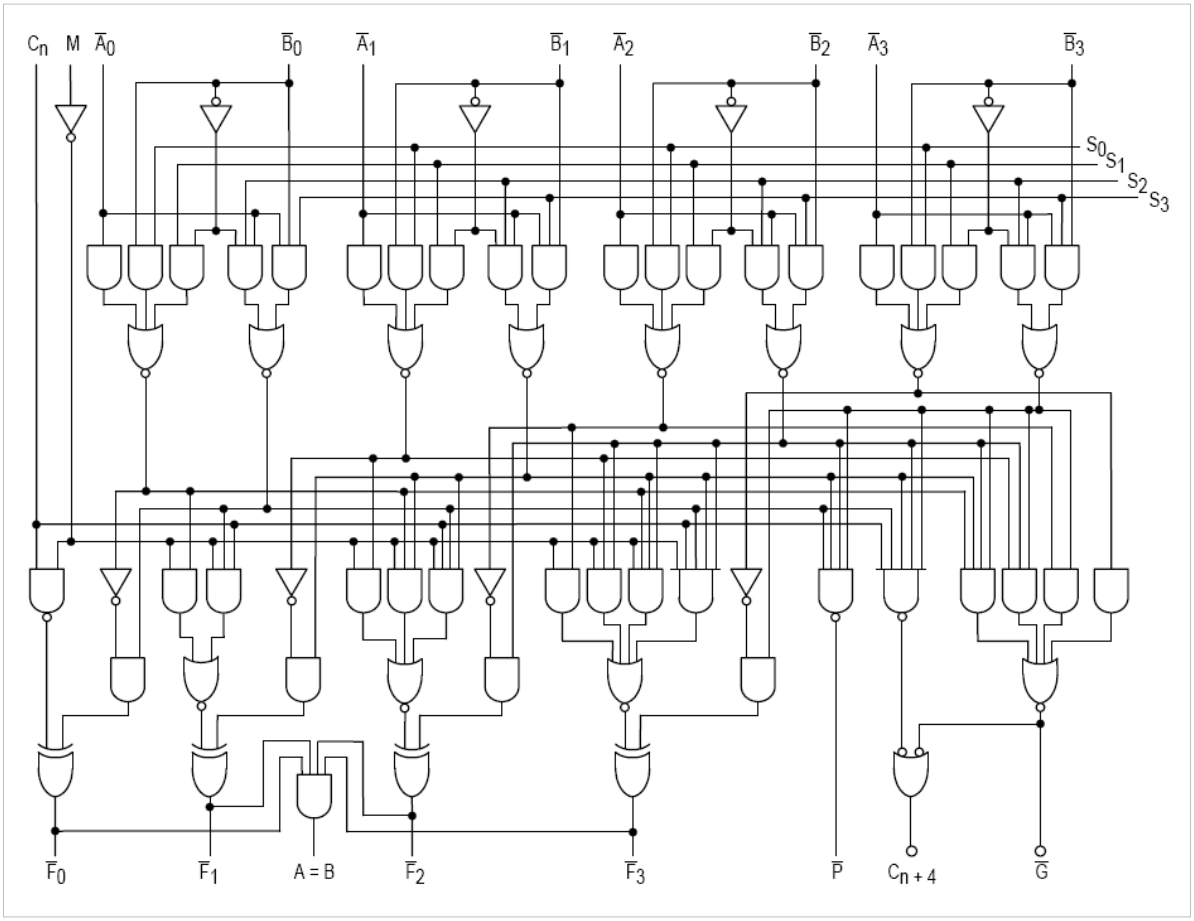
This is an example of a basic 2-bit ALU. The boxes on the right hand side of the image are multiplexers and are used to select between various operations: OR, AND, XOR, and addition.



Notice that all the operations are performed in parallel, and the select signal ("OP") is used to determine which result to pass on to the rest of the datapath. Notice that the carry signal, which is only used for addition, is generated and passed out of the ALU for every operation, so it is important that if we aren't performing addition that we ignore the carry flag.

Example: 4-Bit ALU

Here is a circuit diagram of a 4 bit ALU.



Additional Operations

Logic and addition are some of the easiest, but also the most common operations. For this reason, typical ALUs are designed to handle these operations specially, and other operations, such as multiplication and division, are handled in a separate module.

Notice also that the ALU units that we are discussing here are only for integer datatypes, not floating-point data. Luckily, once integer ALU and multiplier units have been designed, those units can be used to create floating-point units (FPU).

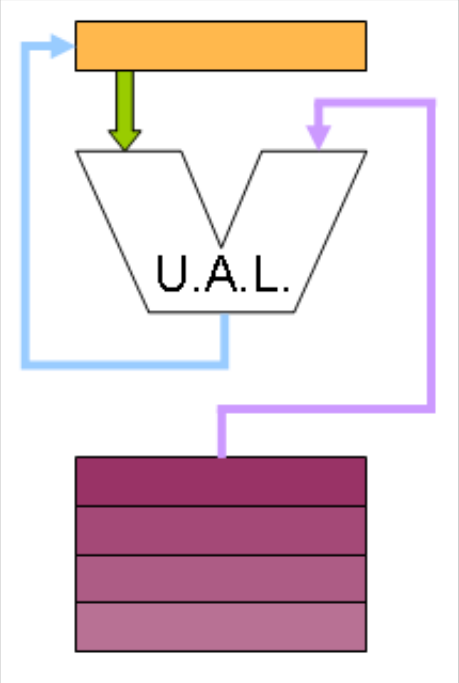
ALU Configurations

Once an ALU is designed, we need to define how it interacts with the rest of the processor. We can choose any one of a number of different configurations, all with advantages and disadvantages. Each category of instruction set architecture (ISA) -- stack, accumulator, register-memory, or register-register-load-store -- requires a different way of connecting the ALU. [1] In all images below, the orange represents memory structures internal to the CPU (registers), and the purple represents external memory (RAM).

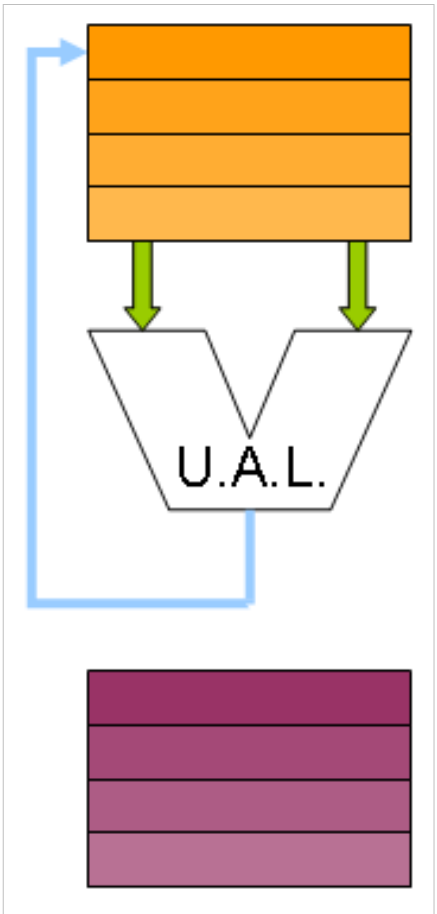
Accumulator

An accumulator machine has one special register, called the accumulator. The accumulator stores the result of every ALU operation, and is also one of the operands to every instruction. This means that our ISA can be less complicated, because instructions only need to specify one operand, instead of two operands and a destination. Accumulator architectures have simple ISAs and are typically very fast, but additional software needs to be written to load the accumulator with proper values. Unfortunately, accumulator machines are difficult to pipeline.

One example of a type of computer system that is likely to use an accumulator is a common desk calculator.

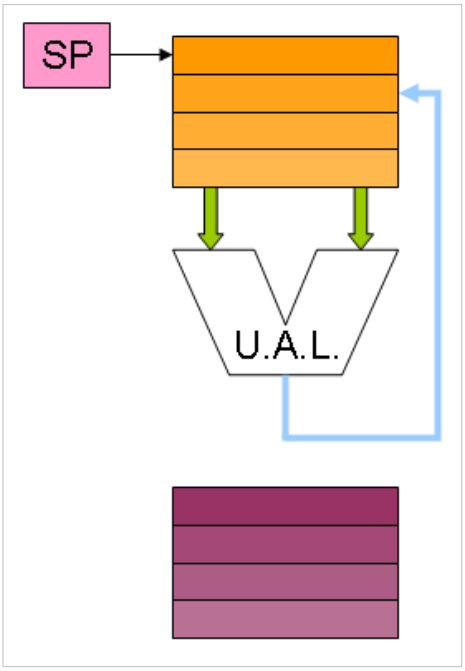


Register-to-Register



One of the more common architectures is a Register-to-register architecture, also called a 3 register operand machine. In this configuration, the programmer can specify both source operands, and a destination register. Unfortunately, the ISA needs to be expanded to include fields for both source operands and the destination operands. This requires longer instruction word lengths, and it also requires additional effort (compared to the accumulator) to write results back to the register file after execution. This write-back step can cause synchronization issues in pipelined processors (we will discuss pipelining later).

Register Stack

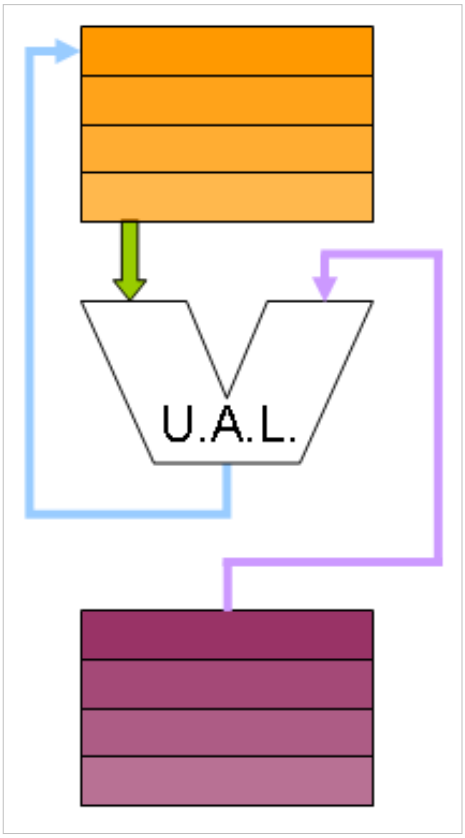


A register stack is like a combination of the Register-to-Register and the accumulator structures. In a register stack, the ALU reads the operands from the top of the stack, and the result is pushed onto the top of the stack. Complicated mathematical operations require decomposition into Reverse-Polish form, which can be difficult for programmers to use. However, many computer language compilers can produce reverse-polish notation easily because of the use of binary trees to represent instructions internally. Also, hardware needs to be created to implement the register stack, including PUSH and POP operations, in addition to hardware to detect and handle stack errors (pushing on a full stack, or popping an empty stack).

The benefit comes from a highly simplified ISA. These machines are called "0-operand" or "zero address machines" because operands don't need to be specified, because all operations act on specified stack locations.

In the diagram at right, "SP" is the pointer to the top of the stack. This is just one way to implement a stack structure, although it might be one of the easiest.

Register-and-Memory



One complicated structure is a Register-and-Memory structure, like that shown at right. In this structure, one operand comes from a register file, and the other comes from external memory. In this structure, the ISA is complicated because each instruction word needs to be able to store a complete memory address, which can be very long. In practice, this scheme is not used directly, but is typically integrated into another scheme, such as a Register-to-Register scheme, for flexibility.

Some CISC architectures have the option of specifying one of the operands to an instruction as a memory address, although they are typically specified as a register address.

Complicated Structures

There are a number of other structures available, some of which are novel, and others are combinations of the types listed above. It is up to the designer to decide exactly how to structure the microprocessor, and feed data into the ALU.

Example: IA-32

The Intel IA-32 ISA (x86 processors) use a register stack architecture for the floating point unit, but it uses a modified Register-to-Register structure for integer operations. All integer operations can specify a register as the first operand, and a register or memory location as the second operand. The first operand acts as an

accumulator, so that the result is stored in the first operand register. The downside to this is that the instruction words are not uniform in length, which means that the instruction fetch and decode modules of the processor need to be very complex.

A typical IA-32 instruction is written as:

```
ADD AX, BX
```

Where **AX** and **BX** are the names of the registers. The resulting equation produces **AX** = **AX** + **BX**, so the result is stored back into **AX**.

Example: MIPS

MIPS uses a Register-to-Register structure. Each operation can specify two register operands, and a third destination register. The downside is that memory reads need to be made in separate operations, and the small format of the instruction words means that space is at a premium, and some tasks are difficult to perform.

An example of a MIPS instruction is:

```
ADD R1, R2, R3
```

Where **R1**, **R2** and **R3** are the names of registers. The resulting equation looks like: **R1** = **R2** + **R3**.

References

[1] "Instruction Set Principles: Basic ISA Classes" (http://users.encs.concordia.ca/~tahar/coen6741/notes/Chapter2-4p.pdf) by Dr. Sofiène Tahar

- Digital Circuits/ALU
- Electronics/ALU

Microprocessor Design/FPU

Similar to the ALU is the **Floating-Point Unit**, or FPU. The FPU performs arithmetic operations on floating point numbers.

An FPU is complicated to design, although the IEEE 754 standard helps to answer some of the specific questions about implementation. It isn't always necessary to follow the IEEE standard when designing an FPU, but it certainly does help.

Floating point numbers

This section is just going to serve as a brief refresher on floating point numbers. For more information, see the Floating Point book.

Floating point numbers are specified in two parts: the exponent (e), and the mantissa (m). The value of a floating point number, v, is generally calculated as:

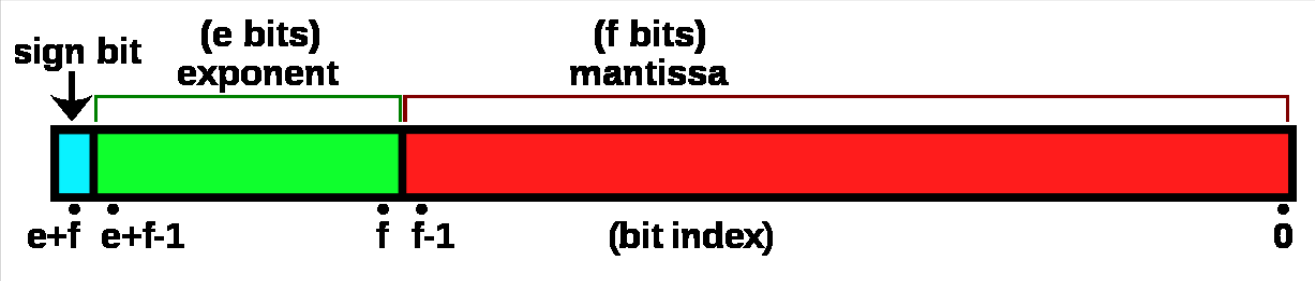
$$v = m \times 2^e$$

IEEE 754

IEEE 754 format numbers are calculated as:

$$v = (1 + m) \times 2^e$$

The mantissa, m, is "normalized" in this standard, so that it falls between the numbers 1.0 and 2.0.



Floating Point Multiplication

Multiplying two floating point numbers is done as such:

$$v_1 \times v_2 = (m_1 \times m_2) \times 2^{(e_1+e_2)}$$

Likewise, division can be performed by:

$$\frac{v_1}{v_2} = \frac{m_1}{m_2} \times 2^{(e_1-e_2)}$$

To perform floating point multiplication then, we can follow these steps:

1. Separate out the mantissa from the exponent
2. Multiply (or divide) the mantissa parts together
3. Add (or subtract) the exponents together
4. Combine the two results into the new value
5. Normalize the result value (optional).

Floating Point Addition

Floating point addition—and by extension, subtraction— is more difficult than multiplication. The only way that floating point numbers can be added together is if the exponents of both numbers are the same. This means that when we add two numbers together, we need first to scale the numbers so that they have the same exponent. Here is the algorithm:

1. Separate the mantissa from the exponent of each number
2. Compare the two exponents, and determine the difference between them.
3. Add the difference to the smaller exponent, to make both exponents the same.
4. Logically right-shift the mantissa of the number with the smaller exponent a number of spaces equal to the difference.
5. Add the two mantissas together
6. Normalize the result value (optional).

Floating Point Unit Design

As we have seen from the two algorithms above, an FPU needs the following components:

For addition/Subtraction

- A comparator (subtracter) to determine the difference between exponents, and to determine the smaller of the two exponents.
- An adder unit to add that difference to the smaller exponent.
- A shift unit, to shift the mantissa the specified number of spaces.
- An adder to add the mantissas together

For multiplication/division

- A multiplier (or a divider) for the mantissa part
- An adder for the exponent prts.

Both operation types require a complex control unit.

Both algorithms require some kind of addition/subtraction unit for the exponent part, so it seems likely that we can use just one component to perform both tasks (since both addition and multiplication won't be happening at the same time in the same unit). Because the exponent is typically a smaller field than the mantissa, we will call this the "Small ALU". We also need an ALU and a multiplier unit to handle the operations on the mantissa. If we combine the two together, we can call this unit the "Large ALU". We can also integrate the fast shifter for the mantissa into the large ALU.

Once we have an integer ALU designed, we can copy those components almost directly into our FPU design.

Further Reading

- Floating Point

Microprocessor Design/Control Unit

Microprocessor Design

The control unit reads the opcode and instruction bits from the machine code instruction, and creates a series of control codes to activate and operate the various components to perform the desired task.

Simple Control Unit

In it's most simple form, a control unit can take the form of a lookup table. The machine word opcode is used as the index into the table, and the various control signals are output to the respective destinations.

Complex Control Unit

A more complex version of a control unit is implemented as a finite state machine (FSM). Multi-cycle, Pipelined, and other advanced processor designs may require an FSM-based control unit.

ALU Design

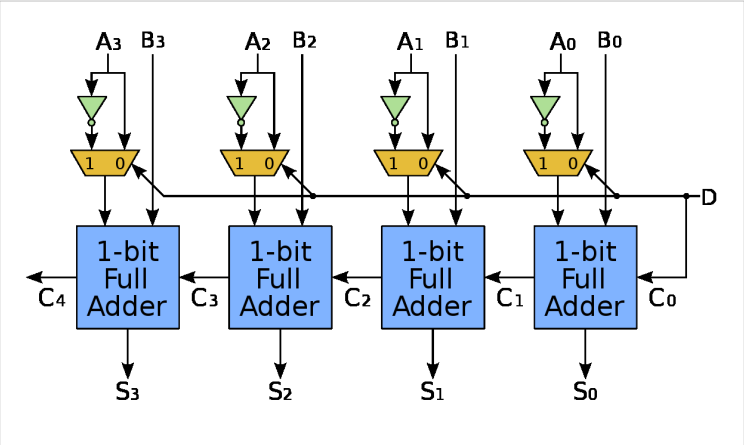
Microprocessor Design/Add and Subtract Blocks

Addition and Subtraction

Addition and subtraction are similar algorithms. Taking a look at subtraction, we can see that:

$$a - b = a + (-b)$$

Using this simple relationship, we can see that addition and subtraction can be performed using the same hardware. Using this setup, however, care must be taken to invert the value of the second operand if we are performing subtraction. Note also that in twos-compliment arithmetic, the value of the second operand must not only be inverted, but 1 must be added to it. For this reason, when performing subtraction, the carry input into the LSB should be a 1 and not a zero.

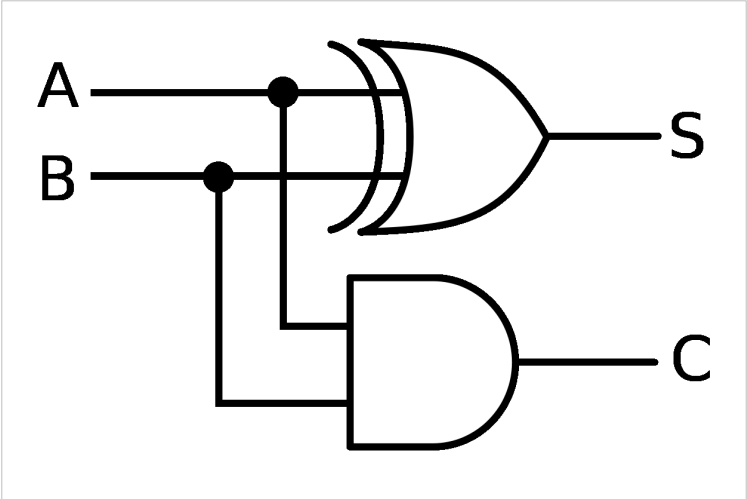


Our goal on this page, then, is to find suitable hardware for performing addition.

Bit Adders

Half Adder

A half adder is a circuit that performs binary addition on two bits. A half adder does not explicitly account for a carry input signal.

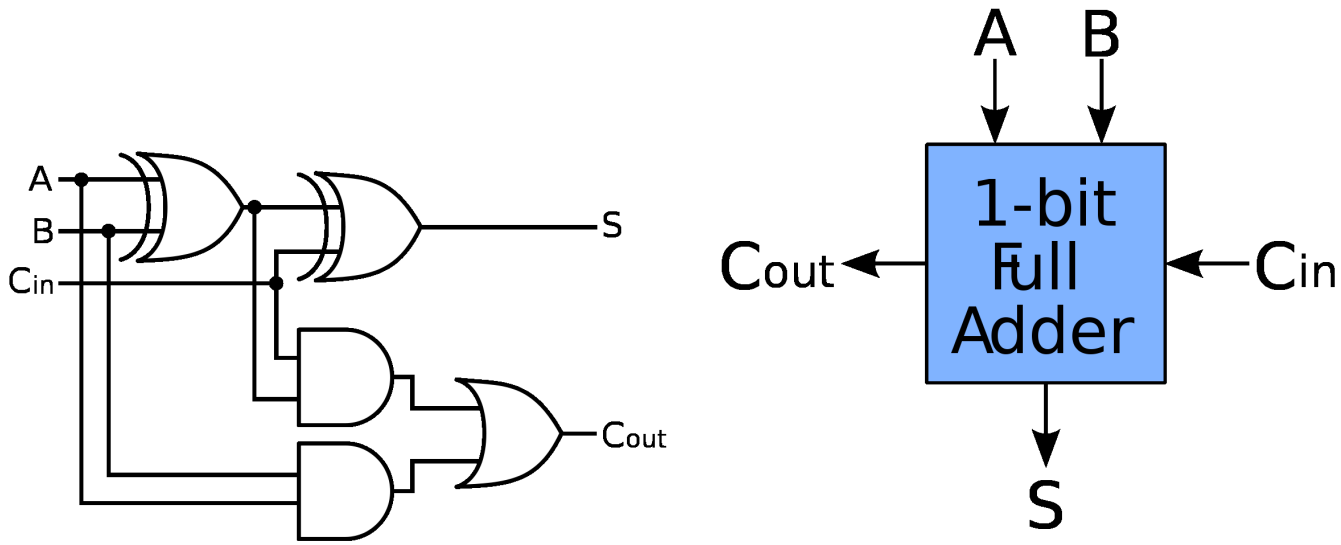


In verilog, a half-adder can be implemented as follows:

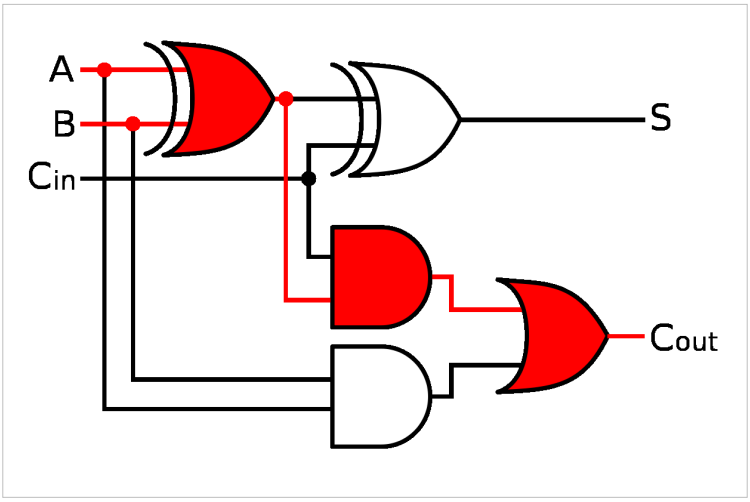
```
module half_adder(a, b, c, s)
   input a, b;
   output s, c;
   s = a ^ b;
   c = a & b;
endmodule
```

Full Adder

Full adder circuits are similar to the half-adder, except that they do account for a carry input and a carry output. Full adders can be treated as a 3-bit adder with a 2-bit result, or they can be treated as a single stage (a 3:2 compressor) in a larger adder.



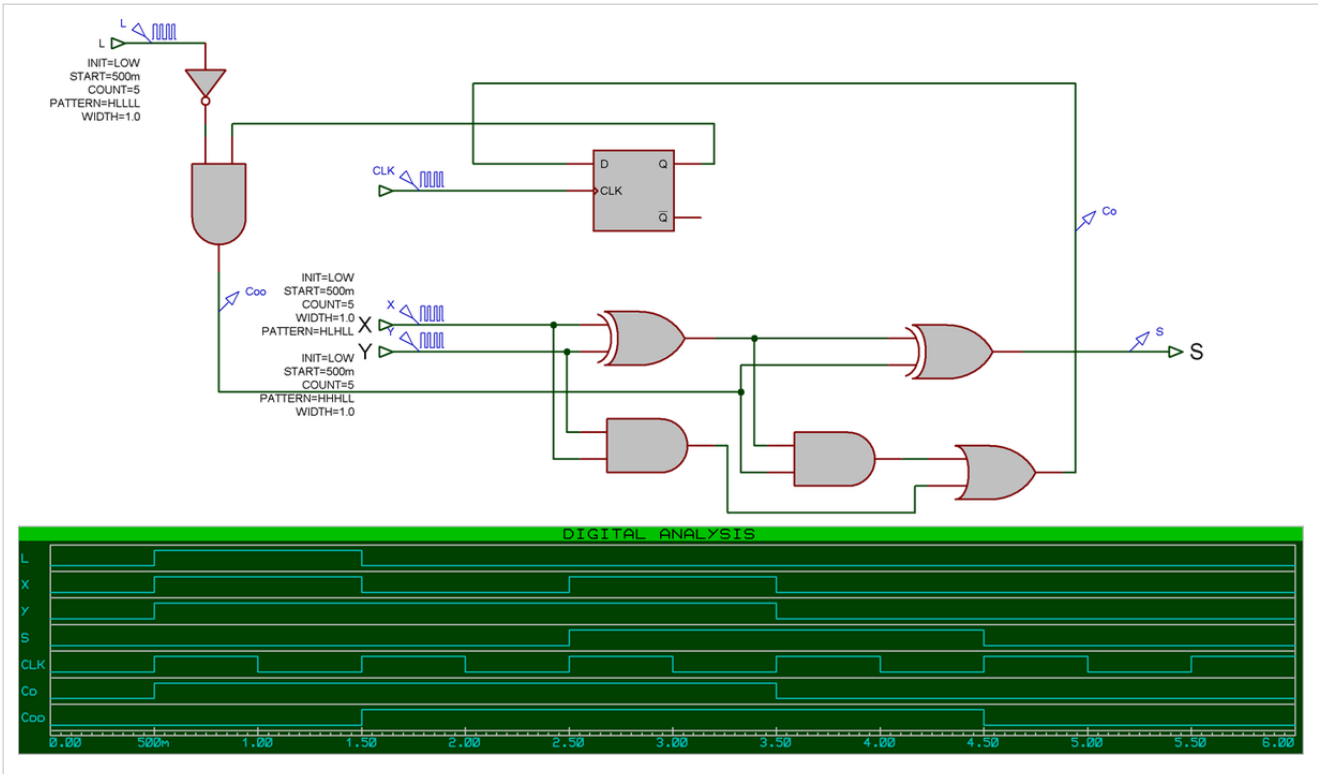
As can be seen below, the number of gate delays in a full-adder circuit is 3:



We can use verilog to implement a full adder module:

```
module full_adder(a, b, cin, cout, s);
   input a, b, cin;
   output cout, s;
   wire temp;
   temp = a ^ b;
   s = temp ^ cin;
   cout = (cin & temp) | (a & b);
endmodule
```

Serial Adder



A serial adder is a kind of ../ALU/ that calculates each bit of the output, one at a time, re-using one full adder (total). This image shows a 2-bit serial adder, and the associated waveforms.

Serial adders have the benefit that they require the least amount of hardware of all adders, but they suffer by being the slowest.

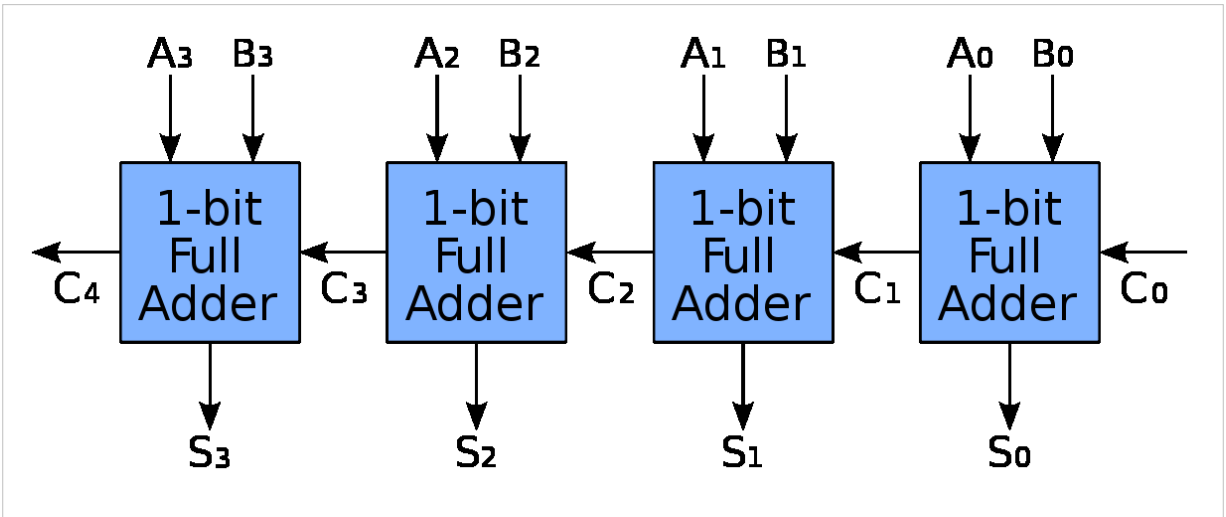
Parallel Adder

A parallel adder is a kind of ../ALU/ that calculates every bit of the output more or less simultaneously, using one full adder for each output bit. The 1947 Whirlwind computer was the first computer to use a parallel adder.

In many CPUs, the CPU latches the final carry-out of the parallel adder in an external "carry flag" in a "status register".

In a few CPUs, the latched value of the carry flag is always wired to the first carry-in of the parallel adder; this gives "Add with carry" with 2s' complement addition. (In a very few CPUs, an end-around carry -- the final carry-out of the parallel adder is directly connected to the first carry-in of the same parallel adder -- gives 1's complement addition).

Ripple Carry Adder



Numbers of more than 1 bit long require more then just a single full adder to manipulate using arithmetic and bitwise logic instructions[*citation needed*]. A simple way of operating on larger numbers is to cascade a number of full-adder blocks together into a **ripple-carry adder**, seen above. Ripple Carry adders are so called because the carry value "ripples" from one block to the next, down the entire chain of full adders. The output values of the higher-order bits are not correct, and the arithmetic is not complete, until the carry signal has completely propagated down the chain of full adders.

If each full adder requires 3 gate delays for computation, then an *n*-bit ripple carry adder will require *3n* gate delays. For 32 or 64 bit computers (or higher) this delay can be overwhelmingly large.

Ripple carry adders have the benefit that they require the least amount of hardware of all adders (except for serial adders), but they suffer by being the slowest (except for serial adders).

With the full-adder verilog module we defined above, we can define a 4-bit ripple-carry adder in Verilog. The adder can be expanded logically:

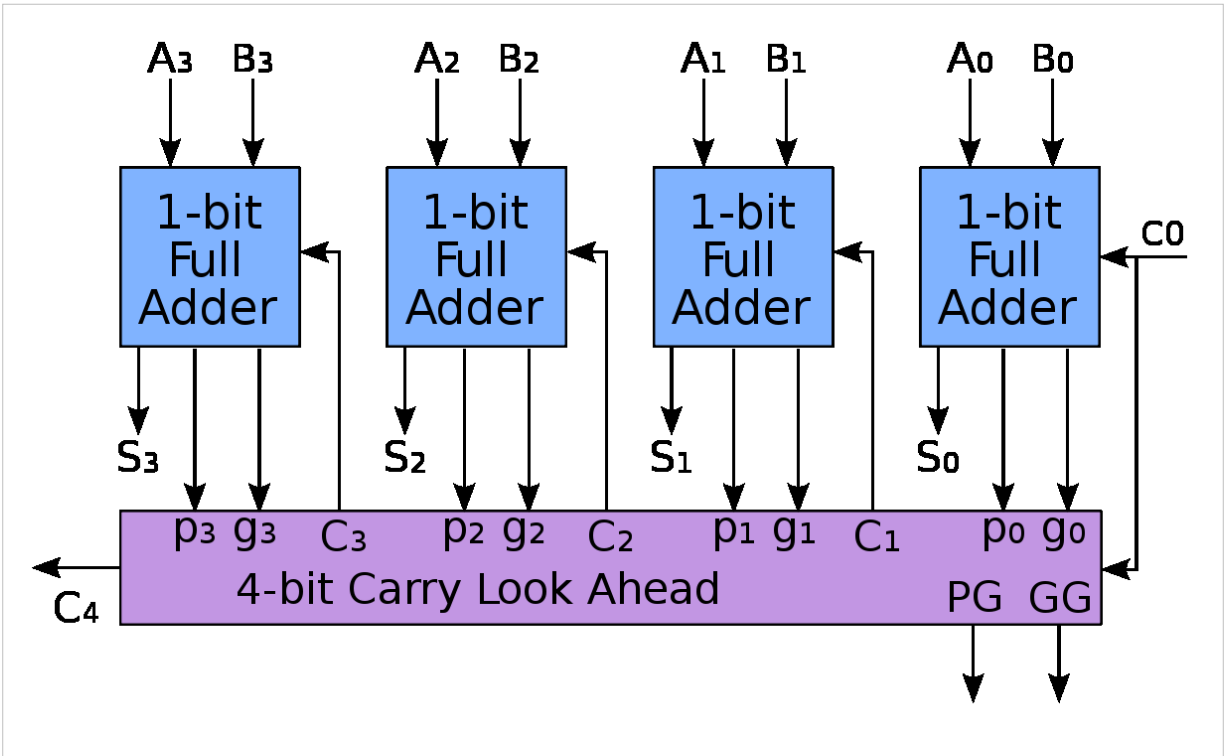
```
wire [3:0] c;
wire [3:0] s;
full_adder fa1(a[0], b[0], 1'b0, c[0], s[0]);
full_adder fa1(a[1], b[1], c[0], c[1], s[1]);
full_adder fa1(a[2], b[2], c[1], c[2], s[2]);
```

```
full_adder fa1(a[3], b[3], c[2], c[3], s[3]);
```

At the end of this module, *s* contains the 4 bit sum, and c[3] contains the final carry out.

This "ripple carry" arrangement makes "add" and "subtract" take much longer than the other operations of an ALU (AND, NAND, shift-left, divide-by-two, etc). A few CPUs use a ripple carry ALU, and require the programmer to insert NOPs to give the "add" time to settle.[1] A few other CPUs use a ripple carry adder, and simply set the clock rate slow enough that there is plenty of time for the carry bits to ripple through the adder. A few CPUs use a ripple carry adder, and make the "add" instruction take more clocks than the "XOR" instruction, in order to give the carry bits more time to ripple through the adder on an "add", but without unnecessarily slowing down the CPU during a "XOR". However, it makes pipelining much simpler if every instruction takes the same number of clocks to execute.

Carry Lookahead Adder



Carry-lookahead adders use special "look ahead" blocks to compute the carry from a group of 4 full-adders, and passes this carry signal to the next group of 4 full adders. lookahead units can also be cascaded, to minimize the number of gate delays to completely propagate the carry signal to the end of the chain. Carry lookahead adders are some of the fastest adder circuits available, but they suffer from requiring large amounts of hardware to implement. The number of transistors needed to implement a carry-lookahead adder is proportional to the number of inputs cubed.

The addition of two 1-digit inputs `A` and `B` is said to *generate* if the addition will always carry, regardless of whether there is an input carry (equivalently, regardless of whether any less significant digits in the sum carry). For example, in the decimal addition 52 + 67, the addition of the tens digits 5 and 6 *generates* because the result carries to the hundreds digit regardless of whether the ones digit carries (in the example, the ones digit clearly does not carry).

In the case of binary addition, $A + B$ generates if and only if both `A` and `B` are 1. If we write $G(A, B)$ to represent the binary predicate that is true if and only if $A + B$ generates, we have:

$$G(A, B) = A \cdot B$$

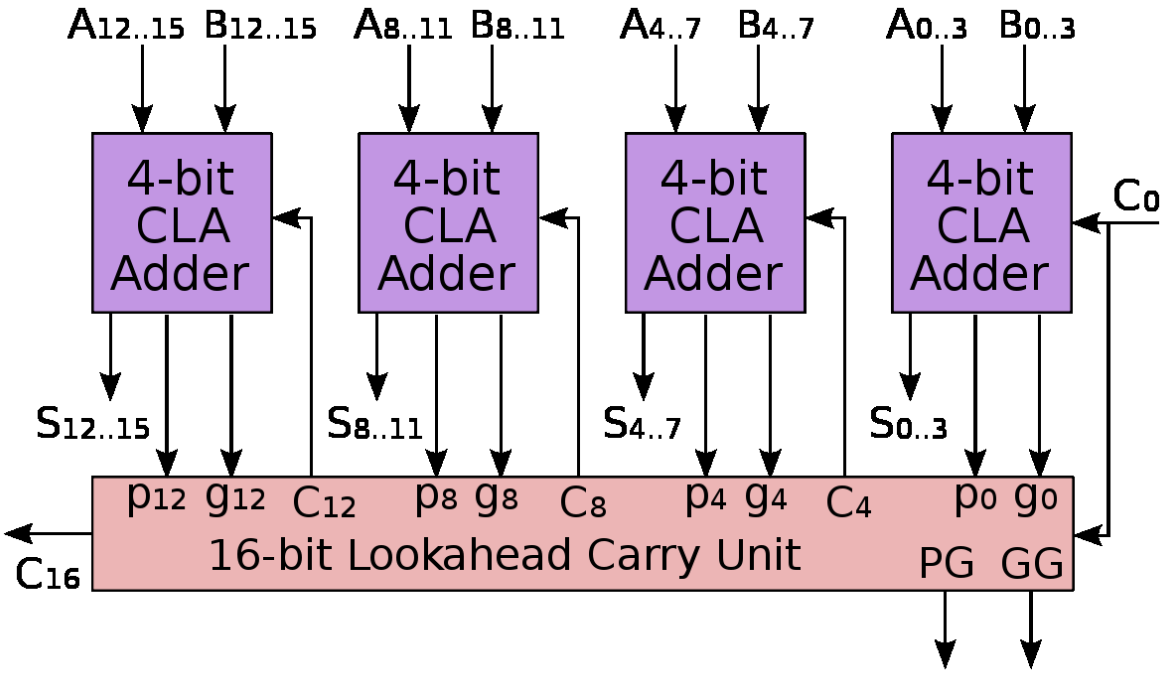
The addition of two 1-digit inputs `A` and `B` is said to *propagate* if the addition will carry whenever there is an input carry (equivalently, when the next less significant digit in the sum carries). For example, in the decimal addition 37 + 62, the addition of the tens digits 3 and 6 *propagate* because the result would carry to the hundreds digit *if* the ones were to carry (which in this example, it does not). Note that propagate and generate are defined with respect to a single digit of addition and do not depend on any other digits in the sum.

In the case of binary addition, $A + B$ propagates if and only if at least one of `A` or `B` is 1. If we write $P(A, B)$ to represent the binary predicate that is true if and only if $A + B$ propagates, we have:

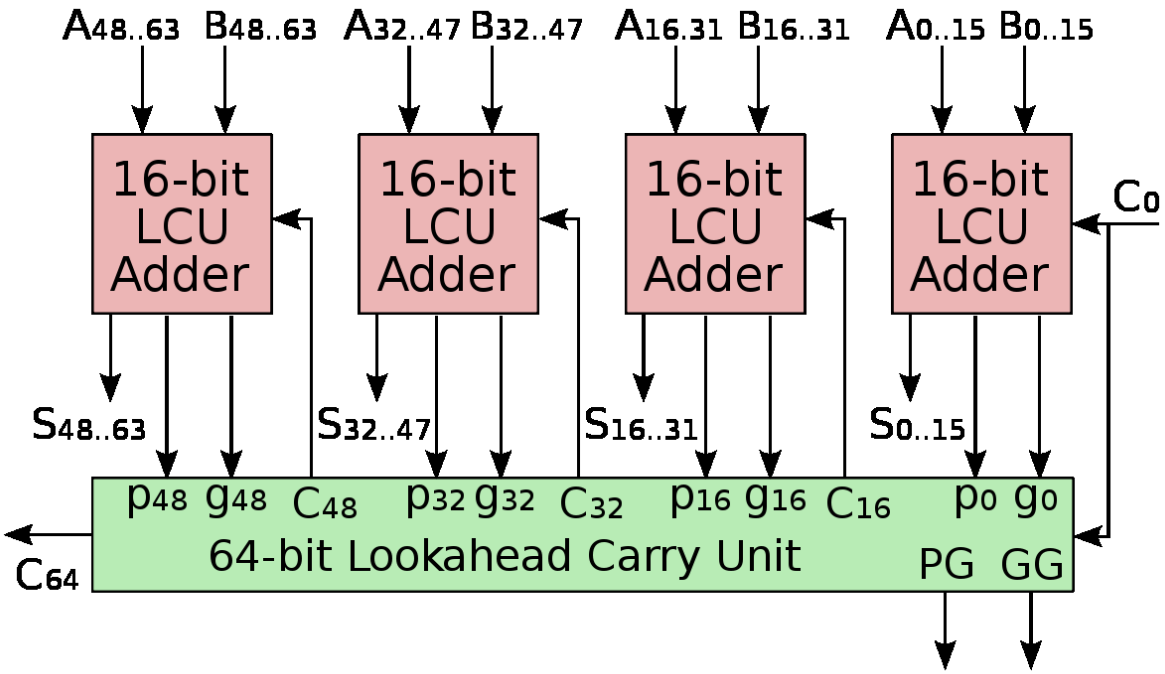
$$P(A, B) = A + B$$

Cascading Adders

The power of carry-lookahead adders is that the bit-length of the adder can be expanded without increasing the propagation delay too much. By cascading lookahead modules, and passing "propagate" and "generate" signals to the next level of the lookahead module. For instance, once we have 4 adders combined into a simple lookahead module, we can use that to create a 16-bit and a 64-bit adder through cascading:

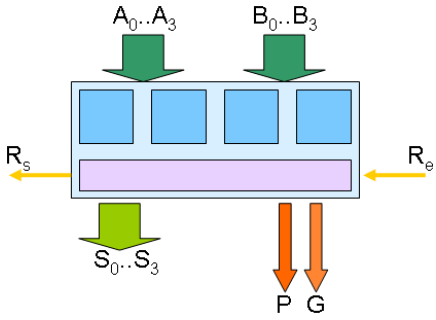


The 16-Bit carry lookahead unit is exactly the same as the 4-bit carry lookahead adder.

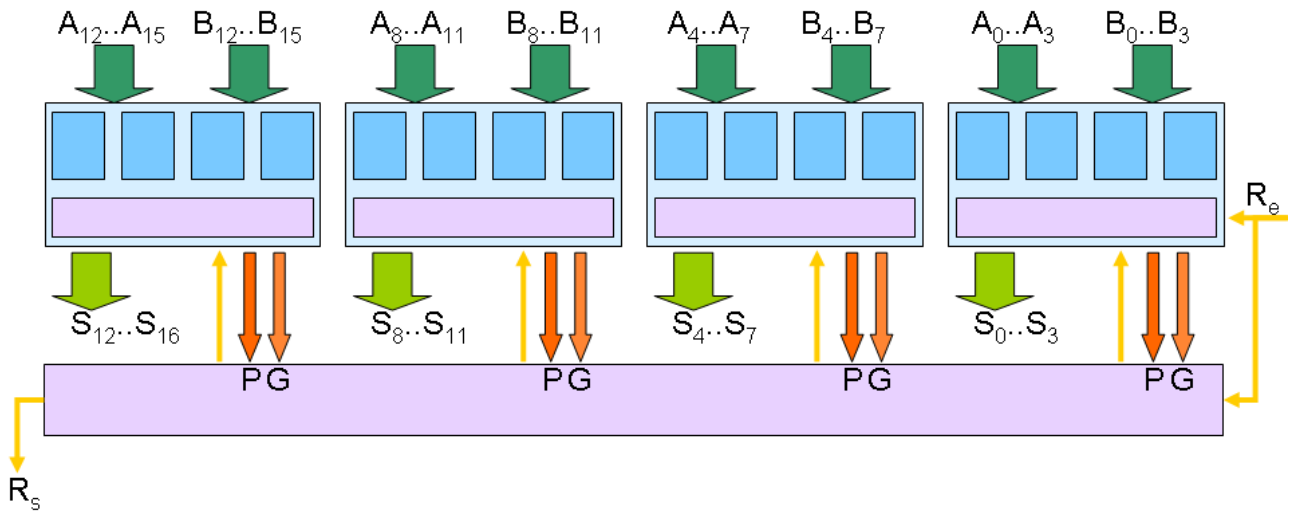


the 64-bit carry lookahead unit is exactly the same as the 4-bit and 16-bit units. This means that once we have designed one carry lookahead module, we can cascade it to any large size.

Generalized Cascading



A generalized CLA block diagram. Each of the turquoise blocks represents a smaller CLA adder.



We can cascade the generalized CLA block above to form a larger CLA block. This larger block can then be cascaded into a larger CLA block using the same method.

Sources

[1] "MuP21 Machine Forth": "Ripple Carry on + and +*" (http://www.ultratechnology.com/mfp21.htm)

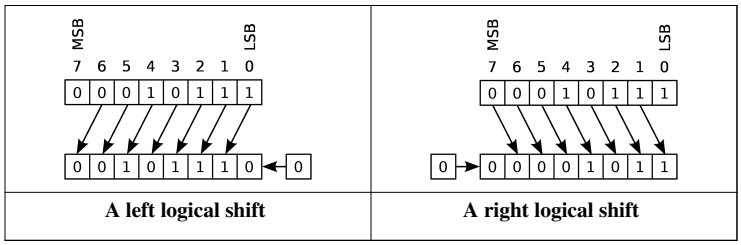
Microprocessor Design/Shift and Rotate Blocks

Microprocessor Design

Shift and Rotate

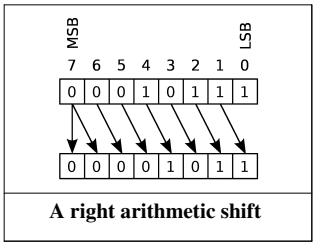
Shift and rotate blocks are essential elements in most processors. They are useful on their own, but they also are used in multiplication and division modules. In a binary computer, a left shift has the same effect as a multiplication by 2, and a right shift has the same effect as a division by 2. Since shift and rotate operations perform much more quickly then multiplication and division, they are useful as a tool in program optimization.

Logical Shift



In a logical shift, the data is shifted in the appropriate direction, and a zero is shifted into the new location.

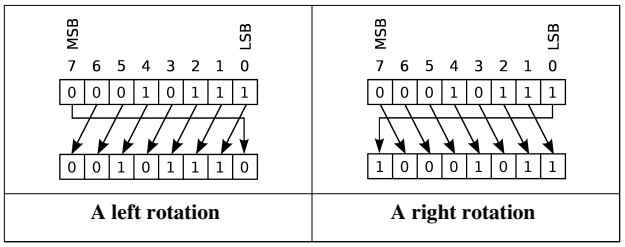
Arithmetic shift



A right arithmetic shift

In an arithmetic shift, the data is shifted right so that the sign of the data item is preserved. This means that the MSB is the value that is shifted into the new position. An arithmetic left shift is the same as a logical left shift, and is therefore not shown here.

Rotations



A left rotation **A right rotation**

A rotation is like a shift, except the bit shifted off the end of the register is then shifted into the new spot.

Fast Shift Implementations

The above images in each section help to indicate a method to shift a register more quickly, at the expense of requiring additional hardware. Instead of having one register that attempts to shift in place, we have have two registers in parallel, with wires connecting the various blocks together. When a shift is indicated, gates open that allow the data to pass from one register to the next, the proper number of spaces forward or backward.

In practice, fast shift blocks are implemented as a "barrel shifter". The barrel shifter includes several "levels" of multiplexers, each connected to the previous one by straight wires (wires that transfer the data without a shift), and wires that cause a shift by successive powers of two. For instance, the first level of shift would be 4 spaces, the next level would be 2 spaces, and the last level would be 1 space. In this way, the value of each shift level corresponds to the binary representation of the number of spaces to shift. This implementation makes for very fast shifters that can shift an arbitrary number of spaces in a single clock cycle.

Further reading

32-Bit Barrel Shifter Implementation Using 74-Series Integrated Circuits [1]

References

[1] http://ixeelectronics.com/Chipset/CPU/V3264/BarrelShifter32.html

Microprocessor Design/Multiply and Divide Blocks

Microprocessor Design

Multiply and Divide Problems

Multiplication and Division operations are significantly more complicated then addition or subtraction operations. This additional complexity leads to more hardware, more complicated hardware, and longer processing time.

In hardware, multiplication and division are performed by a series of sequential additions and arithmetic shifts. for this reason, it is imperative that we have efficient adders and shifters at our disposal.

Multipliers and dividers are composed of shifters and adders. It is typically not possible, or not desirable to to use the main adder and shifter units of the ALU, so a microprocessor will typically have multiple ALU units (a primary unit for addition and subtraction, and units embedded in the multiplication and division units). These are other good reasons why our ALU and shifters need to be small and fast.

Multiplication Algorithms

Multiply and Acculumulate

Multiply and accumulate (MAC) operations perform a multiplication and an addition in a single instruction. For instance, the instruction:

```
MAC A, B, C
```

Would perform the operation:

```
A = A + (B × C)
```

This is valuable for math-intensive processors, such as graphics processors and DSPs.

An MAC tends to have a long critical path, so if your processor has an MAC operation it is probably possible to include other complicated arithmetic operations.

In a processor with an accumulator architecture, MAC operations will use the accumulator as the destination register, so the instruction:

```
MAC B, C
```

Will perform the operation:

```
ACC = ACC + (B × C)
```

Fused Multiply-Add

A **fused multiply-add** operation is a floating-point operation that is similar to the MAC. However, in the fused operation, the floating-point values are not rounded between the multiply and the add, they are rounded afterwards. For more information about floating-point rounding, see Floating Point.

Microprocessor Design/ALU Flags

For a number of reasons, it can be important to export a number of status codes from the ALU, for detecting errors, and for making decisions.

Comparisons

Comparisons between two values are typically performed by subtracting them. We can determine the relationship between the two values by examining the difference:

- If the first is larger than the second, the result will be positive
- If the second is larger than the first, the result will be negative
- If the two are equal, the result will be zero.

Zero Flag

Determining whether two values are equal requires the ALU to determine whether the result is zero. This can be accomplished by feeding each bit of the result into a NOR gate. The beauty of this is that a single multi-port NOR gate requires less hardware than an entire array of equivalent 2-port gates.

Overflow Flag

It is good to know when the result of an addition or multiplication is larger than the maximum result size. Likewise, it is also good to know if the result of a subtraction or a division is smaller than possible, and thus creates underflow. Either two separate flags can be used for these conditions, or one flag can be interpreted in different ways, depending on the input operation.

Carry/Borrow flag

This flag indicates when an operation results in a value larger than the accumulator can represent (carry/overflow) or smaller than the accumulator can represent (borrow/underflow). It can be used by software to implement arbitrary-width arithmetic, such as a "bignum" library.

Comparisons

Many ALUs need to compare data items, and determine if a particular value is greater than or less than another value. In these cases, the ALU will also export flags for these values.

A comparison in a processor can typically be performed by a subtraction operation. If the result is a positive number, the first item is greater than the second item. If the result is a negative number, the first item is less than the second. If the numbers being compared are unsigned, the value of the carry flag will serve the same purpose as the greater-than or less-than flag.

Latch ALU flags or not?

Some instruction sets refer to the ALU flags from some previous instruction:

```
CMP R1,R2 // compare
...
BEQ equal_routine // branch if equal
```

Such instruction sets force the CPU designer to latch those ALU flags in some sort of "status register", and to be very careful to make sure it is possible to preserve those flags during an interrupt routine.

Other instruction sets never refer to previous ALU flags -- they always use the results from the ALU in the same instruction that they are calculated:

```
BEQ R1,R2,equal_routine // compare and branch if equal
```

or

```
SKEQ R1,R2 // compare and skip next instruction if equal
  JMP equal_routine
```

Some CPU designers prefer such instruction sets that never refer to previous ALU flags. Such instruction sets make out-of-order execution much simpler. Many of Chuck Moore's CPU designs never refer to the ALU flags from any previous instruction.

Design Paradigms

Microprocessor Design/Single Cycle Processors

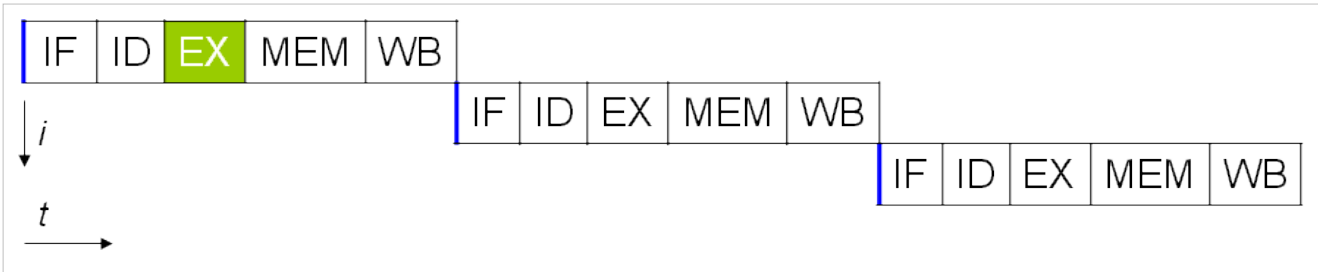
Microprocessor Design

Single-cycle processors are what we have been studying so far: an instruction is fetched from memory, it is executed, and the results are stored all in a single clock cycle.

The benefits of single-cycle processors is that they tend to be the most simple in terms of hardware requirements, and they are easy to design. Unfortunately, they tend to have poor data throughput, and require long clock cycles (slow clock rate) in order to perform all the necessary computations in time.

Cycle Times

The length of the cycle must be long enough to accommodate the longest possible propagation delay in the processor. This means that some instructions (typically the arithmetic instructions) will complete quickly, and time will be wasted each cycle. Other instructions (typically memory read or write instructions) will have a much longer propagation delay.



As this image shows, an instruction is not over until all 5 components have acted. This means that the length of the cycle must be the length of the longest instruction. The longest path from one end of the processor to the other is called the **critical path** and is used to determine the cycle time.

Redundant Hardware

Single cycle processors typically require a number of ALUs (or a single master ALU, and smaller ALUs) to handle the increment operations on the instruction pointer, and the memory address calculations for the data memory. When resources are at a premium, having multiple ALU units in your design can be costly and pointless. It requires nearly as many resources to construct an adder that adds a constant value as it does to construct a more general purpose adder unit.

Single Cycle Designs

It is very rare, if not completely unheard of, for a modern processor unit to have a single-cycle design. The reasons for this are the long cycle times, the wasted resources, and the large amount of wasted time in each cycle. What the single-cycle lacks in timing and efficiency, it makes up for in simplicity and elegance. It is for this reason that single-cycle processors work as a good teaching tool, but are not often employed in actual designs.

Microprocessor Design/Multi Cycle Processors

Microprocessor Design

Single-cycle processors suffer from poor speed performance. Control and data signals must propagate completely through the processor in a single cycle, which means that cycle times need to be long, and many parts of the hardware tend to be dormant for much of the cycle.

Multi-Cycle Stages =

Multi-cycle processors break up the instruction into its fundamental parts, and executes each part of the instruction in a different clock cycle. Since signals have less distance to travel in a single cycle, the cycle times can be sped up considerably.

Typically, an instruction is executed over at least 5 cycles, which are named as such:

IF

: Fetch the instruction from memory

ID

: Decode the instruction, and generate the necessary control signals

EX

: Feed the necessary control signals into the ALU and produce a result

MEM

: Read from memory, if specified

WB

: Write the result back to the register file or to memory.

This is just a textbook example, and modern processes tend to use many more steps than this to execute an instruction.

Example: MicroChip PIC16 Microcontroller

The PIC Microcontroller, manufactured by MicroChip Technology Inc, is a family of embedded microcontrollers. The PIC units vary, but execute an instruction every 2-4 clock cycles. All instructions typically execute in the same number of cycles, except for branch instructions.

Hardware Reuse

The primary benefit to a multicycle design is to be able to share hardware elements, specifically the ALU, among various tasks. In a multicycle processor, a single ALU can be used to update the instruction pointer (in the **IF** cycle), perform the operation (in the **EX** cycle), and calculate a necessary memory address (in the **MEM** cycle). Multicycle processors also allow computers that have a single memory unit, instead of the two separate instruction and data memory units of the traditional harvard machine. This is because the instructions are loaded on one cycle, and the data memory is interfaced on another cycle.

Multi-cycle processors are typically used in applications where resources are at a premium, and speed is not as important.

Microprocessor Design/Pipelined Processors

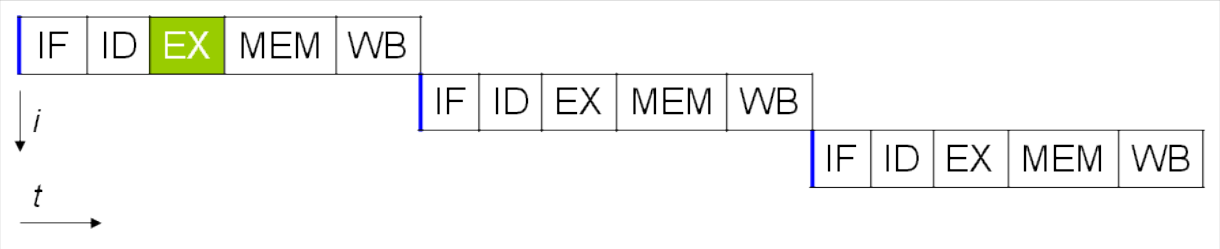
Microprocessor Design

Pipelining Introduction

Let us break down our microprocessor into 5 distinct activities, which generally correspond to 5 distinct pieces of hardware:

1. Instruction fetch (IF)
2. Instruction Decode (ID)
3. Execution (EX)
4. Memory Read/Write (MEM)
5. Result Writeback (WB)

Any given instruction will only require one of these modules at a time, generally in this order. The following timing diagram of the multi-cycle processor will show this in more detail:



This is all fine and good, but at any moment, 4 out of 5 units are not active, and could likely be used for other things.

Pipelining Philosophy

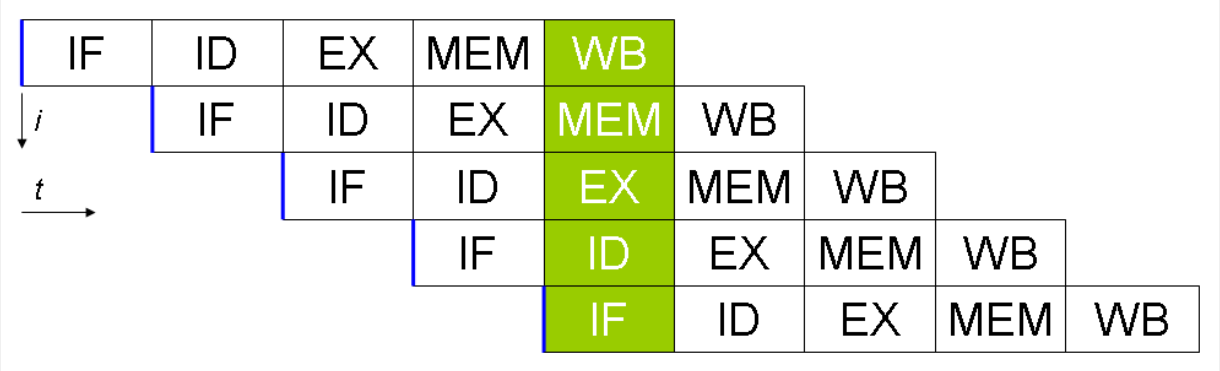
Pipelining is concerned with the following tasks:

- Use multi-cycle methodologies to reduce the amount of computation in a single cycle.
- Shorter computations per cycle allow for faster clock cycles.
- Overlapping instructions allows all components of a processor to be operating on a different instruction.
- Throughput is increased by having instructions complete more frequently.

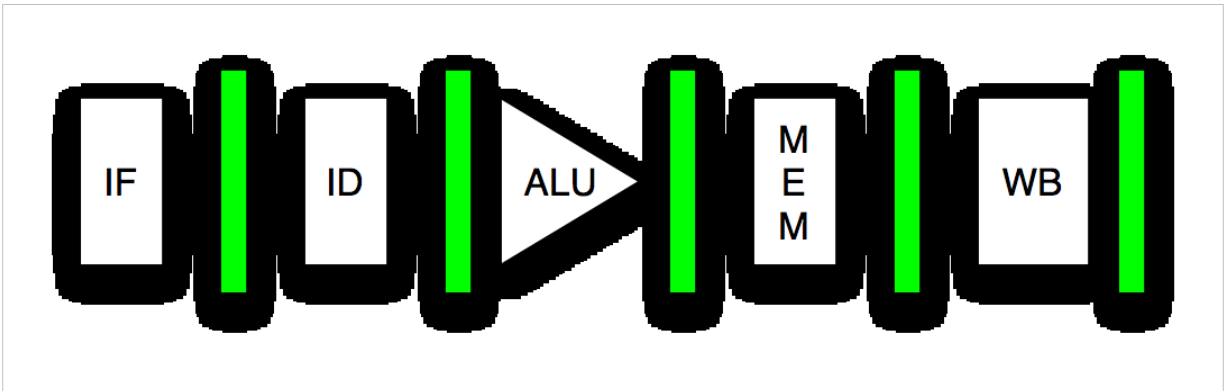
We will talk about how to make these things happen in the remainder of the chapter.

Pipelining Hardware

Given our multicycle processor, what if we wanted to overlap our execution, so that up to 5 instructions could be processed at the same time? Let's contract our timing diagram a little bit to show this idea:

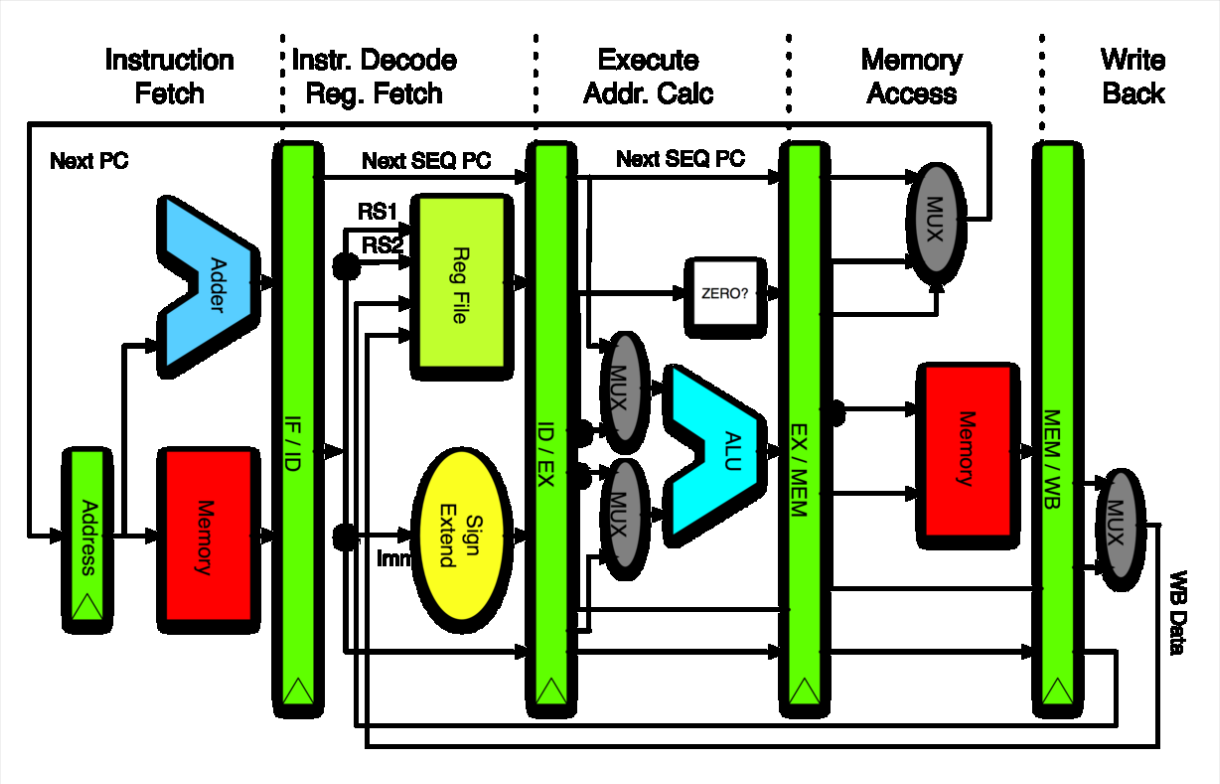


As this diagram shows, each element in the processor is active in every cycle, and the instruction rate of the processor has been increased by 5 times! The question now is, what additional hardware do we need in order to perform this task? We need to add storage registers between each pipeline state to store the partial results between cycles, and we also need to reintroduce the redundant hardware from the single-cycle CPU. We can continue to use a single memory module (for instructions and data), so long as we restrict memory read operations to the first half of the cycle, and memory write operations to the second half of the cycle (or vice-versa). We can save time on the memory access by calculating the memory addresses in the previous stage.

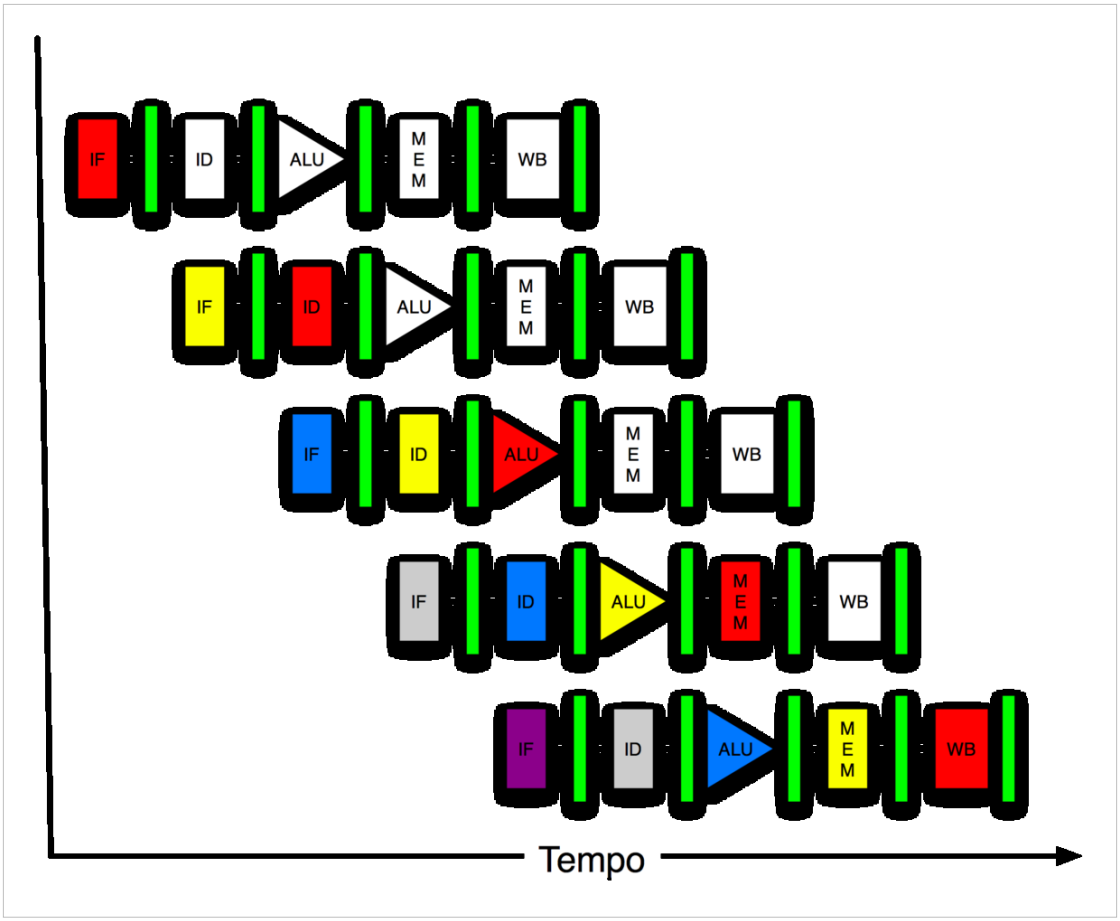


The registers would need to hold the data from the pipeline at that point, and also the necessary control codes to operate the remainder of the pipeline.

Our resultant processor design will look similar to this:



If we have 5 instructions, we can show them in our pipeline using different colors. In the diagram below, white corresponds to a NOP, and the different colors correspond to other instructions in the pipeline. Each stage, the instructions shift forward through the pipeline.



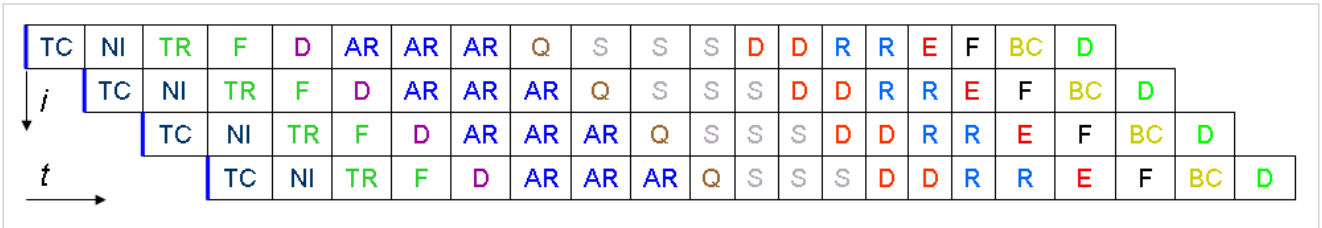
Superpipeline

Superpipelining is the technique of raising the pipeline depth in order to increase the clock speed and reduce the latency of individual stages. If the ALU takes three times longer then any other module, we can divide the ALU into three separate stages, which will reduce the amount of time wasted on shorter stages. The problem here is that we need to find a way to subdivide our stages into shorter stages, and we also need to construct more complicated control units to operate the pipeline and prevent all the possible **hazards**.

It is not uncommon for modern high-end processors to have more than 20 pipeline stages.

Example: Intel Pentium 4

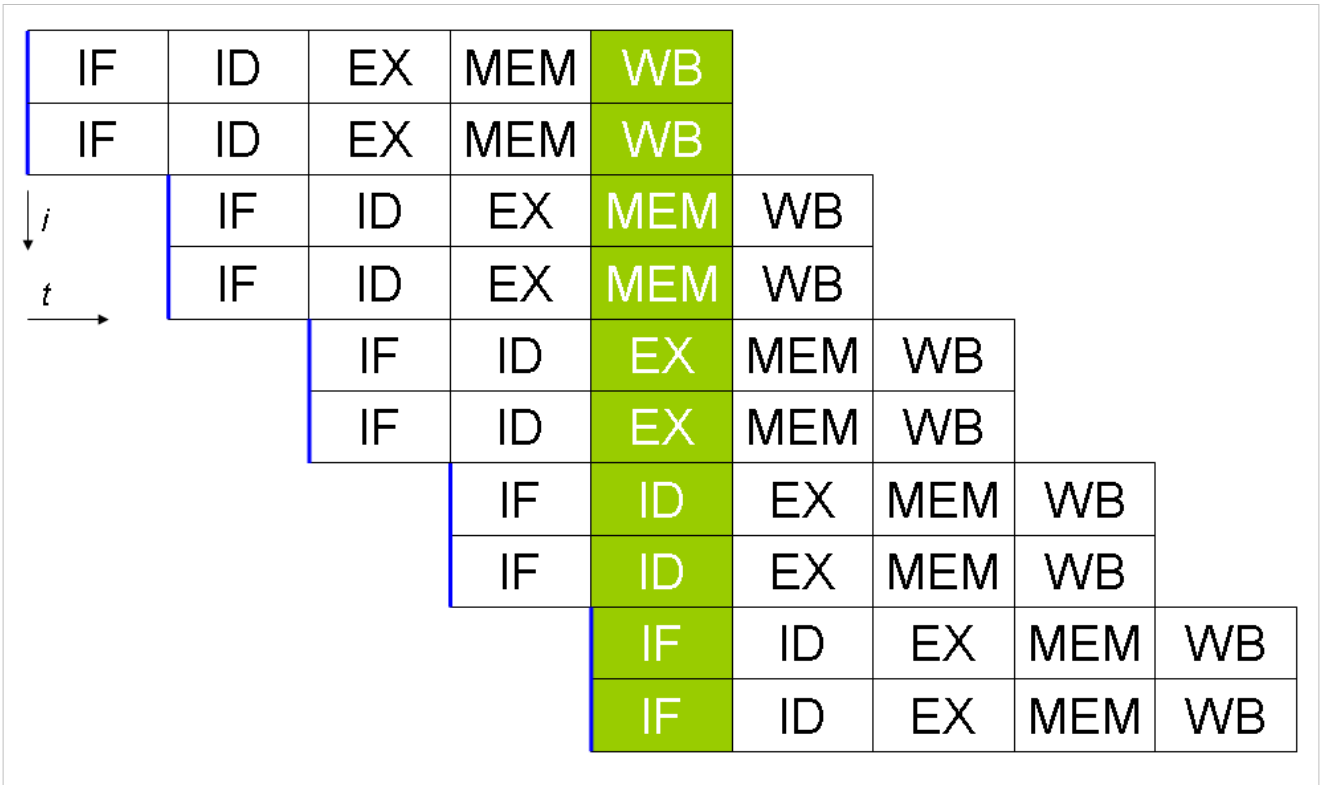
The Intel Pentium 4 processor is a recent example of a super-pipelined processor. This diagram shows a Pentium 4 pipeline with 20 stages.



Microprocessor Design/Superscalar Processors

Microprocessor Design

In a superscalar design, the processor actually has multiple datapaths, and multiple instructions can be exectuted simultaneously, one in each datapath. It is not uncommon for a superscalar CPU to have multiple ALU and FPU units, for each datapath.



In this image, all the stages highlighted in green are executing simultaneously. As we can see from this image, there are two execution cores operating simultaneously.

Microprocessor Design/VLIW Processors

Microprocessor Design

Very Long Instruction Words (VLIW) can be used to simultaneously specify multiple instructions in parallel with one another.

VLIW Vs Superscalar

In a superscalar design, the microprocessor will have multiple independant execution units. An instruction scheduler determines which instructions will be executed on which execution unit, at what time. This scheduler unit requires large amounts of additional hardware complexity.

VLIW is similar to superscalar architecture except that instead of using scheduling hardware to map instructions to available execution units, instructions for all units are provided in every instruction word. The scheduling is performed by the compiler at compile time.

The term VLIW comes from the fact that multiple instructions typically requires large instruction words. If each instruction is 32 bits (including opcode, source and destination registers, etc), and the processor has 4 execution cores, then the total instruction word length is 128 bits long!

Multi-Issue

Similar to the VLIW design, a multi-issue processor will issue an unfixed number of instructions per cycle, and each will be executed simultaneously.

Microprocessor Design/Vector Processors

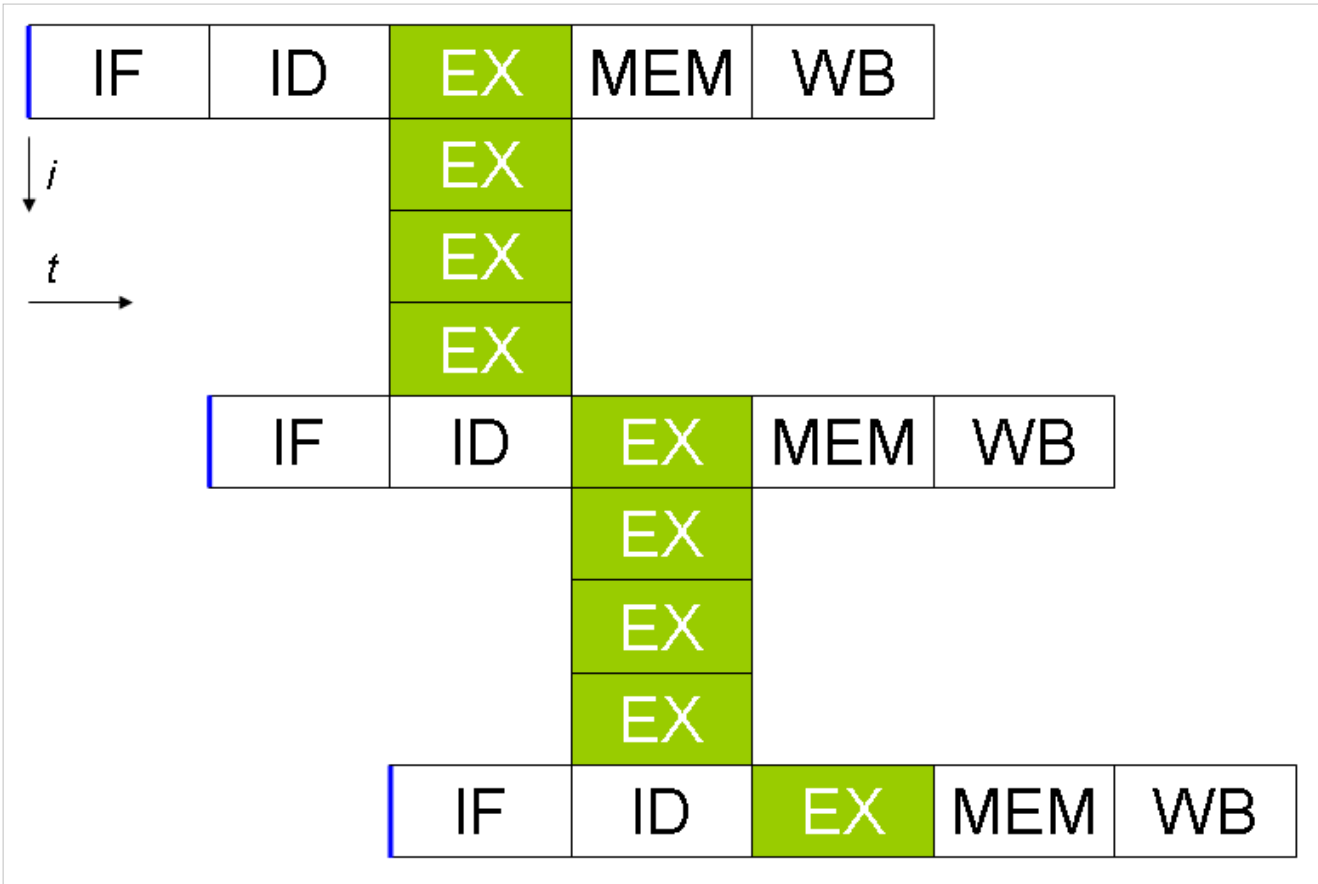
Microprocessor Design

Vector processors, or SIMD processors are microprocessors that are specialized for operating on vector or matrix data elements. These processors have specialized hardware for performing vector operations such as vector addition, vector multiplication, and other operations.

Modern graphics processors and GPUs tend to be vector-based processors. Modern Intel-based chips also have SIMD capabilities known as SSE or MMX operations.

Parallel Execution

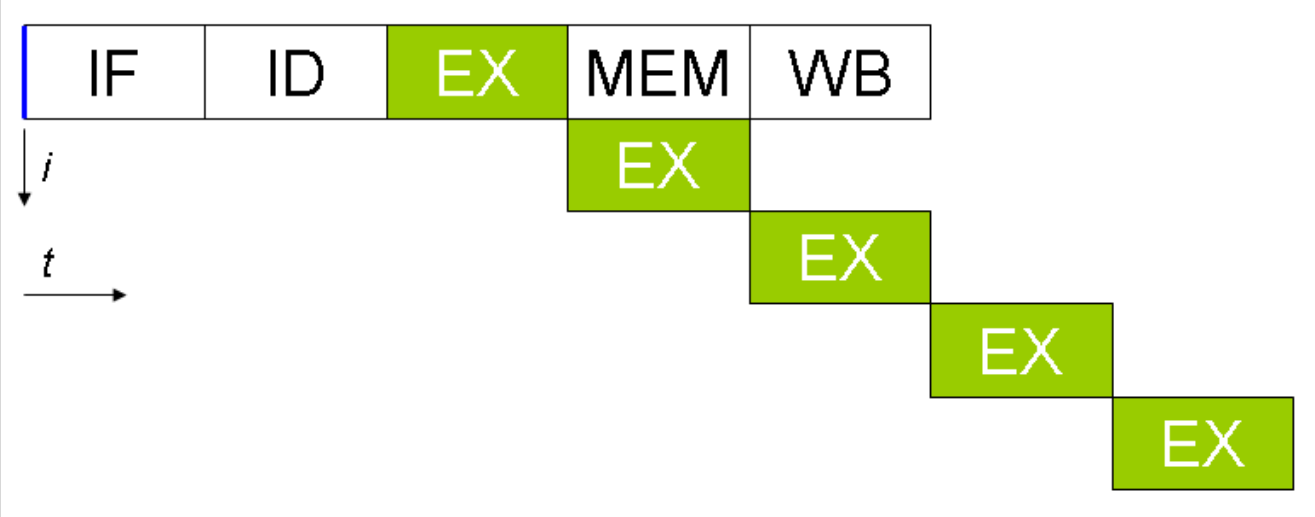
Vector processors which perform an instruction on all data elements simultaneously are said to execute in parallel.



Each EX in this image shows a separate execution core (typically an ALU) operating in parallel with one another.

Non-Parallel Execution

Vector processors which reuse a single ALU for a vector operation look like this:



As this diagram shows, each EX stage is a new set of data from the first instruction being loaded into the execution core. The next instruction will not be fetched until all the data has been acted upon.

Microprocessor Design/Multicore Processors

Taking the idea of superscalar operations to the next level, it is possible (and frequently desirable) to put multiple microprocessor cores onto a single chip, and have the cores operate in parallel with one another.

Symmetric Multicore

A symmetric multicore processor is one that has multiple cores on a single chip, and all of those cores are identical.

Example: Intel Core 2:

The Intel Core 2 is an example of a symmetric multicore processor. The Core 2 can have either 2 cores on chip ("Core 2 Duo") or 4 cores on chip ("Core 2 Quad"). Each core in the Core 2 chip is symmetrical, and can function independantly of one another. It requires a mixture of scheduling software and hardware to farm tasks out to each core.

Example: Parallax Propeller:

The Parallax Propeller is an example of a symmetric multicore processor. The Parallax Propeller has 8 cores on chip, each one a 32-bit RISC processor. Each core in the Parallax Propeller chip is symmetrical, and can function independently of one another.

Asymmetric Multicore

An asymmetric multicore processor is one that has multiple cores on a single chip, but those cores might be different designs. For instance, there could be 2 general purpose cores and 2 vector cores on a single chip.

Example: Cell Processor

IBM's Cell processor, used in the Sony PlayStation 3 video game console is an asymmetrical multicore processor. The Cell has 9 processor cores on board, one general purpose processor, and 8 data-processing cores. The one multipurpose core, known as the **Power Processor Element** (PPE) controls the communication between the other cores, and distributes computing tasks to the other cores for processing. The other 8 cores are known as **Synergistic Processor Elements** (SPE), and are specially designed to have high floating-point throughput, especially with vector operations.

Example: Kilocore

Rapport's Kilocore processor, is an asymmetrical multicore processor. The Kilocore has one general purpose processor, a PowerPC processing core, and either 256 or 1024 data processing cores on-chip. The cores are designed to run at extremely low power, so the overall chip is faster and yet uses less power than typical desktop CPUs[1] .

Symmetric Multicore

A symmetric multicore processor is a processor which has multiple cores that are all exactly the same. Every single core has the same architecture and the same capabilities. An example of a symmetric multicore system is the Intel Core 2 Duo processor.

Each core has the same capabilities, so it requires that there is an arbitration unit to give each core a specific task. Software that uses techniques like **multithreading** makes the best use of a multi-core processor like the Intel Core 2.

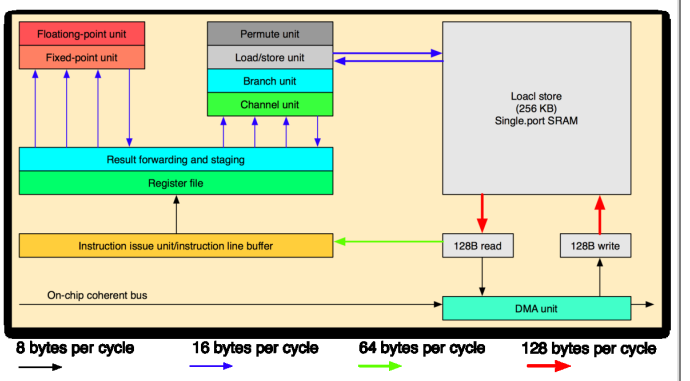
The Intel Core 2 Duo

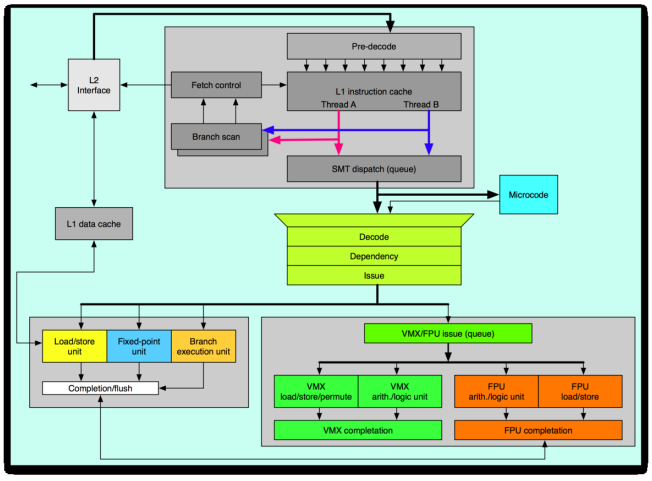
Asymmetric Multicore

In an asymmetric multicore processor, the chip has multiple cores onboard, but the cores might be different designs. Each core will have different capabilities.

Example: IBM Cell Processor

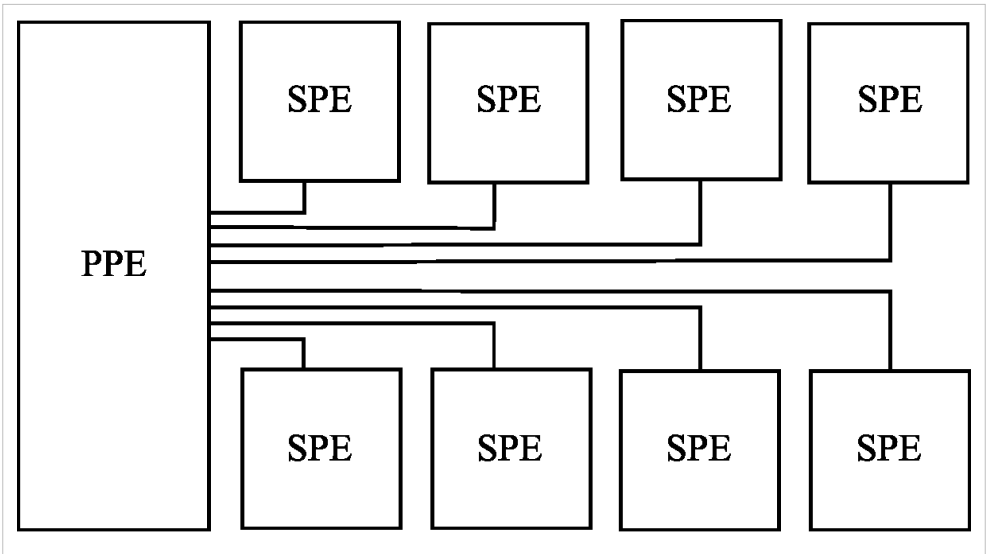
An example of an asymmetric multicore processor is the IBM Cell processor.





Block diagrams of the IBM Cell processor. The Cell processor has 8 SPE cores (left) and 1 PPE core (right). The PPE core is the primary core, and controls the behavior of the SPE cores.

The IBM Cell processor has 1 PPE that controls the chip, and 8 SPEs that are designed for high mathematical throughput. The IBM Cell processor is designed as follows:



Notice how the SPE cores only connect to the PPE, and not to each other. Notice also that the PPE core is much larger then the individual SPE cores.

further reading

[1] Tom's hardware: "IBM says Kilocore technology will outrun today's mobile processors" (http://www.tomshardware.com/news/ibm-rapport-kilocore,2575.html) 2006

Execution Problems

Microprocessor Design/Exceptions

Microprocessor Design

Exceptions, are situations where the processor needs to stop executing the current code because of an error. In these cases, the processor typically begins running an **exception handling routine** to resolve the error, and then returns to the normal program flow. For instance, if the ALU attempts to divide by zero, or if an addition causes overflow, an exception might be triggered. The processor needs to stop operation and fix the error before the program can be resumed.

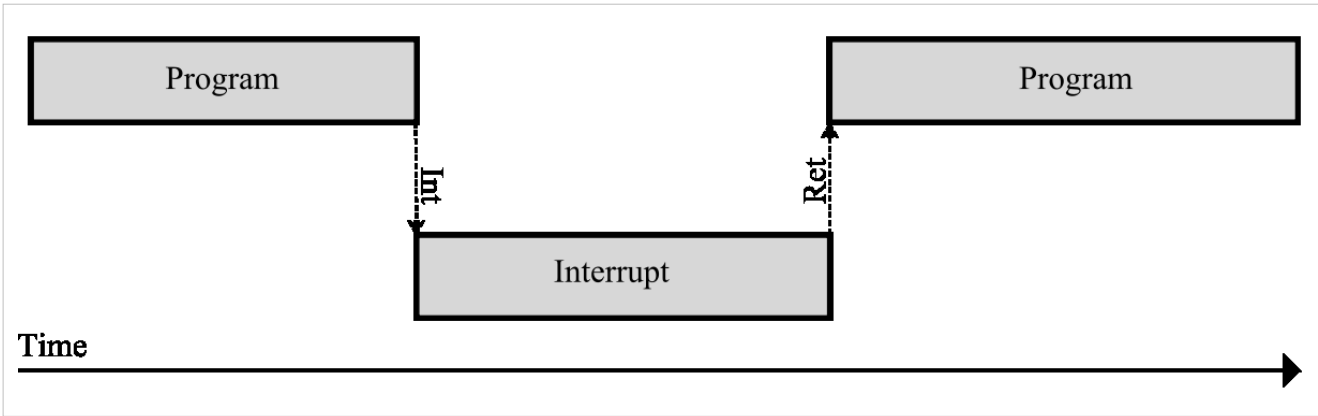
Some common examples of exceptions are arithmetic overflow or underflow, division by zero, or attempting to access a memory location that does not exist.

Microprocessor Design/Interrupts

Microprocessor Design

An **interrupt** is a condition that causes the microprocessor to temporarily work on a different task, and then later return to its previous task. Interrupts can be internal or external. Internal interrupts, or "software interrupts," are triggered by a software instruction and operate similarly to a jump or branch instruction. An external interrupt, or a "hardware interrupt," is caused by an external hardware module. As an example, many computer systems use **interrupt driven I/O**, a process where pressing a key on the keyboard or clicking a button on the mouse triggers an interrupt. The processor stops what it is doing, it reads the input from the keyboard or mouse, and then it returns to the current program.

The image below shows conceptually how an interrupt happens:



The grey bars represent the control flow. The top line is the program that is currently running, and the bottom bar is the interrupt service routine (ISR). Notice that when the interrupt (**Int**) occurs, the program stops executing and the microcontroller begins to execute the ISR. Once the ISR is complete, the microcontroller returns to processing the program where it left off.

What happens when external hardware requests another interrupt while the processor is already in the middle of executing the ISR for a previous interrupt request?

When the first interrupt was requested, hardware in the processor causes it to finish the current instruction, disable further interrupts, and jump to the interrupt handler.

The processor ignores further interrupts until it gets to the part of the interrupt handler that has the "return from interrupt" instruction, which re-enables interrupts.

If an interrupt occurs while interrupts were turned off, some processors will immediately jump to that interrupt handler as soon as interrupts are turned back on. With this sort of processor, an interrupt storm "starves" the main loop background task. Other processors[*citation needed*] execute at least one instruction of the main loop before handling the interrupt, so the main loop may execute extremely slowly, but at least it never "starves".

further reading

- Operating System Design/Processes/Interrupt

Microprocessor Design/Hazards

Microprocessor Design

A **hazard** is an error in the operation of the microcontroller, caused by the simultaneous execution of multiple stages in a pipelined processor.

There are three types of hazards: Data hazards, control hazards, and structural hazards.

Data Hazards

Data hazards are caused by attempting to access data or modify data simultaneously. In the MIPS design, the result is written back to the register file at the same time that another instruction decode stage is reading the register file. There are three basic types of data hazards:

Read After Write (RAW)

> In these hazards, the read process happens after the write process, although both processes happen in the same clock cycle. If the write process takes a long time, it may not complete by the time the read occurs, which will produce incorrect data.

Write After Read (WAR)

> In a WAR hazard, the write from a previous instruction will not complete before the successive read instruction. This means that the next value read will be a previous value, not the correct current value.

Write After Write (WAW)

> WAW hazards occur when two processes try to write to a data storage element at the same time. If this occurs in a single clock cycle, there will be no time in between to read the intermediate value. If the instructions execute out of order, the incorrect value may be left in the register.

Race Conditions

If data hazards are not explicitly accounted for, a **race condition** can arise where the proper execution of the processor is a matter of timing. If things occur in the proper times and the proper sequence, there might be no problems. However, In a race condition it is frequently likely that things will occur out of order, or at different time intervals, and this will cause a problem.

Control Hazards

Control hazards occur when a branch instruction is processed. While the branch instruction is traveling through the pipeline, the instruction fetch module will continue to read sequential instructions from the instruction memory. The problem is that because of the branch, the next instructions might execute out of order, which will cause problems.

Structural Hazards

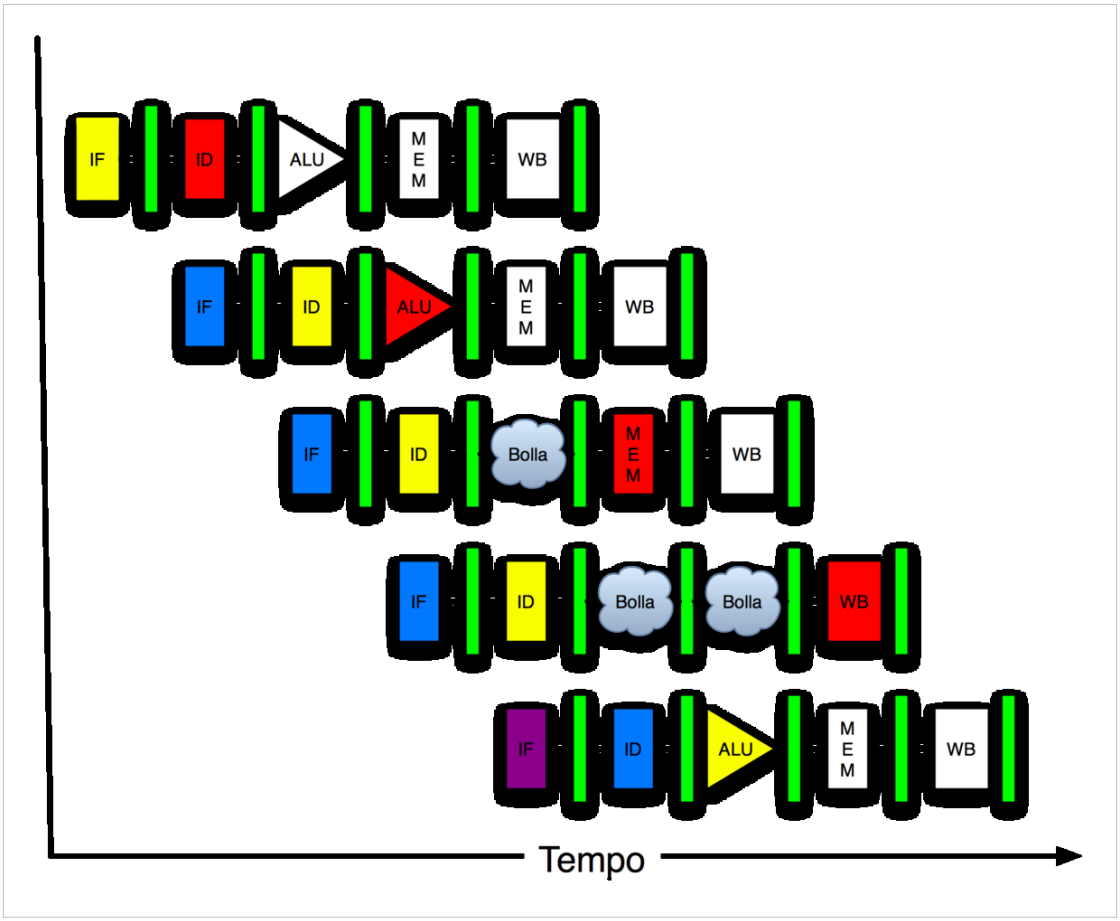
A structural hazard occurs when two separate instructions attempt to access a particular hardware module at the same time.

Fixing Hazards

There are a number of ways to avoid or eliminate hazards.

Stall

A **stall**, or a "bubble" in the pipeline occurs when the control unit detects that a hazard will occur. When this happens, the control unit stops the instruction fetch mechanism and puts NOPs into the pipeline instead. In this way, the sensitive instructions will be forced to occur alone, without any other instructions being processed at the same time.



In this image we can see "bubbles" drawn where data hazards occur. A bubble signifies that the instruction has stalled in the pipeline until a previous instruction completes. Once the previous instruction has completed, the stalled instruction continues moving.

Notice in this image that the yellow instruction stops at the ID stage for 2 cycles, while the red instruction continues.

Forwarding

When an result from one instruction is to be used as the input to the ALU in the next instruction, we can use **forwarding** to move data directly from the ALU output into the ALU input of the next cycle, before that data has been written to the register. In this way, we can avoid the need for a stall in these situations, but at the expense of adding an additional **forwarding unit** to control this mechanism.

Register renaming

Instead of having fixed numbers for registers, registers can be renamed or renumbered. Consider the following ADD instruction:

```
add R1, R2, R1
```

We are adding the values in R1 and R2, and we are storing the result back in R1. What if the name "R1" pointed to two different physical storage areas, that is the value is read from one location, the "old R1", and is written to a new storage area, the "new R1".

Register renaming can be used to prevent hazards caused by out-of-order execution (OOOE).

Speculative execution

During a branch, it is frequently possible to "guess" about the outcome of the branch. By guessing about the destination, instructions can be executed speculatively. If the guess is wrong, the pipeline will need to be emptied, which takes the same amount of time as a stall. However, if the guess is right, no time is wasted and the processor continues operation as normal.

The process of guessing which way the branch will take is a complicated subject and is beyond the current scope of this book.

Branch delay

A **branch delay** is an instruction written in the assembly source code after the branch, that is designed to execute whether the branch is taken or not. If there are no instructions that can be executed without a dependency on the branch, then a NOP should be inserted instead. Some assemblers are capable of rearranging code in this fashion, although other assemblers that use this technique require the programmer to handle branch delays manually.

Branch Predication

In a **branch predication** scheme, all instructions, or most instructions in the ISA may be conditionally executed based on some condition. In other words, the instruction will be loaded from memory, decoded, and then the processor will determine whether or not to execute it. In the event of a branch, for instance, the instructions in the pipeline after the branch can be turned off if the branch went the other direction. Branch predication is very closely related to speculative execution.

Branch Prediction

Branch Prediction is the act of guessing about the direction a branch instruction will take. Typically, branch predictors base these decisions off register values, and past branch history. In a large loop, for instance, a particular program may branch back to the top of the loop many many times before the loop terminates. Consider this high-level pseudo code:

```
while(condition)
    do this
end
```

Which roughly translates to this assembly pseudo code:

```
top of loop:
compare condition and 0.
branch to end of loop if equal
do this
branch to top of loop
bottom of loop:
```

This loop will continue to repeat until the *condition* flag is 0. This code will likely loop many times before the one time that it exits. In a while structure like this, it takes the branch every time except for the last time, and it only doesn't take the branch once. It makes good sense to assume, therefore, that every branch that we come to will be taken, which can increase the accuracy of our speculative execution.

Example: Loop Optimization

In modern processors, branch prediction will frequently look at the history of recent branches to determine how to guess the outcome of a future branch. Consider the following loop structure with a nested conditional:

```
while(loop condition)
   if(branch condition)
      do this
   else
      do that
 end
```

If we know statistically that the *branch condition* will be false (0) 90% of the time, and that the *loop condition* will be true (1) nearly 100% of the time. We can decompose this into assembly pseudo code:

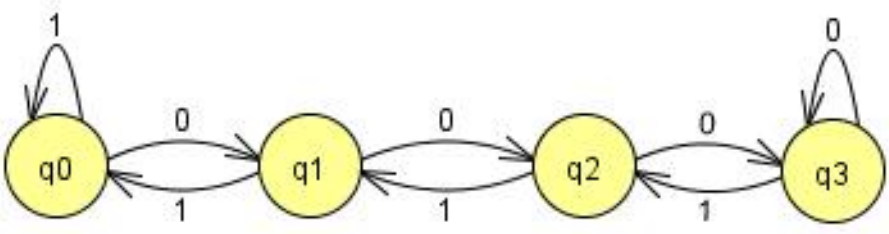
```
1) compare loop condition and 0
2) branch to end of loop if equal
3) compare branch condition and 0
4) branch to branch true if not equal
5) do that
6) branch to end of if
7) branch true
8) do this
9) end of if
10)branch to top of loop
```

If we look at this loop structure, we can see that the branch on line 10 is taken most of the time. We can also see that the branch on line on line 4 only occurs if the branch condition is 1. We know that the branch condition is true only 10% of the time, so this loop will have bad branch prediction. A better loop in this case would be:

```
while(loop condition)
   if(not branch condition)
      do this
   else
      do that
 end
```

so that the branch in the conditional is taken 90% of the time, so that the branch predictor will be more accurate.

A branch predictor typically acts like a counter. Every time a branch is taken, the counter is incremented, and every time a branch is not taken, the counter is decremented. Consider a 2-bit predictor. If the predictor is 0 or 1, the branch is not taken, but if the predictor is 2 or 3, the branch is taken.



A 2-bit branch predictor with 4 stages.

We can treat a branch predictor like a finite-state-machine (FSM) like in the diagram above. This FSM has 4 stages, corresponding to the following "guesses":

- q0:Strong Take
- q1:Weak Take
- q2:Weak Not Take
- q3:Strong Not Take

The zeros in this diagram refer to a branch not being taken, and a 1 corresponds to a branch being taken. If many branches are taken, the state moves towards the right. If branches are not taken, the stage moves towards the left.

Benchmarking and Optimization

Microprocessor Design/Performance Metrics

Microprocessor Design

Performance Metrics

Performance metrics are measurements of a microprocessor that help to determine how well a microprocessor performs.

For many years, computers were excruciatingly slow. Much time and effort were dedicated to finding ways of getting typical batch programs to run faster -- to reduce runtime or, in other words, to improve throughput. In the process, many other things were sacrificed.

Runtime

Runtime is the time it takes to run a program.

We will discuss some of the subtleties of accurately measuring runtime in a later Benchmarking section.

For now, let us note that, for any program running on any computer,

time per program = clock period * cycles per instruction * instructions executed per program

You see there are 3 different factors involved in the total time. If you can reduce any one of those factors, then the time will be shorter, making your users happier.

Alas, all too often attempts at making one factor shorter result in making some other factor larger. Sometimes a CPU designer will focusing on only one factor, trying to make it as small as possible, and hoping that the resulting increases in the other factors will be small enough that there is still a net improvement.

Clock rate

Clock rate (often called "clock speed") is one of the easiest to measure performance metrics, and the most over-emphasized.

As of 2008, clock rate of most CPUs is measured in MHz. A typical FPGA soft processor runs at about 10 MHz (a clock period of 100 ns), but later in this book we will explain techniques for increasing the clock rate of a FPGA soft processor to over 100 MHz (a clock period of less than 10 ns).

Cycles per Instruction

Historically, all early computers used many clock cycles during the execution of even the simplest instruction. During the RISC revolution, many designers focused on reducing this factor closer to the apparent minimum of 1 cycle per instruction. We will discuss some of the techniques used later in this book. Since then, CPUs that use techniques such as superscalar execution and multicore computing have reduced this even further. Such CPUs can (on average) use less than 1 cycle per instruction.

"CPI" is a throughput measure of how many instructions are completed (on average) for a given number of clocks. A CPU that can complete, on average, 2 instructions per cycle (a CPI of 0.5) may have a 20 stage pipeline, which inevitably causes a 20 cycle latency between an instruction fetch to the completion of that instruction. We ignore

those 20 cycles when we calculate CPI.

instructions executed per program

If the program you need to run is a binary executable, this number can't be changed.

However, some CPU designers have the freedom of designing a new instruction set (or at least adding a few instructions to an old instruction set).

Early CPU designers attempted to reduce this number by adding new, more complicated instructions, that did more work. (Later this idea was retroactively called "CISC"). When a given program (perhaps a benchmark program) is re-compiled for this new instruction set and executed, it requires fewer total executed instructions to finish. Alas, these more complicated instructions often require more cycles to execute -- or worse, a longer clock period, which slows down every instruction -- so the net benefit was not as great as was hoped. In a surprising number of cases, such "RICH" instructions actually made the runtime worse (longer). Benchmarking is required to see if such changes to the instruction set are worthwhile.

Some examples where it did turn out to be worthwhile:

More complicated instructions that do more work include the "load multiple" and "store multiple" instructions of the ARM processors, the "multimedia extensions" of other processors, the MAC instructions used by most DSPs, etc.

Sometimes a CPU can be tweaked in ways that fewer instructions need to be executed in a program, without adding complexity -- the "every instruction is conditional" technique used by ARM processors (the "conditional logic" was needed anyway for conditional branches); the "add more registers" and "register windowing" ideas, each of which attempts to reduce the number of register spill/reload instructions; widening the width of the data bus, so more data can be transferred per "load" or "store" instruction (also enabling wider instructions); etc.

There are a few chips that do things in a few cycles of a single "instruction" that any von Neumann CPU would require hundreds of cycles to implement -- such as content-addressible RAM.

MIPS/$

When building a computer cluster, the raw MIPS of any one chip is irrelevant. When someone needs a teraflop of performance, no one chip can do it. The person is forced to keep adding CPUs until he gets the performance he wants. There are many tricks (that we will discuss later) that slightly reduce the runtime of one program on one CPU, but make that CPU much more expensive. Rather than build a teraflop system out of a few of the lowest-runtime chips, usually people build such a system out of CPUs that take slightly longer to perform any particular task, but then these people simply use a lot more of them.

In such systems is useful if the CPUs are specifically designed to coordinate their work and synchronize rapidly.

Latency

In hard real-time systems, low latency is critical.

MIPS/mW

[1]

Most CPUs in mobile electronics -- PDAs, cell phones, laptops, wireless keyboards, MP3 players, etc. -- are underclocked.

Why do people deliberately clock them at a rate far below their potential runtime performance? Because clocking them any faster squanders battery life.

Every clock tick to a particular CPU uses up (approximately) some fixed amount of energy. If it takes (hypothetically) 900,000 clock ticks on that CPU to decode one second worth of MP3, then we maximize battery life by clocking the CPU at 0.9 MHz while playing MP3s.

Say we have some other CPU that requires 4,000,000 clock ticks to decode one second worth of MP3. Which CPU should we use? The absolute fastest MIPS rating at the maximum speed is irrelevant. The "clock ticks required to decode one second worth of MP3" is irrelevant. The better CPU for a MP3 player is the one that gives the maximum battery life, assuming we are smart enough to underclock each CPU to give its maximum battery life. Or in other words (since the amount of "work" done decoding an MP3 is fixed, and the amount of energy stored in a battery is fixed), the better CPU is the one with more MIPS/mW.

Further reading

[1] "CISC, RISC, and DSP Microprocessors" (http://www.ifp.uiuc.edu/~jones/RISCvCISCvDSP.pdf) by Douglas L. Jones 2000 "Most quoted numbers for DSP uPs not MIPS, but MIPS/$$, MIPS/mW" "Why have RISC/CISC converged?"

Microprocessor Design/Benchmarking

Common Benchmarks

- block move
- Eratosthenes sieve
- matrix multiply
- MINC/Benchmarks
- "AN910: ST7 and ST9 performance benchmarking" [1] describes a collection of short benchmark programs that measure interrupt latency and execution time and code size, and discusses architectural features that affect the scores. Eratosthenes sieve, Ackermann function, string search, block move, block translation, etc.
- "Benchmarks and Case Studies of Forth Kernels" [2] describes some very frequently used, very short code fragments.
- Wikipedia: benchmark (computing)
- Wikipedia: EEMBC Embedded Microprocessor Benchmark Consortium
- MIPS/Watt benchmarks; "Philips Challenges 8-bit MCUs" [3]; "Innovative Techniques for Extremely Low Power Consumption with 8-bit Microcontrollers" [4]
- real-time benchmark: "the number of voices of MIDI-driven OPL2-style FM synthesis (at a 48k sample rate) that each chip can perform ... the clock required for sample output has the potential to test

interrupt latency ... it scales down to the lowest PICs ... and up to the scary fast GPUs ..." -- Gwenhwyfaer 2008 [5]

Further reading

- Cyberbotics' Robot Curriculum/Cognitive Benchmarks

References

[1] http://www.st.com/stonline/books/pdf/docs/5039.pdf
[2] http://www.bradrodriguez.com/papers/moving2.htm
[3] http://www.standardics.nxp.com/support/documents/microcontrollers/pdf/article.challenge.8-bit.mcu.pdf
[4] http://atmel.com/dyn/resources/prod_documents/doc7903.pdf
[5] http://www.nabble.com/Re:-Intellasys-question-for-Jeff-Fox-p17450680.html

Microprocessor Design/Optimizations

Microprocessor Design

Once the microprocessor is designed, there is typically a large amount of room for that design to be made more efficient through optimization. The control unit specifically can be subject to logical minimizations.

As the specific requirements of each component are understood better, through simulation and prototyping, the clock speed of the system can be increased to reduce waste.

The most common operations, memory loads, memory stores, and basic arithmetic can be laid out in such a way that they can be performed quickly and easily.

The word "optimization" is likely a misnomer because it is unlikely that the best possible solution will ever be found to the complex problems that arise during microcontroller or microprocessor design. However, there are typically ways to make things better, even if they can't be made optimal.

Parallel Processing

Microprocessor Design/Multi-Core Systems

Microprocessor Design

Taking the idea of superscalar operations to the next level, it is possible (and frequently desirable) to put multiple microprocessor cores onto a single chip, and have the cores operate in parallel with one another.

Symmetric Multicore

A symmetric multicore processor is one that has multiple cores on a single chip, and all of those cores are identical.

Example: Intel Core 2:

The Intel Core 2 is an example of a symmetric multicore processor. The Core 2 can have either 2 cores on chip ("Core 2 Duo") or 4 cores on chip ("Core 2 Quad"). Each core in the Core 2 chip is symmetrical, and can function independantly of one another. It requires a mixture of scheduling software and hardware to farm tasks out to each core.

Example: Parallax Propeller:

The Parallax Propeller is an example of a symmetric multicore processor. The Parallax Propeller has 8 cores on chip, each one a 32-bit RISC processor. Each core in the Parallax Propeller chip is symmetrical, and can function independently of one another.

Asymmetric Multicore

An asymmetric multicore processor is one that has multiple cores on a single chip, but those cores might be different designs. For instance, there could be 2 general purpose cores and 2 vector cores on a single chip.

Example: Cell Processor

IBM's Cell processor, used in the Sony PlayStation 3 video game console is an asymmetrical multicore processor. The Cell has 9 processor cores on board, one general purpose processor, and 8 data-processing cores. The one multipurpose core, known as the **Power Processor Element** (PPE) controls the communication between the other cores, and distributes computing tasks to the other cores for processing. The other 8 cores are known as **Synergistic Processor Elements** (SPE), and are specially designed to have high floating-point throughput, especially with vector operations.

Example: Kilocore

Rapport's Kilocore processor, is an asymmetrical multicore processor. The Kilocore has one general purpose processor, a PowerPC processing core, and either 256 or 1024 data processing cores on-chip. The cores are designed to run at extremely low power, so the overall chip is faster and yet uses less power than typical desktop CPUs[1] .

Symmetric Multicore

A symmetric multicore processor is a processor which has multiple cores that are all exactly the same. Every single core has the same architecture and the same capabilities. An example of a symmetric multicore system is the Intel Core 2 Duo processor.

Each core has the same capabilities, so it requires that there is an arbitration unit to give each core a specific task. Software that uses techniques like **multithreading** makes the best use of a multi-core processor like the Intel Core 2.

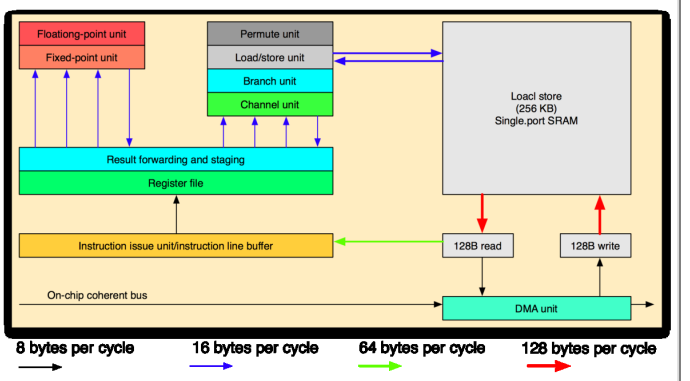
The Intel Core 2 Duo

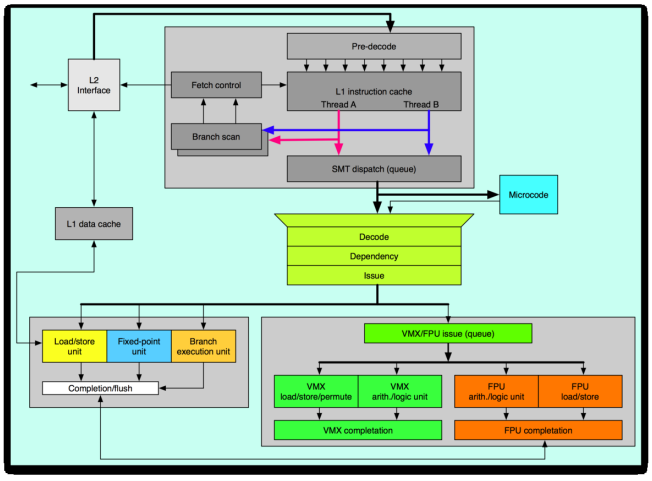
Asymmetric Multicore

In an asymmetric multicore processor, the chip has multiple cores onboard, but the cores might be different designs. Each core will have different capabilities.

Example: IBM Cell Processor

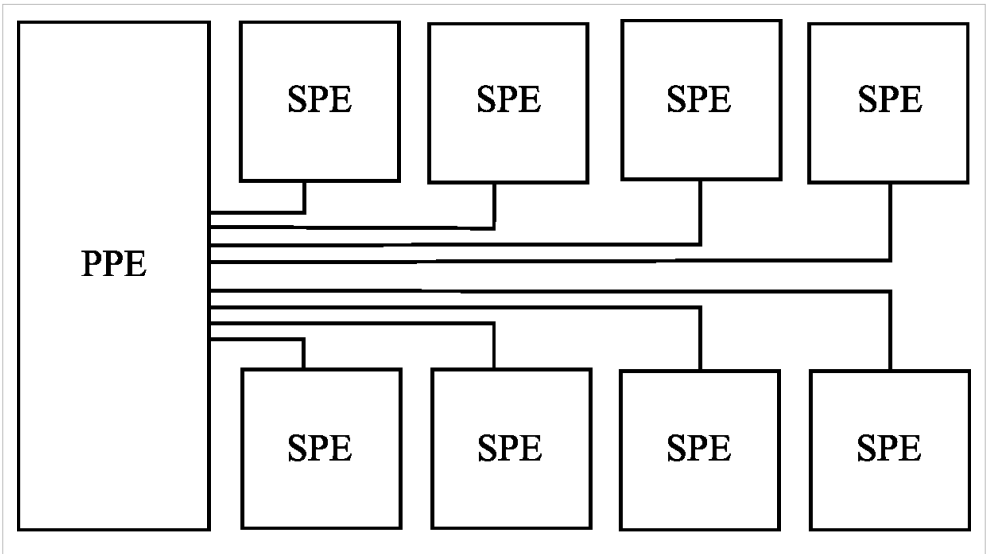
An example of an asymmetric multicore processor is the IBM Cell processor.





Block diagrams of the IBM Cell processor. The Cell processor has 8 SPE cores (left) and 1 PPE core (right). The PPE core is the primary core, and controls the behavior of the SPE cores.

The IBM Cell processor has 1 PPE that controls the chip, and 8 SPEs that are designed for high mathematical throughput. The IBM Cell processor is designed as follows:



Notice how the SPE cores only connect to the PPE, and not to each other. Notice also that the PPE core is much larger then the individual SPE cores.

further reading

[1] Tom's hardware: "IBM says Kilocore technology will outrun today's mobile processors" (http://www.tomshardware.com/news/ibm-rapport-kilocore,2575.html) 2006

Microprocessor Design/Memory-Level Parallelism

Memory-Level Parallelism

Memory-Level Parallelism (MLP) is the ability to perform multiple memory transactions at once. In many architectures, this manifests itself as the ability to perform both a read and write operation at once, although it also commonly exists as being able to perform multiple reads at once. It is rare to perform multiple write operations at once, because of the risk of potential conflicts (trying to write two different values to the same location).

Notice that this is not the same as vectorized memory operations, such as reading 4 separate but contiguous 8-bit values in a single 32-bit read.

Microprocessor Design/Out Of Order Execution

In a superscalar or similar processor design, there are multiple execution units that can be used to process pieces of data simultaneously. However, these execution units are not always used at the same time, and some processing power is lost. Sometimes, it is possible to feed instructions to all the execution units if we take the instructions out of their original order. **Out of order execution** (OOOE) is when a processor is capable of executing instructions out of their original order, in an attempt to do more work in parallel, and execute programs more quickly.

Hazards

OOOE comes with some significant hazards, and the hazard detection units in these processors are not trivial. The dependencies of all instructions need to be determined, and instructions cannot execute before or at the same time as instructions on which they are dependent.

Example: Intel Hyperthreading

Hyperthreading is the name for a technology developed by Intel for use in the Pentium 4 chip. Hyperthreading works by duplicating some architectural components of the processor, such as the status flags, the control registers, and the general purpose registers. Hyperthreading does not, however, duplicate any of the execution units.

In a hyperthreaded system, it appears to the operating system that there are two separate processors, when there is only one processor. The OOOE engine feeds instructions from the two separate execution threads to the execution cores, to try and keep all of the cores simultaneously busy. In general hyperthreading increases performance, although in some instances this additional complexity actually decreased performance.

Support Software

Microprocessor Design/Assembler

Microprocessor Design

Simply having a new microprocessor is not much of a benefit, unless you have a way to program it. The most simple and direct way to program a microprocessor is through the use of an assembler. An assembler converts mnemonics into corresponding machine code instructions. Once you have an ISA, it's a trivial task to map mnemonics to the various instruction opcodes.

Once an ISA is finalized, the design work can usually be split into two teams: a hardware team to design the datapath and control units, and a software team to write an assembler and other programs, such as a simulator and a compiler. This is not the way it is always done, however, as a single group of people is perfectly capable of doing both sets of tasks.

Microprocessor Design/Simulator

Microprocessor Design

Simulators are software programs that have grown in popularity among design groups in recent years. Once an ISA is finalized, and the basics of the datapath are mapped out (especially the timing and delays), a simulator can be a very valuable project to work on.

A simulator allows software for your new microprocessor to be tested on a separate computer. This means that people can write and test software for your new processor even before you have finished designing it!

Simulators have lead to a fascinating new realm of productivity known as *hardware software co-design*.

Microprocessor Design/Compiler

With an assembler written, it is typically a good idea (although not always) to write a high-level language compiler for your new processor. Typically the high-level language in these situations is C because of its small number of builtin constructions, and the close relationship that C shares with the underlying assembly language.

Compilers help to speed up the development process, so that more complicated software can be written without the tedium of writing large assembly language programs. Another benefit to this is that there are a number of pre-existing tools for use with higher-level languages, such as simulators and debuggers that can increase the efficiency of your software team.

Further reading

- Compiler Construction discusses how to write a compiler from scratch
- GNU Compiler Collection has a section on how to adapt the GCC "backend" to your particular processor.

Microprocessor Production

Microprocessor Design/FPGA

Microprocessor Design

For more information about FPGAs, Verilog and VHDL, see Programmable Logic.

Field-Programmable Gate Arrays (FPGA) are programmable logic elements. FPGAs can be designed using a hardware description language (HDL) such as Verilog or VHDL, and that design can be mapped to a hardware design by the HDL synthesizer. FPGAs are the successors of their previous similar components, PLAs and PALs, used at the first steps of the programmable logic era.

FPGAs are quick to design, and because they are reprogrammable, troubleshooting is quick and easy.

FPGAs are useful for designing microcontrollers, which is why we have discussed HDL implementations of various components in the text of this book. In this chapter we will will discuss the implementation of a microcontroller in HDL, and some of the consequences of that implementation.

Dozens of FPGA CPU designs are available for download and tinkering. An appendix to this book, Microprocessor Design/Resources, lists details on how to get them.

Microprocessor Design/Wire Wrap

Microprocessor Design

Historically, most of the early CPUs were built by attaching integrated circuits (ICs) to circuit boards and wiring them up.

Nowadays, it's much faster to design and implement a new CPU in a FPGA -- the result will probably run faster and use less power than anything spread out over multiple ICs.

However, some people still design and build CPUs the old-fashioned way. Such a CPU is sometimes called a "home brew CPU" or a "home built CPU".

Some people feel that physically constructing a CPU in this way, since it allows students to probe the inner workings of the CPU, it helps them "Touch the magic"[1] , helps them learn and understand the underlying electronics and hardware.

Overview

(FIXME: How do I get the picture from http://en.wikipedia.org/wiki/Image:YUNTEN.gif *to show up here?)*

A homebrew CPU is a central processing unit constructed using a number of simple integrated circuits, usually from the 7400 Series. When planning such a CPU, the designer must not only consider the hardware of the device but also the instructions the CPU will have, how they will operate, the bit patterns for each one, and their mnemonics. Before the existence of computer based circuit simulation many commercial processors from manufacturers such as Motorola were first constructed and tested using discrete logic (see Motorola 6809). [2]

Although no limit exists on data bus sizes when constructing such a CPU, the number of components required to complete a design increases exponentially as bus size gets wider. Common physical data bus sizes are 1-bit, 4-bits, 8-bits, and 16-bits, although incomplete design documents exist for a 40-bit CPU.

A microcoded CPU may be able to present a significantly different instruction set to the application programmer than seems to be directly supported by the hardware used to implement it. For example, the 68000 presented a 32-bit instruction set to the application programmer -- a 32-bit "add" was a single instruction -- even though internally it was implemented with 16-bit ALUs.

For example, w:serial computers, even though they do calculations one bit per clock cycle, present a instruction set that deals with much wider words -- often 12 bits (PDP-14), 24 bits (D-17B), or even wider -- 39 bits (Elliott 803).

Notable Homebrew CPUs

The Magic-1 is a CPU with an 8-bit data bus and 16-bit address bus running at about ~~3.75MHz~~ 4.09 Mhz. [3]

The Mark I FORTH also has a 8-bit data bus and 16-bit address bus, but runs at 1MHz. [4]

The V1648CPU is a CPU with a 16-bit data bus and 48-bit address bus that is currently being designed. [5]

Parts

...

chips

bus

Practically all CPU designs include several 3-state buses -- an "address bus", a "data bus", and various internal buses.

A 3-state bus is functionally the same as a multiplexer. However, there is no physical part you can point to and say "that is the multiplexer" in a 3-state bus; it's a pattern of activity shared among many parts. The only reason to use a 3-state bus is when it requires fewer chips or fewer, shorter wires, compared to an equivalent multiplexer arrangement. When you want to select between very few pieces of data that are close together, and most of that data is stored on a chip that only has 2-state outputs, it may require fewer chips and less wiring to use actual multiplexer chips. When you want to select between many pieces of data (one of many registers, or one of many memory chips, etc.), or many of the chips holding that data already have 3-state outputs, it usually requires fewer chips to use a 3-state bus (even counting the "extra" 3-state buffer between the bus and each thing that doesn't already have 3-state outputs).

A typical register file connected to a 3-state 16-bit bus on a TTL CPU includes:

- octal 2-state output registers (such as 74x273), 2 chips per 16-bit register
- octal 3-state non-inverting buffers (such as 74x241), 2 chips per 16-bit register per bus
- a demultiplexer with N inputs (driven by microcode) and 2^N output wires that select the 3-state buffers of one of up to 2^N possible things that can drive the bus, 1 chip per bus.

Later we discuss a other shortcuts that may require fewer chips.

74181

Like many historically important commercial computers, many home-brew CPUs use some version of the 74181, the first complete ALU on a single chip.[6] (Versions of the 74181 include the 74F181, the 40181[*citation needed*], the 74AS181, the 72LS181, the 74HCT181, etc.). The 74181 is a 4-bit wide ALU can perform all the traditional add / subtract / decrement operations with or without carry, as well as AND / NAND, OR / NOR, XOR, and shift.

A typical home-brew CPU uses 4 of these 74181 chips to build an ALU that can handle 16 bits at once. The simplest home-brew CPUs have only one ALU, which at different times is used to increment the program counter, do arithmetic on data, do logic operations on data, and calculate addresses from base+offset.

Some people who build home-brew CPUs attempt to "save chips" by building that one ALU of less than the largest word size (which is often 16 bits in TTL computers). Unfortunately, this adds complexity elsewhere, and may actually increase the total number of chips needed.[7] [8] [9]

The simplest (and slowest) 16-bit TTL ALU wires the carry-out of each 74181 chip to the carry-in of the next, creating a ripple-carry adder.

Historically, some version of the look ahead carry generator 74182 was used to speed up "add" and "subtract" to be about the same speed as the other ALU operations.

Historically, some people who built TTL CPUs put two or more independent ALU blocks in a single CPU -- a general-purpose ALU for data calculations, a PC incrementer, an index register incrementer/decrementer, a base+offset address adder, etc.

We discuss ripple-carry adders, look-ahead carry generators, and their effects on other parts of a CPU at Microprocessor Design/Add and Subtract Blocks.

alternatives to 74181

Some people find that '181 chips are becoming hard to find.

Quite a few people building "TTL CPUs" use GAL chips (which can be erased and reprogrammed). [10] A single GAL20V8 chip can replace a 74181 chip.[11] Often another GAL chip can replace 2 or 3 other TTL chips.

Other people building "TTL CPUs" find it more magical to build a programmable machine entirely out of discrete non-programmable chips. Are there any reasonable alternatives to the '181 for building an ALU out of discrete chips? The Magic-1 uses 74F381s and a 74F382 ALUs;[12] is there any variant of the '381 and '382 chips that are any easier to find than a '181? ... the 74HC283, 74HCT283, MC14008 chips only add; they don't do AND, NAND, etc. ...

One could build the entire CPU -- including the ALU -- out of sufficient quantities of the 74153 multiplexer.[13]

One designer "built-from-scratch" a 4-bit ALU that does add, subtract, increment, decrement, "and", "or", "xor", etc. -- roughly equivalent to the 4-bit 74181 -- out of about 14 simple TTL chips: 2-input XOR, AND, OR gates.[14]

Another designer has posted a 8-bit ALU design that has more functionality than two 74181 chips -- the 74181 can't shift right -- built from 14 complex TTL chips: two 74283 4-bit adders, some 4:1 mux, and some 2:1 mux.[15]

The designers of the LM3000 CPU posted an ALU design that has less functionality than the 74181. The 8 bit "ALU" in the LM3000 can't actually do any logical operations, only "add" and "subtract", built from two 74LS283 4-bit adders and a few other chips. Apparently those "logical" operations aren't really necessary.[16]

other parts

solderless breadboard approach

Solderless breadboards are perhaps the fastest way to build experimental prototypes that involve lots of changes.

For about a decade, every student taking the 6.004 class at MIT was part of a team -- each team had one semester to design and build a simple 8 bit CPU out of 7400 series integrated circuits.[17] These CPUs were built out of TTL chips plugged into several solderless breadboards connected with lots of 22 AWG (0.33 mm^2) solid copper wires.[18]

wire-wrap

Traditionally, minicomputers built from TTL chips were constructed with lots of wire-wrap sockets (with long square pins) plugged into perfboard and lots of wire-wrap wire, assembled with a "wire-wrap pencil" or "wire-wrap gun".

stripboard

More recently, some "retrocomputer" builders have been using standard sockets plugged into stripboard and lots of wire-wrap wire, assembled with solder and a soldering iron.[19]

Tools

...

Design Tips

There are many ways to categorize CPUs. Each "way to categorize" represents a design question, and the various categories of that way represent various possible answers to that question that needs to be decided before the CPU implementation can be completed.

One way to categorize CPU that has a large impact on implementation is: "How many memory cycles will I hold one instruction before fetching the next instruction?"

- 0: load-instruction on every memory cycle (Harvard architecture)
- 1: At most 1 memory cycle between each load-instruction memory cycle (load-store architecture)
- more: some instructions have 2 or more memory cycles between load-instruction memory cycles (memory-memory architecture)

Another way to categorize CPUs is "Will my control lines be controlled by a flexible microprogramming, a fixed control store, or by hard-wired control decoder that directly decodes the instruction?"

The load-store and memory-memory architectures require a "instruction register" (IR). At the end of every instruction (and after coming out of reset), the next instruction is fetched from memory[PC] and stored into the instruction register, and from then on the information in the instruction register (directly or indirectly) controls everything that goes on in the CPU until the next instruction is stored in the instruction register.

For homebrew CPUs, the 2 most popular architectures are[*citation needed*]:

- direct-decode Harvard architecture
- flexible microprogramming that supports the possibility of memory-memory architecture.

Another way to categorize CPUs is "How many sub-states are in a complete clock cycle?"

Many textbooks imply that a CPU has only one clock signal -- a bunch of D flip-flops each hold 1 bit of the current state of the CPU, and those flip-flops drive that state out their "Q" output. Those flip-flops always hold their internal state constant, except at the instant of the rising edge of the one and only clock, where each flip-flop briefly "glances" at their "D" input and latches the new bit, and shortly afterwards (when the new bit is different from the old bit) changes the "Q" output to the new bit.

Single clock signals are nice in theory. Alas, in practice we can never get the clock signal to every flip-flop precisely simultaneously -- there is always some clock skew (differences in propagation delay). One way to avoid these timing issues is with a series of different clock signals.[20] Another way is to use enough power[21] and carefully design a w: clock distribution network (perhaps in the form of an w: H tree) with w: timing analysis to reduce the clock skew to negligible amounts.

Relay computers are forced to use at least 2 different clock signals, because of the "contact bounce" problem.

Many chips have a single "clock input" pin, giving the illusion that they use a single clock signal -- but internally a "clock generator" circuit converts that single external clock to the multiple clock signals used by the chip.

Many historically and commercially important CPUs have many sub-states in a complete clock cycle, with two or more "non-overlapping clock signals". Most MOS ICs used dual clock signals (a two-phase clock) in the 1970s[22]

shortcuts

Building a CPU from individual chips and wires takes a person a long time. So many people take various shortcuts to reduce the amount of stuff that needs to be connected, and the amount of wiring they need to do.

- 3-state bus rather than 2-state bus often requires fewer and shorter connections.
- Rather than general-purpose registers that can be used (at different times) to drive the data bus (during STORE) or the address bus (during indexed LOAD), sometimes it requires less hardware to have separate address registers and data registers and other special-purpose registers.
- If the software guy insists on general-purpose registers that can be used (at different times) to drive the data bus (during STORE) or the address bus (during indexed LOAD), it may require less hardware to emulate them: have all programmer-visible registers drive only one internal microarchitectural bus, and (at different times) load the microarchitectural registers MAR and MDR from that internal bus, and later drive the external address bus from MAR and the external data bus from MDR. This sacrifices a little speed and requires more microcode to make it easier to build.
- Rather than 32-bit or 64-bit address and data registers, it usually requires less hardware to have 8-bit data registers (occasionally combining 2 of them to get a 16-bit address register).
- If the software guy insists on 16-bit or 32-bit or 64-bit data registers and ALU operations, it may require less hardware to emulate them: use multiple narrow micro-architectural registers to store each programmer-visible register, and feed 1 or 4 or 8 or 16 bits at a time through a narrow bus to the ALU to get the partial result each cycle, or to sub-sections of the wide MAR or MDR. This sacrifices a little speed (and adds complexity elsewhere) to make the bus easier to build. (See: 68000, as mentioned above)
- Rather than many registers, it usually requires less hardware to have fewer registers.
- If the software guy insists on many registers, it may require less hardware to emulate some of them (like some proposed MMIX implementations) or perhaps all of them (like some PDP computers): use reserved locations in RAM to store most or all programmer-visible registers, and load them as needed. This sacrifices speed to make the CPU easier to build. Alas, it seems impossible to eliminate all registers -- even if you put all programmer-visible registers in RAM, it seems that you still need a few micro-architectural registers: IR (instruction register), MAR (memory address register), MDR (memory data register), and ... what else?
- Harvard architecture usually requires less hardware than Princeton architecture. This is one of the few ways to make the CPU simpler to build *and* go faster.

Harvard architecture

The simplest kinds of CPU control logic use the Harvard architecture, rather than Princeton architecture. However, Harvard architecture requires 2 separate storage units -- the program memory and the data memory. Some Harvard architecture machines, such as "Mark's TTL microprocessor", don't even have an instruction register -- in those machines, the address in the program counter is always applied to the program memory, and the data coming out of the program memory directly controls everything that goes on in the CPU until the program counter changes. Alas, Harvard architecture makes storing new programs into the program memory a bit tricky.

microcode architecture

See Microprocessor Design/Microcodes.

Assembly Tips

...

> "I don't recommend that anybody but total crazies wirewrap their own machines out of loose chips anymore, although it was a common enough thing to do in the mid- to late Seventies". -- Jeff Duntemann [23]

Programming Tips

...

Further Reading

[1] "Touch the magic. By this I meant to gain a deeper understanding of how computers work" -- Bill Buzbee (http://www.homebrewcpu.com/)

[2] "To evaluate the 6800 architecture while the chip was being designed, Jeff's team built an equivalent circuit using 451 small scale TTL ICs on five 10 by 10 inch (25 by 25 cm) circuit boards. Later they reduced this to 114 ICs on one board by using ROMs and MSI logic devices." -- w:Motorola_6800#Development_team

[3] http://www.homebrewcpu.com/

[4] http://www.holmea.demon.co.uk/Mk1/Architecture.htm

[5] http://ixeelectronics.com/Development/Chipset/CPU_Goals.html

[6] "The 74181 is a bit slice arithmetic logic unit (ALU)... The first complete ALU on a single chip ... Many computer CPUs and subsystems were based on the '181, including ... the ... PDP-11 - Most popular minicomputer of all time" -- Wikipedia:74181

[7] "My Home-Built TTL Computer Processor (CPU)" (http://cpuville.com/Main_2.htm) by Donn Stewart

[8] "The basic algorithm executed by the instruction execution unit is most easily expressed if a memory address fits exactly in a word." -- "The Ultimate RISC" (http://www.cs.uiowa.edu/~jones/arch/risc/) by Douglas W. Jones

[9] "it just really sucks if the largest datum you can manipulate is smaller than your address size. This means that the accumulator needs to be the same size as the PC -- 16-bits." -- "Computer Architecture" (http://david.carybros.com/html/computer_architecture.html)

[10] Andrew Holme. "Mark 2 FORTH Computer" (http://www.holmea.demon.co.uk/Mk2/Architecture.htm)

[11] GALU - A Gate Array Logic based ALU IC. (http://sites.google.com/site/libby8dev/ga)

[12] Bill Buzbee. "Magic-1 Microarchitecture" (http://www.homebrewcpu.com/microarchitecture.htm).

[13] Dieter Mueller. "Multiplexers: the tactical Nuke of Logic Design" (http://www.6502.org/users/dieter/a1/a1_4.htm) 2004.

[14] Rodney Moffitt. Micro Programmed Arithmetic Processor (http://rod.info/MicroProgram). 55 TTL chips. The core 4-bit adder/subtracter has about 7 SSI chips. The ALU has about 7 additional SSI chips of logic around that core to support "and", "or", "xor", "increment", "decrement". An instruction register and a micro-programmed sequencer around the ALU handle (4-bit) "multiply" and "divide".

[15] Dieter Mueller. ALU with Adder (http://www.6502.org/users/dieter/a1/a1_6.htm). 2004.

[16] LM3000 CPU (http://wiki.bennington.edu/mediawiki/index.php/LM3000)

[17] the VHS (http://sub-zero.mit.edu/fbyte/hacks/vhs/), a 32 bit CPU built by Kevin McCormick, Colin Bulthaup, Scott Grant and Eric Prebys for their MIT 6.004 class.

[18] 6.004 Contest Photos (http://www.bunniestudios.com/bunnie/6004/contest.html) (http://www.xenatera.com/bunnie/6004/contest.html)

[19] "Libby8" neo-retro computer (http://sites.google.com/site/libby8dev/libby8/circuit) by Julian Skidmore

[20] Bill Buzbee. Magic-1 Homebrew CPU: Clocks (http://www.homebrewcpu.com/microarchitecture.htm#Clocks)

[21] "Intel's Atom Architecture: The Journey Begins" (http://www.anandtech.com/showdoc.aspx?i=3276&p=14) by Anand Lal Shimpi, 2008. In a large microprocessor, the power used to drive the clock signal can be over 30% of the total power used by the entire chip.

[22] Wikipedia:Two-phase_clock#cite_ref-MC6870_0-0

[23] http://artematrix.org/Projects/TTL.processor/processor.design.htm

comparisons

- Svarychevski Michail Aleksandrovich. "Homemade CPU – from scratch" (http://3.14.by/en/read/homemade-cpus). Briefly compares a few notable hobbyist-built CPUs.

relay computers

- Harry Porter's Relay Computer (http://web.cecs.pdx.edu/~harry/Relay/) (415 Relays, all identical 4PDT)
- "Relay Computer Two" (http://www.electronixandmore.com/project/relaycomputertwo/index.html) by Jon Stanley (281 relays, of 2 types: 177 SPDT, and 104 4PDT)
- Zusie - My Relay Computer (http://nablaman.com/relay/) by Fredrik Andersson (uses around 330 relays, of 2 types: 4-pole and 6-pole double-throw relays, plus ~30 integrated circuits for RAM and microcode)
- relay computers (http://www.relaiscomputer.de/) by Kilian Leonhardt (in German): a "large computer" with around 1500 relays and a program EEPROM, and a "small computer" with 171 relays.
- DUO 14 PREMIUM (http://web.me.com/teisenmann/ostracod/relay.html) by Jack Eisenmann (around 50 relays, including 4 addressable "crumbs" of RAM where each crumb is 2 bits, plus 48 bits of program ROM in 6x8-switch DIP switches. The only semiconductor components: 555 timer, decade counter, and transistors in the clock generator. Each command has 6 bits, and the 8 commands in the program ROM are selected by a 3-bit program counter).
- Wikipedia: Z3 (computer), designed by Konrad Zuse, the world's first working programmable, fully automatic computing machine. built with 2,000 relays.
- Z3 Nachbau (http://www.horst-zuse.homepage.t-online.de/z3-nachbau-2001.html) (http://www.zib.de/zuse/Inhalt/Rep/Z3/Z3Rep/index.html), Horst Zuse's (Konrad Zuse's son) and Raul Rojas' 2001 reconstruction of the classic Z3. The 32-word, 22-bit-wide memory is also constructed entirely from relays, about 700 relays. (in German)
- Horst Zuse's new Z3 reconstruction (http://z3-computer.de/): Created 2010 for the 100 year anniversary of Konrad Zuse's birth. About 2500 modern relays. (in German)

TTL computers

- A Minimal TTL Processor for Architecture Exploration (http://www.zetetics.com/bj/papers/piscedu2.htm) by Bradford J. Rodriguez (aka PISC, the Pathetic Instruction Set Computer)
- Wikipedia:Apollo Guidance Computer
- V1648 (http://ixeelectronics.com/Development/Chipset/CPU_Goals.html): (16 bit data) (48 bit address bus?)
- "the Ultimate RISC" and "the Minimal CISC" (http://www.cs.uiowa.edu/~jones/arch/)
- alt.comp.hardware.homebuilt FAQ (http://www.faqs.org/faqs/homebuilt-comp-FAQ/index.html)
- Mark's TTL microprocessor (http://web.archive.org/web/20050316211858/http://www.venturalink.net/~jamesc/ttl/) (uses only 8 chips ... "Without using the two PALs I used, it would be 16 chips.") *(is there a better URL for this?)*
- "Prehistoric Cpu's & Octal Amps" (http://www.jetnet.ab.ca/users/bfranchuk/) (18 bit data bus? 24 bit data bus?)

reviews

- other homemade CPUs (http://www.holmea.demon.co.uk/Links.htm#Homemade)
- yet more homemade CPUs (http://david.carybros.com/html/computer_architecture.html#simple_cpu)

TTL computers

- "Viktor's Amazing 4-bit Processor" (http://www.vttoth.com/vicproc.htm) ... can re-program in-circuit using manual switches. About 90 chips.
- Galactic 4 bit CPU (http://www.galacticelectronics.com/Simple4BitCPU.HTML) by Jon Qualey. Two, 2716 EPROMs are used to store the micro-instruction code and two, 2114 static RAMs are used for program memory. 25 ICs in all, 74LS TTL.

discrete transistor computers

- MT15 by Dieter Mueller (http://freenet-homepage.de/dieter.02/mt15.htm) is built almost entirely out of (around 3000) individual SMT transistors ... also has some essays on microprogramming and ALU design (http://freenet-homepage.de/dieter.02/).
- The Q1 Computer (http://joewing.net/hardware/q1/) by Joe Wingbermuehle. Built almost entirely out of (3105) individual through-hole PN2222A transistors. "Clock phases are used so that transparent latches can be used for registers to reduce transistor count at the price of speed." 8 bit data bus, 16 bit address bus.

TTL computers

- LM3000 CPU (http://wiki.bennington.edu/mediawiki/index.php/LM3000) designed and built by five students at Bennington College, Vermont, using fifty-three integrated circuits.
- The D16/M by John Doran (http://www.timefracture.org/D16.html) is a 16-bit digital computer implemented with SSI and MSI HCMOS integrated logic and constructed using wire-wrap techniques. Its timing and control unit is microprogrammed (fully horizontal, with a 72-bit control word).

pneumatic computers

- "8 bit processor using logic gates made of pneumatic valves" (http://blog.makezine.com/archive/2009/09/claims_of_pnuematic_processor_full.html) by Minsoung Rhee and Mark Burns

TTL computers

- BMOW 1 (Big Mess o' Wires) (http://www.stevechamberlin.com/) by Steve Chamberlin is an 8 bit CPU built from discrete 7400-series logic, and a few 22V10 and 20V8 GALs. All the digital electronics on a single large Augat wire-wrap board to interconnect the 50 or so chips. BMOW 1 contains roughly 1250 wires connecting the components. All data busses are 8 bit; the address bus is 24 bit. 3 parallel microcode ROMs generate the 24 bit microcode word. VGA video output is 512×480 with two colors, or 128×240 with 256 colors. The microcode emulates a 6502 (more or less). Uses two 4-bit 74LS181s to form the core 8 bit ALU.
- "Asychronous 40-bit TTL CPU" (http://www.hanssummers.com/ttlcpu.html) by Hans Summers 1992
- "a proprietary 8-bit engine built out 3 PROM's and a few dozen TTL chips" (http://www.laughtonelectronics.com/arcana/DiabloCPU.html) as described by Jeff Laughton.
- "One-bit Computing at 60 Hertz" (http://www.laughtonelectronics.com/arcana/One-bit computer.html): a tiny computer made from an EPROM and a few logic chips; designed by Jeff Laughton.
- "Bride of Son of Cheap Video - the KimKlone" (http://www.laughtonelectronics.com/arcana/BrideOfSonPg1.html): TTL chips and a EPROM add extra programmer-visible registers and instructions to a microcontroller (a 65C02).

- The MyCPU - Project (http://www.mycpu.eu/) (http://mycpu.selfhost.it/): "everybody is invited to participate and contribute to the project." The CPU is built from 65 integrated circuits on 5 boards. 1 MByte bank switched RAM. Originally developed by Dennis Kuschel. Apparently several MyCPU systems have been built? One MyCPU system runs a HTTP web server; another MyCPU system runs a (text-only) web browser).
- HJS22 - a homebrew TTL computer. (http://www.ibmsystem3.nl/hjs22/) Nice front panels with lots of lights and switches.
- The Electronics Australia EDUC-8 microcomputer (http://www.ljw.me.uk/educ8/): "one of the first build-it-yourself microcomputers". "The internal implementation is bit-serial which gives good economy of components as most data paths are only 1 bit wide."
- "Homebrew CPUs/Low Level Design" (http://electronics.stackexchange.com/questions/18803/homebrew-cpus-low-level-design) recommends a few books with low-level TTL CPU design information.

Microprocessor Design/Photolithography

Microprocessor Design

The current state-of-the-art process for manufacturing processors and small ICs in general is to use **photolithography**. Photolithography is a complicated multi-step process.

Wafers

A **wafer** is a large circular disk, typically made of doped silicon. Each wafer can hold multiple chips arranged like tiles. The number of chips per wafer is known as the **yield**.

Basic Photolithography

In photolithography, there are typically two important chemicals: an acid and a resist. A photo-negative of the design is exposed to light, and the pattern is projected onto the wafer. Resist is applied to the wafer, and it sticks to the portions of the wafer that are exposed to light. Once the resist is applied to the wafer, it is dipped in the acid. The acid eats away a layer of everything that is not covered in resist. ~~After the top layer has been disolved, the wafer is washed (to remove any remaining acid and resist), and a new layer of doped silicon is applied to the top of the wafer. Once the new layer of silicon has been applied, the process is repeated again.~~

The first two applications of resist are used to convert thin, carefully shaped regions of the base silicon wafer into n-type and p-type w:doping (semiconductor) (no net material is added or taken away after these steps). *Wait up -- I thought doping occurred after the polysilicon was added? Is that an additional doping stage, or is doping not really the first 2 stages?*

After that, layers of polysilicon, silicon oxide, and metal are added, coating the entire wafer. After each layer of desired material is added, resist and acid are used to "pattern" the layer, keeping the desired regions and removing the undesired regions of that layer.

packaging

After all the layers specified by the design have been applied, the wafer is "diced" into individual rectangular "die". Then each die packaged.

... does testing happen before the wafer is diced? Before and after? ...

further reading

- MOSIS (Metal Oxide Semiconductor Implementation Service [1]) is probably the oldest (1981) integrated circuit (IC) foundry service. Many VLSI students have sent their chips to MOSIS for fabrication.

References

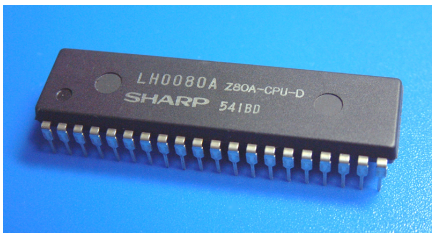
[1] http://www.mosis.com/

Microprocessor Design/Sockets and interfacing

Microprocessor Design

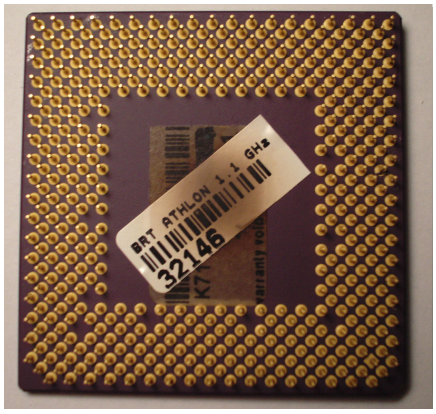
Form Factors

A matter that is peripheral to the subject of microprocessor design, but not wholely unrelated is the subject of form factors, sockets, and interfacing. When it comes to microprocessors and microcontrollers there is no standard size or shape, no standard connectors, etc. An Intel Pentium chip cannot plug into the same socket as an AMD Athalon chip, even though they are both IA32 chips, and both of them can run the same software. The size, shape, number of connectors and orientation of the connectors are known collectively as the **form factor** of the chip. Each separate form factor requires a specific interface for the chip to connect to called a **socket**.



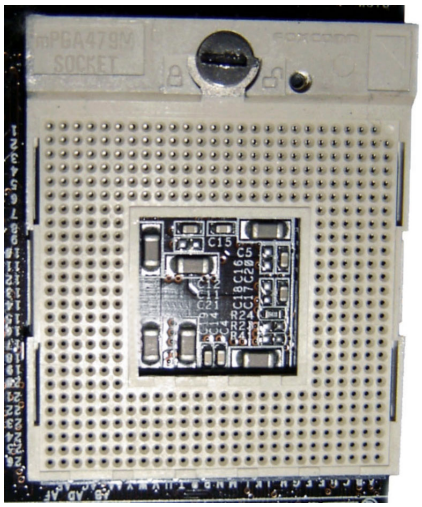
A Sharp Z80 microprocessor, with a Dual-Inline Package (DIP) with 40 pins.

Connectors



The underside of an AMD Athlon processor, showing the connector pins.

Sockets



A Socket 479, one of many different and incompatible socket types.

Advanced Topics

Microprocessor Design/Microcodes

Microprocessor Design

RISC units are typically faster and more efficient than CISC units. For this reason, many CISC processors have complicated instruction decoders that actually convert the CISC machine code into a RISC-like set of internal instructions known as **microcodes**. These microcodes are then fed into the internal core of the processor, which is based on the RISC design.

The most common way[*citation needed*] to implement memory-memory architecture CPUs (even with single-chip microprocessors, not just wire-wrapped machines) uses a small "control store" ROM. The output data bits of the control store drive a "pipeline register". The clock signal determining the cycle time of the system primarily clocks this pipeline register. The bits stored in the pipeline register directly control everything that goes on in the CPU.

Some of the bits in the pipeline register do nothing but drive some of the address bits of the control store. Those bits -- that sub-field of the pipeline register -- is sometimes called the "microprogram counter", even though it is merely a latch -- typically the control store is programmed such that those bits increment on every clock cycle, and reset to zero when a new instruction is loaded into the instruction register. The instruction register directly drives some of the address lines of the control store ROM. A few more address lines of the control store ROM are driven by status bits such as the Z flag and the C flag.

Some CPUs, such as the ECOMIPS[1] , the Intel Core 2 and the Intel Xeon,[2] use "writable microcode" -- rather than storing the microcode in ROM or hard-wired logic, the microcode is stored in a RAM called a Writable Control Store or WCS.

further reading

- "Viktor's Amazing 4-bit Processor" [3] has microcode that he says *could* have been implemented with about 90 diodes in a traditional diode matrix; but instead he implemented microcode with a Flash memory chip he can re-program in-circuit using manual switches.
- MT15 by Dieter Mueller [4] uses transistors instead of diodes in a big AND-OR PLA (programmable logic array) matrix to implement the microcode.

[1] "ECOMIPS: An Economic MIPS CPU Design on FPGA" (http://www.lixizhi.net/download/xizhil_ecomips.pdf) by Xizhi Li and Tiecai Li

[2] Wikipedia: microcode#Writable_control_stores

[3] http://www.vttoth.com/vicproc.htm

[4] http://freenet-homepage.de/dieter.02/mt15.htm

Microprocessor Design/Cache

Cache

A cache is a small amount of memory which operates more quickly than main memory. Data is moved from the main memory to the cache, so that it can be accessed faster. Modern chip designers put several caches on the same die as the processor; designers often allocate more die area to caches than the CPU itself. Increasing chip performance is typically achieved by increasing the speed and efficiency of chip cache.

Cache works by storing a small subset of the external memory contents, typically out of it's original order. Data and instructions that are being used frequently, such as a data array or a small instruction loop, are stored in the cache and can be read quickly without having to access the main memory. Cache runs at the same speed as the rest of the processor, which is typically much faster than the external RAM operates at. This means that if data is in the cache, accessing it is faster than accessing memory.

Cache helps to speed up processors because it works on the **principal of locality**.

Principal of Locality

There are two types of locality, **spatial** and **temporal**. Modern computer programs are typically loop-based, and therefore we have two rules about locality:

Spatial Locality

> When a data item is accessed, it is likely that data items in sequential memory locations will also be accessed. Consider the traversal of an array, or the act of storing local variables on a stack. In these cases, when one data item is accessed, it is a good idea to load the surrounding memory area into the cache at the same time.

Temporal Locality

> When data item is accessed, it is likely that the same data item will be accessed again. For instance, variables are typically read and written to in rapid succession. If is a good idea to keep recently used items in the cache, and not over-write data that has been recently used.

Hit or Miss

A **hit** when talking about cache is when the processor finds data in the cache that it is looking for. A **miss** is when the processor looks for data in the cache, but the data is not available. In the event of a miss, the cache controller unit must gather the data from the main memory, which can cost more time for the processor.

Flushing the Cache

When the processor needs data, it looks in the cache. If the data is not in the cache, it will then go to memory to find the data. Data from memory is moved to the cache and then used by the processor. Sometimes the entire cache contains useless or old data, and it needs to be **flushed**. Flushing occurs when the cache controller determines that the cache contains more potential misses than hits. Flushing the cache takes several processor cycles, so much research has gone into developing algorithms to keep the cache up to date.

Cache Hierarchy

Cache is typically divided between multiple levels. The most common levels are L1, L2, and L3. L1 is the smallest but the fastest. L3 is the largest but the slowest. Many chips do not have L3 cache. Some chips that do have an L3 cache actually have an external L3 module that exists on the motherboard between the microprocessor and the RAM.

Size of Cache

There are a number of factors that affect the size of cache on a chip:

1. Moore's law provides an increasing number of transistors per chip. After around 1989, more transistors are available per chip than a designer can use to make a CPU. These extra transistors are easily converted to large caches.
2. Processor components become smaller as transistors become smaller. This means there is more area on the die for additional cache.
3. More cache means fewer delays in accessing data, and therefore better performance.

Because of these factors, chip caches tend to get larger and larger with each generation of chip.

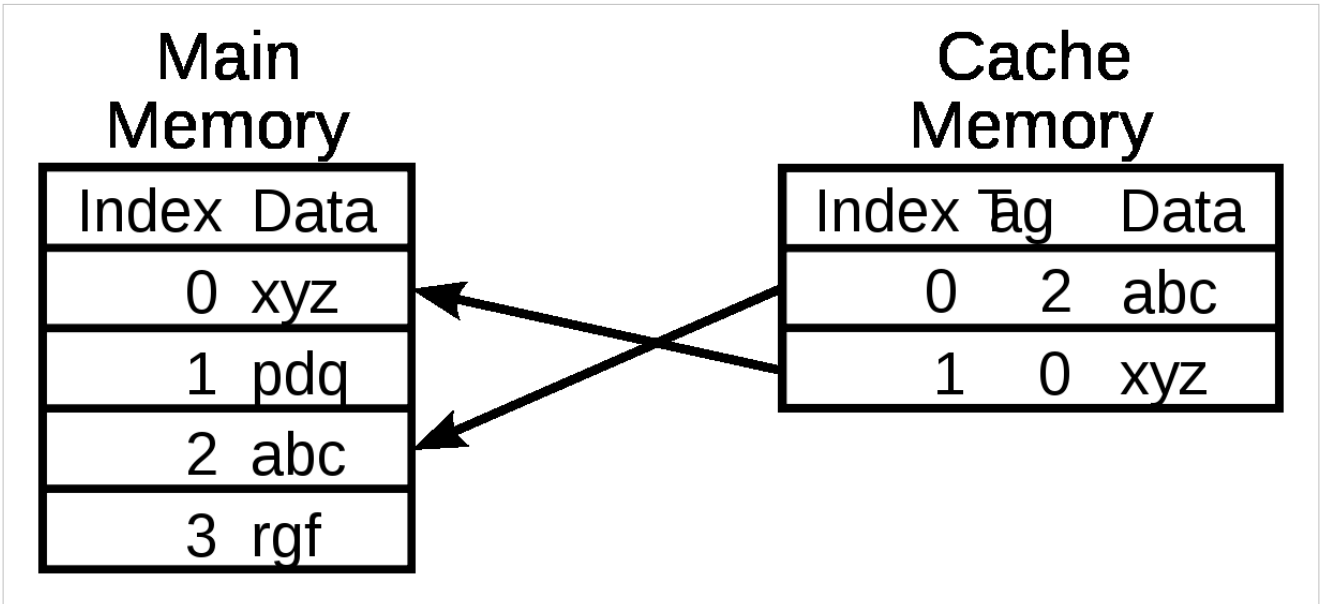
The Pentium Pro chip was one of the largest microprocessors ever manufactured. It was so large because it contained the largest cache of any chip at the time.

Cache Tagging

Cache can contain non-sequential data items in no particular order. A block of memory in the cache might be empty and contain no data at all. In order for hardware to check the validity of entries in the cache, every cache entry needs to maintain the following pieces of information:

1. A status bit to determine if the block is empty or full
2. The memory address of the data in the block
3. The data from the specified memory address

When the processor looks for data in the cache, it sends a memory address to the cache controller. the cache controller checks the address against all the address fields in the cache. If there is a hit, the cache controller returns the data. If there is a miss, the cache controller must pass the request to the next level of cache or to the main memory unit.



The memory address of the data in the cache is known as the **tag**.

Memory Stall Cycles

If the cache misses, the processor will need to stall the current instruction until the cache can fetch the correct data from a higher level. The amount of time lost by the stall is dependent on a number of factors. The number of memory accesses in a particular program is denoted as A_m; some of those accesses will hit the cache, and the rest will miss the cache. The rate of misses, equal to the probability that any particular access will miss, is denoted r_m. The average amount of time lost for each miss is known as the miss penalty, and is denoted as P_m. We can calculate the amount of time wasted because of cache miss stalls as:

$$\text{stall time} = A_m \times r_m \times P_m$$

Likewise, if we have the total number of instructions in a program, *N*, and the average number of misses per instruction, *MPI*, we can calculate the lost time as:

$$\text{stall time} = N \times MPI \times P_m$$

If instead of lost time we measure the miss penalty in the amount of lost cycles, the calculation will instead produce the number of cycles lost to memory stalls, instead of the amount of time lost to memory stalls.

Read Stall Times

To calculate the amount of time lost to cache read misses, we can perform the same basic calculations as above:

$$\text{read-stall time} = A_r \times r_r \times P_r$$

A_r is the average number of read accesses, r_r is the miss rate on reads, and P_r is the time or cycle penalty associated with a read miss.

Write Stall Times

Determining the amount of time lost to write stalls is similar, but an additional additive term that represents stalls in the write buffer needs to be included:

$$\text{write-stall time} = A_w \times r_w \times P_w + T_{wb}$$

Where T_{wb} is the amount of time lost because of stalls in the write buffer. The write buffer can stall when the cache attempts to synchronize with main memory.

Hierarchy Stall Times

In a hierarchical cache system, miss time penalties can be compounded when data is missed in multiple levels of cache. If data is missed in the L1 cache, it will be looked for in the L2 cache. However, if it also misses in the L2 cache, there will be a double-penalty. The L2 needs to load the data from the main memory (or the L3 cache, if the system has one), and then the data needs to be loaded into the L1. Notice that missing in two cache levels and then having to access main memory takes longer than if we had just accessed memory directly.

Design Considerations

L1 cache is typically designed with the intent of minimizing the time it takes to make a hit. If hit times are sufficiently fast, a sizable miss rate can be accepted. Misses in the L1 will be redirected to the L2 and that is still significantly faster than accesses to main memory. L1 cache tends to have smaller block sizes, but benefits from having more available blocks for the same amount of space. In order to make L1 hit times minimal, L1 are typically direct-mapped or even narrowly 2-way set associative.

L2 cache, on the other hand, needs to have a lower miss rate to help avoid accesses to main memory. Accesses to L2 cache are much faster than accesses to memory, so we should do everything possible to ensure that we maximize our hit rate. For this reason, L2 cache tends to be fully associative with large block sizes. This is because memory is typically read and written in sequential memory cells, so large block sizes can take advantage of that sequentiality.

L3 cache further continues this trend, with larger block sizes, and minimized miss rate.

Associativity

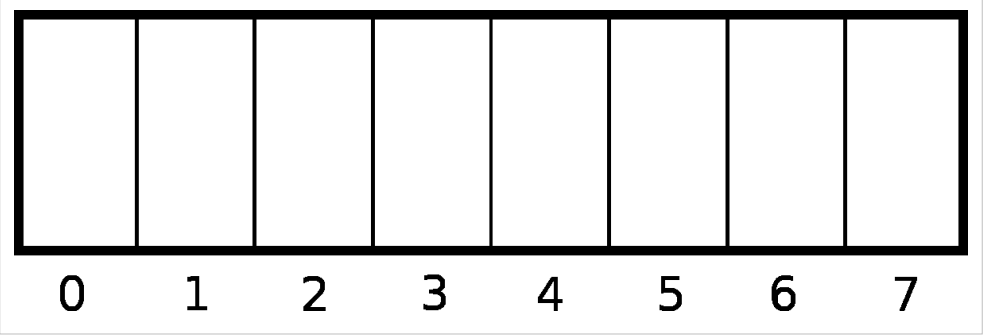
In order to increase the read speed in a cache, many cache designers implement some level of **associativity**. An associative cache creates a relationship between the original memory location and the location in the cache where that data is stored. The relationship between the address in main memory and the location where the data is stored is known as the **mapping** of the cache. In this way, if the data exists in the cache at all, the cache controller knows that it can only be in certain locations that satisfy the mapping.

Direct-Mapped

A direct-mapped system uses a hashing algorithm to assign an identifier to a memory address. A common hashing algorithm for this purpose is the modulo operation. The modulo operation divides the address by a certain number, *p*, and takes the remainder *r* as the result. If *a* is the main memory address, and *n* is an arbitrary non-negative integer, then the hashing algorithm must satisfy the following equation:

$$a = p \times n + r$$

If *p* is chosen properly by the designer, data will be evenly distributed throughout the cache.



In a direct-mapped system, each memory address corresponds to only a single cache location, but a single cache location can correspond to many memory locations. The image above shows a simple cache diagram with 8 blocks. All memory addresses therefore are calculated as *n mod 8*, where *n* is the memory address to read into the cache.

Memory addresses 0, 8, and 16 will all map to block 0 in the cache. Cache performance is worst when multiple data items with the same hash value are read, and performance is best when data items are close together in memory (such as a sequential block of program instructions, or a sequential array).

2-Way Set Associative

In a 2-way set associative cache system, the data value is hashed, but each hash value corresponds to a set of cache blocks. Each block contains multiple data cells, and a data value that is assigned to that block can be inserted anywhere in the block. The read speeds are quick because the cache controller can immediately narrow down its search area to the block that matches the address hash value.

2 way skewed associative

The 2-way skewed associative cache is "the best tradeoff for .... caches whose sizes are in the range 4K-8K bytes" --André SeznecAndré Seznec. "A Case for Two-Way Skewed-Associative Caches" [1]. Retrieved 2007-12-13.</ref>[2]

Fully Associative

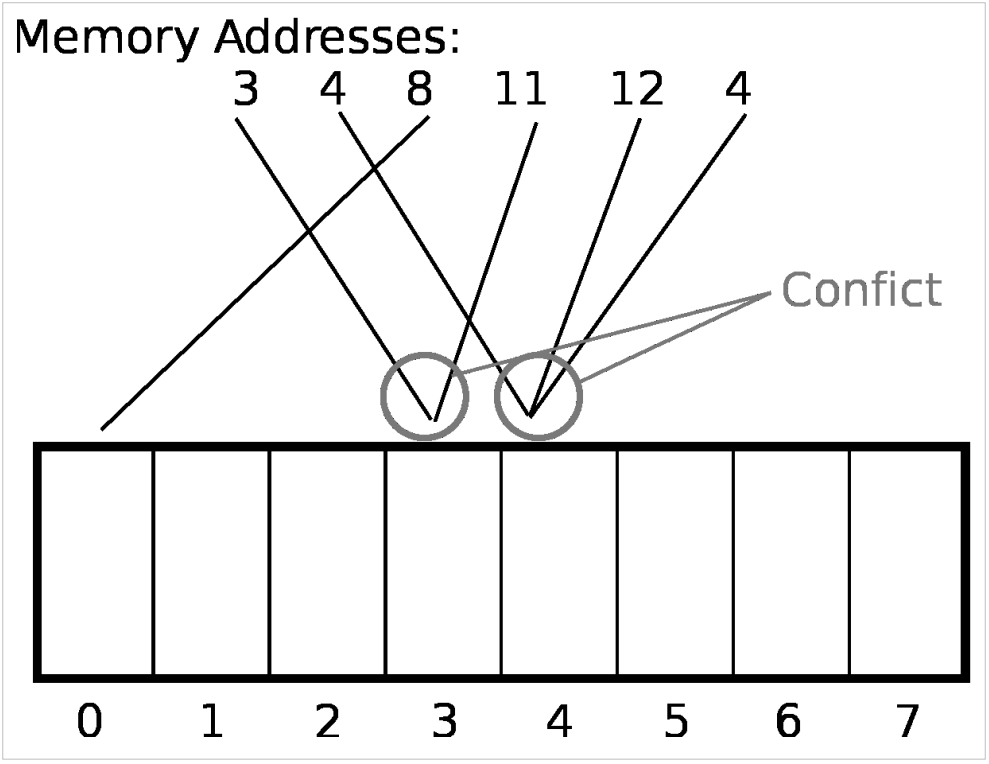
In a fully-associative cache, hash algorithms are not employed and data can be inserted anywhere in the cache that is available. A typical algorithm will write a new data value over the oldest unused data value in the cache. This scheme, however, requires the time an item is loaded or accessed to be stored, which can require lots of additional storage.

Cache Misses

There are three basic types of misses in a cache:

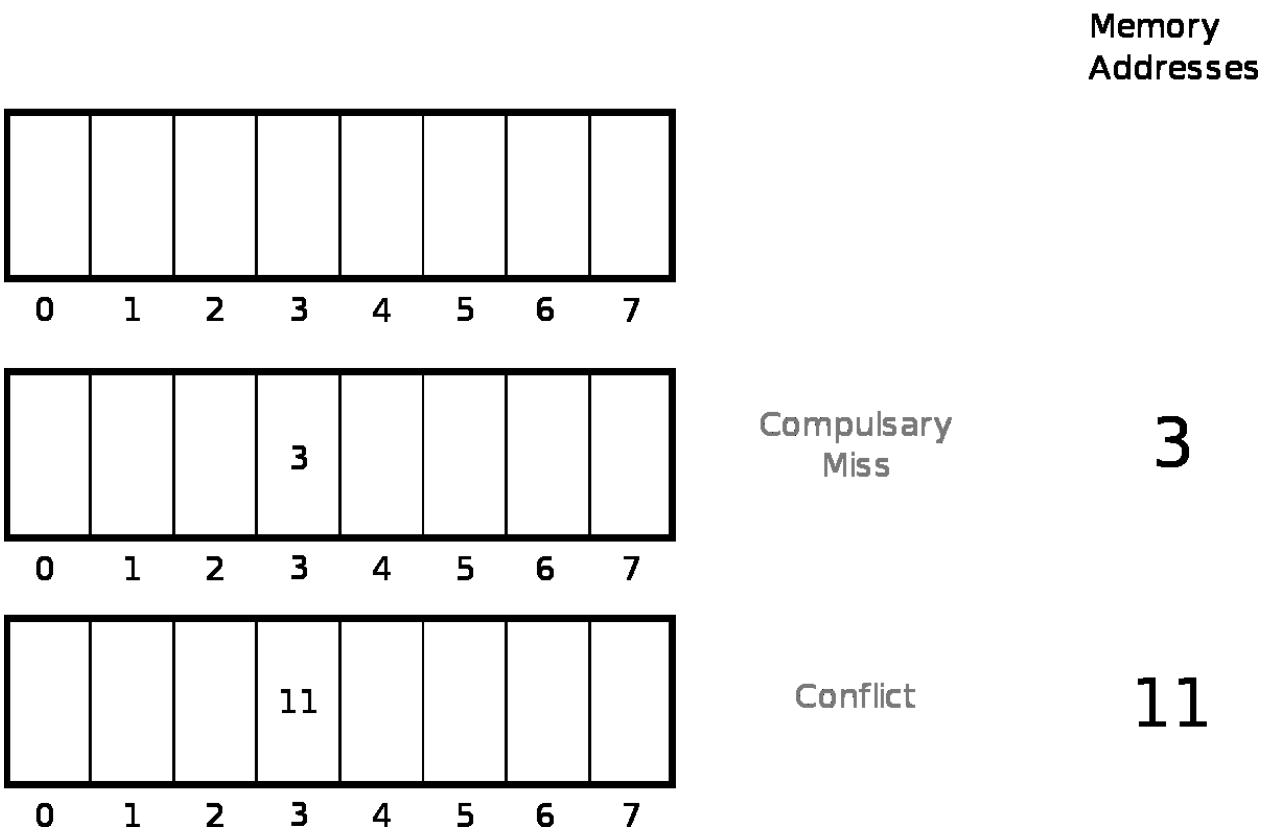
1. Conflict Misses
2. Compulsory Misses
3. Capacity Misses

Conflict Misses



A conflict miss occurs in a direct-mapped and 2-way set associative cache when two data items are mapped to the same cache locations. In a data miss, a recently used data item is overwritten with a new data item.

Compulsory Misses



The image above shows the difference between a conflict miss and a compulsory miss. A compulsory miss is an instance where the cache must miss because it does not contain any data. For instance, when a processor is first powered-on, there is no valid data in the cache and the first few reads will always miss.

The compulsory miss demonstrates the need for a cache to differentiate between a space that is empty and one that is full. Consider when we turn the processor on, and we reset all the address values to zero, an attempt to read a memory location with a hash value of zero will hit. We do not want the cache to hit if the blocks are empty.

Capacity Misses

Capacity misses occur when the cache block is not large enough to hold the data item.

Write Policy

Data writes require the same time delay as a data read. For this reason, caching systems typically will write data to the cache as well. However, when writing to the cache, it is important to ensure that the data is also written to the main memory, so it is not overwritten by the next cache read. If data in the cache is overwritten without being stored in main memory, the data will be lost.

It is imperative that caches write data to the main memory, but exactly when that data is written to the main memory is called the **write policy**. There are two write policies: **write through** and **write back**.

Cache Writing

Write operations take as long to perform as read operations in main memory. Many cached processors therefore will perform write operations on the cache as well as read operations. When data is written to memory, a write request is sent simultaneously to the main memory and to the cache. This way, the result data is available in the cache before it can be written (and then read again) from the main memory. When writing to the cache, it's important to make sure the main memory and the cache are synchronized and they contain the same data.

Write Through

In a write through system, data that is written to the cache is immediately written to the main memory as well. If many writes need to occur is sequential instructions, the write buffer may get backed up and cause a stall.

Write Back

In a write back system, the cache controller keeps track of which data items have been synchronized to main memory. The data items which have not been synchronized are called "dirty", and the cache controller prevents dirty data from being overwritten.

The cache controller will synchronize data during processor cycles where no other data is being written to the cache.

Stale Data

It is possible for the data in main memory to be changed by a component besides the microcontroller. For instance, many computer systems have memory-mapped I/O, or a DMA controller that can alter the data. It is important that the cache controller check that data in the cache is correct. Data in the cache that is old and may be incorrect is called "stale".

References

[1] http://citeseer.ist.psu.edu/seznec93case.html
[2] Micro-Architecture (http://www.irisa.fr/caps/PROJECTS/Architecture/) "Skewed-associative caches have ... major advantages over conventional set-associative caches."

Further reading

- simulators available for download at University of Maryland: Memory-Systems Research: "Computational Artifacts" (http://www.ece.umd.edu/~blj/memory/artifacts.html) can be used to measure cache performance and power dissipation for a microprocessor design without having to actually build it. This makes it much quicker and cheaper to explore various tradeoffs involved in cache design. ("Given a fixed size chip, if I sacrifice some L2 cache in order to make the L1 cache larger, will that make the overall performance better or worse?" "Is it better to use an extremely fast cycle time cache with low associativity, or a somewhat slower cycle time cache with high associativity giving a better hit rate?")

Microprocessor Design/Virtual Memory

Microprocessor Design

Virtual Memory is a computer concept where the main memory is broken up into a series of individual *pages*. Those pages can be moved in memory as a unit, or they can even be moved to secondary storage to make room in main memory for new data. In essence, virtual memory allows a computer to use more RAM then it has available.

Implementation

Virtual memory can be implemented both in hardware and (to a lesser degree) in software, although many modern implementations have both hard and soft components. We discuss virtual memory here because many modern PC and server processors have virtual memory capabilities built in.

Pageing systems are designed to be transparent, that is the the programs running on the microprocessor do not need to be explicitly aware of the paging mechanism to operate correctly.

Many processor systems give pages certain qualifiers to specify what kinds of data can be stored in the page. For instance, many new processors specify whether a page contains instructions or data. Data pages may not be executed as instructions.

Memory Accessing

Memory addresses correspond to a particular page, and an offset within that page. If a page is 2^{12} bytes in a 32-bit computer, then the first 22 bits of the memory address are the page address, and the lower 12 bits are the offset of the data inside that page. The top 22 bits in this case will be separated from the address, and they will be replaced with the current physical address of that page. If the page does not exist in main memory, the processor (or the paging software) will retrieve the page from secondary storage, which can cause a significant delay.

Pages

A page is a basic unit of memory, typically several kilobytes or larger. A page may be moved in memory to different locations, or if it is not being used frequently it can be moved to secondary storage instead. The area in the secondary storage is typically known as the **page file**, or as a "scratchpad" or something similar.

Page Table

The addresses of the various pages are stored in a paging table. The paging table itself can be stored in a memory unit inside the processor, or it can reside in a dedicated area of main memory.

Page Faults

A page fault occurs when the processor cannot find a page in the page table.

Translation Look-Aside Buffer

The translation look-aside buffer (TLB) is a small structure, similar to a cache, that stores the addresses of the most recently used pages. Looking up a page in the TLB is much faster then searching for the page in the page table. When the processor cannot find a particular page in the TLB, it is known as a "TLB Miss". When the TLB misses, the processor looks for the page in the page table. If the page is not in the table either, there is a page fault.

Notice that even though the TLB is similar to the cache mechanism, the two are not related.

Microprocessor Design/Power Dissipation

Microprocessor Design

In addition to power and performance, another useful metric for examining processors is in terms of the amount of power used. Power is a valuable commodity, especially in mobile or embedded environments, and in server farms. Processors that utilize less power are more highly prized in these areas then processors with more capability and better performance.

Reducing the amount of energy used, without reducing the performance of the computer system, is one of the Grand Challenges in computer science.[1]

Gene's Law

Less well known then Moore's law is Gene's Law, named after Gene Frantz. According to Gene's law, the power dissipation in embedded DSP processors will decrease by half every 18 months.

Two reasons to reduce power

The power used by a microprocessor causes 2 problems. Some techniques reduce only peak power; some other techniques reduce only average power.

The peak power problem

- All the power used by a microprocessor is eventually converted to heat energy. If too much heat energy is allowed to build up inside the microprocessor, the temperature will rise high enough to destroy the microprocessor.

This problem is solved by the cooling system, which replaces that problem with another problem:

- The higher the peak power used by a microprocessor, the more expensive the up-front cost of the cooling system necessary to keep that processor from destroying itself.

The average power problem

- The higher the average power used by a microprocessor, the higher the cost to the person who uses that microprocessor. That person must not only pay for the electric power going into the microprocessor, but also pay for cooling to pump waste heat energy all the way from the microprocessor to the outside environment.

In some situations, there are other reasons to reduce power:

- Laptop designers want a small, lightweight laptop. The higher the average power used by a microprocessor, the heavier the battery must be for a given runtime.

Heat

In microprocessors, power is mostly dissipated as heat energy. This conversion to heat energy is a function of the size of the wires and transistors, and the operating frequency of the processor.

As transistors get smaller, the depletion region gets smaller, and current leaks through the transistor even when it is off. This leakage produces additional heat, and wastes additional power.

Heat can also cause materials to expand, which can alter the electrical characteristics of the tiny transistors and wires.

Many small microcontrollers don't need to worry about heat because they generate so little, but larger modern general purpose processors typically need to be accompanied by heat sinks and fans to help cool the processor. If a processor is running too hot, typically it can be slowed down to a lower clock rate to help prevent heat build up.

As power is a function of the square of the voltage, approximately, if you can reduce the power supply voltage by half, you can reduce the power dissipation by possibly three quarters. Because of this, microprocessor chips are quite often designed to run at what were once considered impossibly low voltages. The initial microprocessor chips, the Intel 8080 and the Motorola MC6800, were designed to run at 5.0 volts. More modern microprocessors, like the AMD Athlon K7 chips, are designed to run at 1.65 volts or even lower.

It should be noted that, in order to prevent uncontrollable heat buildup, many modern general-purpose microprocessors dynamically turn off parts of the chip. A computer that is being used for purely integer calculations does not need its floating point unit, and so power to the entire FPU, except possibly the register stack, is turned off. Major sections of the microprocessor, then, can be turned on and off several times per millisecond. While this does cut down average power draw and heat dissipation, it does put extraordinary demands on the power supply for the chip, which can see power requirements that jump 50% in microseconds.

further reading

- How To Assemble A Desktop PC/Silencing describes some of the problems caused cooling fans, which wouldn't be necessary if CPUs generated less heat.

[1] "Revitalizing Computer Architecture Research: Grand Research Challenges in Computer Science and Engineering" (http://www.cra.org/Activities/grand.challenges/architecture/computer.architecture.pdf) edited by Mary Jane Irwin, John Paul Shen, 2005.

Resources

- Frantz, G., "Digital signal processor trends", IEEE Micro, Vol.20, Iss.6, Nov/Dec 2000, Pages:52-59

Article Sources and Contributors

Wikibooks:Collections Preface *Source*: http://en.wikibooks.org/w/index.php?oldid=1795903 *Contributors*: Adrignola, Jomegat, Magesha, Mike.lifeguard, RobinH, Whiteknight

Microprocessor Design/Introduction *Source*: http://en.wikibooks.org/w/index.php?oldid=1632285 *Contributors*: Darklama, DavidCary, Jomegat, Spongebob88, Whiteknight, 1 anonymous edits

Microprocessor Design/Microprocessors *Source*: http://en.wikibooks.org/w/index.php?oldid=2167461 *Contributors*: Ervinn, Hagindaz, Jomegat, M sakarya, Panic2k4, Spongebob88, Whiteknight, 25 anonymous edits

Microprocessor Design/Computer Architecture *Source*: http://en.wikibooks.org/w/index.php?oldid=2146105 *Contributors*: Adrignola, DavidCary, Ervinn, Jomegat, M sakarya, Panic2k4, Recent Runes, Remi, Spongebob88, Wabernat, Whiteknight, 18 anonymous edits

Microprocessor Design/Instruction Set Architectures *Source*: http://en.wikibooks.org/w/index.php?oldid=2162637 *Contributors*: Cburnett, Chazz, DavidCary, Ervinn, Mohammadthalif, Whiteknight, 6 anonymous edits

Microprocessor Design/Memory *Source*: http://en.wikibooks.org/w/index.php?oldid=1720811 *Contributors*: DavidCary, Whiteknight, 7 anonymous edits

Microprocessor Design/Control and Datapath *Source*: http://en.wikibooks.org/w/index.php?oldid=1906678 *Contributors*: DavidCary, Whiteknight

Microprocessor Design/Performance *Source*: http://en.wikibooks.org/w/index.php?oldid=1693874 *Contributors*: DavidCary, Jomegat, M sakarya, Whiteknight, 3 anonymous edits

Microprocessor Design/Assembly Language *Source*: http://en.wikibooks.org/w/index.php?oldid=1502037 *Contributors*: DavidCary, Ervinn, Jomegat, Whiteknight, YMS, 6 anonymous edits

Microprocessor Design/Design Steps *Source*: http://en.wikibooks.org/w/index.php?oldid=2155521 *Contributors*: Avicennasis, DavidCary, Hoo man, Whiteknight, 4 anonymous edits

Microprocessor Design/Basic Components *Source*: http://en.wikibooks.org/w/index.php?oldid=1958943 *Contributors*: Adrignola, Whiteknight, 3 anonymous edits

Microprocessor Design/Program Counter *Source*: http://en.wikibooks.org/w/index.php?oldid=1688803 *Contributors*: Chazz, QuiteUnusual, Whiteknight, 5 anonymous edits

Microprocessor Design/Instruction Decoder *Source*: http://en.wikibooks.org/w/index.php?oldid=1263994 *Contributors*: Whiteknight, 3 anonymous edits

Microprocessor Design/Register File *Source*: http://en.wikibooks.org/w/index.php?oldid=1831565 *Contributors*: Adrignola, Spongebob88, Thenub314, Whiteknight, 2 anonymous edits

Microprocessor Design/Memory Unit *Source*: http://en.wikibooks.org/w/index.php?oldid=2164638 *Contributors*: DavidCary, Whiteknight, 1 anonymous edits

Microprocessor Design/ALU *Source*: http://en.wikibooks.org/w/index.php?oldid=2155400 *Contributors*: DavidCary, Jomegat, NipplesMeCool, Whiteknight, 4 anonymous edits

Microprocessor Design/FPU *Source*: http://en.wikibooks.org/w/index.php?oldid=1490965 *Contributors*: Chazz, Whiteknight, 3 anonymous edits

Microprocessor Design/Control Unit *Source*: http://en.wikibooks.org/w/index.php?oldid=1966505 *Contributors*: Whiteknight, Xania, 4 anonymous edits

Microprocessor Design/Add and Subtract Blocks *Source*: http://en.wikibooks.org/w/index.php?oldid=2163595 *Contributors*: DavidCary, MichaelFrey, Whiteknight, 3 anonymous edits

Microprocessor Design/Shift and Rotate Blocks *Source*: http://en.wikibooks.org/w/index.php?oldid=2120805 *Contributors*: Ixeelectronics, QuiteUnusual, Whiteknight, 2 anonymous edits

Microprocessor Design/Multiply and Divide Blocks *Source*: http://en.wikibooks.org/w/index.php?oldid=1345724 *Contributors*: Mike.lifeguard, Whiteknight, 1 anonymous edits

Microprocessor Design/ALU Flags *Source*: http://en.wikibooks.org/w/index.php?oldid=2000344 *Contributors*: DavidCary, Whiteknight, 8 anonymous edits

Microprocessor Design/Single Cycle Processors *Source*: http://en.wikibooks.org/w/index.php?oldid=1239993 *Contributors*: Spongebob88, Whiteknight

Microprocessor Design/Multi Cycle Processors *Source*: http://en.wikibooks.org/w/index.php?oldid=2163597 *Contributors*: Whiteknight, 2 anonymous edits

Microprocessor Design/Pipelined Processors *Source*: http://en.wikibooks.org/w/index.php?oldid=1414652 *Contributors*: Spongebob88, Whiteknight, 1 anonymous edits

Microprocessor Design/Superscalar Processors *Source*: http://en.wikibooks.org/w/index.php?oldid=818267 *Contributors*: Whiteknight

Microprocessor Design/VLIW Processors *Source*: http://en.wikibooks.org/w/index.php?oldid=818196 *Contributors*: Whiteknight

Microprocessor Design/Vector Processors *Source*: http://en.wikibooks.org/w/index.php?oldid=818260 *Contributors*: Whiteknight

Microprocessor Design/Multicore Processors *Source*: http://en.wikibooks.org/w/index.php?oldid=1397954 *Contributors*: DavidCary, Whiteknight

Microprocessor Design/Exceptions *Source*: http://en.wikibooks.org/w/index.php?oldid=848692 *Contributors*: Whiteknight

Microprocessor Design/Interrupts *Source*: http://en.wikibooks.org/w/index.php?oldid=1635929 *Contributors*: DavidCary, Whiteknight, 2 anonymous edits

Microprocessor Design/Hazards *Source*: http://en.wikibooks.org/w/index.php?oldid=2169282 *Contributors*: Rmac, Whiteknight, 1 anonymous edits

Microprocessor Design/Performance Metrics *Source*: http://en.wikibooks.org/w/index.php?oldid=1363361 *Contributors*: DavidCary, Whiteknight, 1 anonymous edits

Microprocessor Design/Benchmarking *Source*: http://en.wikibooks.org/w/index.php?oldid=1584757 *Contributors*: DavidCary, Whiteknight

Microprocessor Design/Optimizations *Source*: http://en.wikibooks.org/w/index.php?oldid=2175017 *Contributors*: Whiteknight

Microprocessor Design/Multi-Core Systems *Source*: http://en.wikibooks.org/w/index.php?oldid=1397957 *Contributors*: DavidCary, Whiteknight

Microprocessor Design/Memory-Level Parallelism *Source*: http://en.wikibooks.org/w/index.php?oldid=1215387 *Contributors*: Red4tribe, Whiteknight, 1 anonymous edits

Microprocessor Design/Out Of Order Execution *Source*: http://en.wikibooks.org/w/index.php?oldid=1453032 *Contributors*: DavidCary, Whiteknight, 1 anonymous edits

Microprocessor Design/Assembler *Source*: http://en.wikibooks.org/w/index.php?oldid=1490275 *Contributors*: Chazz, DavidCary, NipplesMeCool, Whiteknight, 2 anonymous edits

Microprocessor Design/Simulator *Source*: http://en.wikibooks.org/w/index.php?oldid=1264076 *Contributors*: Whiteknight, 2 anonymous edits

Microprocessor Design/Compiler *Source*: http://en.wikibooks.org/w/index.php?oldid=1555649 *Contributors*: DavidCary, Rmac, Whiteknight

Microprocessor Design/FPGA *Source*: http://en.wikibooks.org/w/index.php?oldid=1721382 *Contributors*: DavidCary, Whiteknight, 2 anonymous edits

Microprocessor Design/Wire Wrap *Source*: http://en.wikibooks.org/w/index.php?oldid=2163304 *Contributors*: DavidCary, Whiteknight, 1 anonymous edits

Microprocessor Design/Photolithography *Source*: http://en.wikibooks.org/w/index.php?oldid=1121240 *Contributors*: DavidCary, Whiteknight

Microprocessor Design/Sockets and interfacing *Source*: http://en.wikibooks.org/w/index.php?oldid=1455496 *Contributors*: CommonsDelinker, Whiteknight, 3 anonymous edits

Microprocessor Design/Microcodes *Source*: http://en.wikibooks.org/w/index.php?oldid=1158439 *Contributors*: DavidCary, Whiteknight, 1 anonymous edits

Microprocessor Design/Cache *Source*: http://en.wikibooks.org/w/index.php?oldid=2028802 *Contributors*: DavidCary, Immibis, NipplesMeCool, Whiteknight, 3 anonymous edits

Microprocessor Design/Virtual Memory *Source*: http://en.wikibooks.org/w/index.php?oldid=850028 *Contributors*: Whiteknight

Microprocessor Design/Power Dissipation *Source*: http://en.wikibooks.org/w/index.php?oldid=1720805 *Contributors*: Chazz, DavidCary, Whiteknight, 2 anonymous edits

Image Sources, Licenses and Contributors

Image:Wikibooks-logo-en-noslogan.svg *Source*: http://en.wikibooks.org/w/index.php?title=File:Wikibooks-logo-en-noslogan.svg *License*: logo *Contributors*: User:Bastique, User:Ramac et al.

Image:Personal computer, exploded 4.svg *Source*: http://en.wikibooks.org/w/index.php?title=File:Personal_computer,_exploded_4.svg *License*: Creative Commons Attribution-ShareAlike 3.0 Unported *Contributors*: Aleator, BMK, Berrucomons, Boivie, Edward, Gustavb, Huhsunqu, J.delanoy, Jon Harald Søby, Kozuch, Lysander89, Mdd, Mhare, Monsterxxl, Origamiemensch, Rocket000, Slovik, Ss181292, UED77, 23 anonymous edits

Image:Moore Law diagram (2004).png *Source*: http://en.wikibooks.org/w/index.php?title=File:Moore_Law_diagram_(2004).png *License*: unknown *Contributors*: AtonX, Helix84, Julben

File:Mips32 addi.svg *Source*: http://en.wikibooks.org/w/index.php?title=File:Mips32_addi.svg *License*: Creative Commons Attribution-ShareAlike 3.0 Unported *Contributors*: en:User:Booyabazooka

Image:6t-SRAM-cell.png *Source*: http://en.wikibooks.org/w/index.php?title=File:6t-SRAM-cell.png *License*: Creative Commons Attribution-ShareAlike 3.0 Unported *Contributors*: Abelsson

Image:Assembler.png *Source*: http://en.wikibooks.org/w/index.php?title=File:Assembler.png *License*: Creative Commons Attribution-ShareAlike 3.0 Unported *Contributors*: User:Moonshadow

Image:Multiplexer 2-to-1.svg *Source*: http://en.wikibooks.org/w/index.php?title=File:Multiplexer_2-to-1.svg *License*: Creative Commons Attribution-ShareAlike 3.0 Unported *Contributors*: en:User:Cburnett

Image:Multiplexer 4-to-1.svg *Source*: http://en.wikibooks.org/w/index.php?title=File:Multiplexer_4-to-1.svg *License*: Creative Commons Attribution-ShareAlike 3.0 Unported *Contributors*: en:User:Cburnett

Image:Multiplexer 8-to-1.svg *Source*: http://en.wikibooks.org/w/index.php?title=File:Multiplexer_8-to-1.svg *License*: Creative Commons Attribution-ShareAlike 3.0 Unported *Contributors*: en:User:Cburnett

Image:Multiplexer 16-to-1.svg *Source*: http://en.wikibooks.org/w/index.php?title=File:Multiplexer_16-to-1.svg *License*: Creative Commons Attribution-ShareAlike 3.0 Unported *Contributors*: en:User:Cburnett

Image:PC Simple.svg *Source*: http://en.wikibooks.org/w/index.php?title=File:PC_Simple.svg *License*: GNU Free Documentation License *Contributors*: Whiteknight

Image:PC Branch.svg *Source*: http://en.wikibooks.org/w/index.php?title=File:PC_Branch.svg *License*: GNU Free Documentation License *Contributors*: Whiteknight

Image:PC Offset Branch.svg *Source*: http://en.wikibooks.org/w/index.php?title=File:PC_Offset_Branch.svg *License*: GNU Free Documentation License *Contributors*: Whiteknight

Image:PC Offset Branch 2.svg *Source*: http://en.wikibooks.org/w/index.php?title=File:PC_Offset_Branch_2.svg *License*: GNU Free Documentation License *Contributors*: Whiteknight

Image:PC Branch Jump.svg *Source*: http://en.wikibooks.org/w/index.php?title=File:PC_Branch_Jump.svg *License*: GNU Free Documentation License *Contributors*: Whiteknight

Image:Register File Simple.svg *Source*: http://en.wikibooks.org/w/index.php?title=File:Register_File_Simple.svg *License*: GNU Free Documentation License *Contributors*: Whiteknight

Image:Register File Medium.svg *Source*: http://en.wikibooks.org/w/index.php?title=File:Register_File_Medium.svg *License*: GNU Free Documentation License *Contributors*: Whiteknight

Image:Register File Large.svg *Source*: http://en.wikibooks.org/w/index.php?title=File:Register_File_Large.svg *License*: GNU Free Documentation License *Contributors*: Whiteknight

Image:Register Bank.svg *Source*: http://en.wikibooks.org/w/index.php?title=File:Register_Bank.svg *License*: GNU Free Documentation License *Contributors*: Whiteknight

Image:Register Bank Address.svg *Source*: http://en.wikibooks.org/w/index.php?title=File:Register_Bank_Address.svg *License*: GNU Free Documentation License *Contributors*: Whiteknight

Image:Memory Unit.svg *Source*: http://en.wikibooks.org/w/index.php?title=File:Memory_Unit.svg *License*: GNU Free Documentation License *Contributors*: Whiteknight

Image:ALU symbol.svg *Source*: http://en.wikibooks.org/w/index.php?title=File:ALU_symbol.svg *License*: Creative Commons Attribution-ShareAlike 3.0 Unported *Contributors*: en:User:Cburnett

Image:2-bit ALU.png *Source*: http://en.wikibooks.org/w/index.php?title=File:2-bit_ALU.png *License*: GNU Free Documentation License *Contributors*: en:User:Cburnett

Image:74181aluschematic.png *Source*: http://en.wikibooks.org/w/index.php?title=File:74181aluschematic.png *License*: GNU Free Documentation License *Contributors*: User:Poil

Image:Isaccumulator.png *Source*: http://en.wikibooks.org/w/index.php?title=File:Isaccumulator.png *License*: GNU Free Documentation License *Contributors*: User:Poil

Image:Isreg2reg.png *Source*: http://en.wikibooks.org/w/index.php?title=File:Isreg2reg.png *License*: GNU Free Documentation License *Contributors*: User:Poil

Image:Is0addr.png *Source*: http://en.wikibooks.org/w/index.php?title=File:Is0addr.png *License*: GNU Free Documentation License *Contributors*: User:Poil

Image:Isregmem.png *Source*: http://en.wikibooks.org/w/index.php?title=File:Isregmem.png *License*: GNU Free Documentation License *Contributors*: User:Poil

Image:General floating point.svg *Source*: http://en.wikibooks.org/w/index.php?title=File:General_floating_point.svg *License*: GNU Free Documentation License *Contributors*: User:Stannered

Image:4-bit ripple carry adder-subtracter.svg *Source*: http://en.wikibooks.org/w/index.php?title=File:4-bit_ripple_carry_adder-subtracter.svg *License*: GNU Free Documentation License *Contributors*: en:User:Cburnett

Image:Half-adder.svg *Source*: http://en.wikibooks.org/w/index.php?title=File:Half-adder.svg *License*: Creative Commons Attribution-ShareAlike 3.0 Unported *Contributors*: en:User:Cburnett

Image:Full-adder.svg *Source*: http://en.wikibooks.org/w/index.php?title=File:Full-adder.svg *License*: GNU Free Documentation License *Contributors*: en:User:Cburnett

Image:1-bit full-adder.svg *Source*: http://en.wikibooks.org/w/index.php?title=File:1-bit_full-adder.svg *License*: Creative Commons Attribution-ShareAlike 3.0 Unported *Contributors*: en:User:Cburnett

Image:Full-adder with gate delay.svg *Source*: http://en.wikibooks.org/w/index.php?title=File:Full-adder_with_gate_delay.svg *License*: GNU Free Documentation License *Contributors*: en:User:Cburnett

Image:Serialadder.png *Source*: http://en.wikibooks.org/w/index.php?title=File:Serialadder.png *License*: GNU Free Documentation License *Contributors*: User:Poil

Image:4-bit ripple carry adder.svg *Source*: http://en.wikibooks.org/w/index.php?title=File:4-bit_ripple_carry_adder.svg *License*: GNU Free Documentation License *Contributors*: en:User:Cburnett

Image:4-bit carry lookahead adder.svg *Source*: http://en.wikibooks.org/w/index.php?title=File:4-bit_carry_lookahead_adder.svg *License*: GNU Free Documentation License *Contributors*: en:User:Cburnett

Image:16-bit lookahead carry unit.svg *Source*: http://en.wikibooks.org/w/index.php?title=File:16-bit_lookahead_carry_unit.svg *License*: Creative Commons Attribution-ShareAlike 3.0 Unported *Contributors*: en:User:Cburnett

Image:64-bit lookahead carry unit.svg *Source*: http://en.wikibooks.org/w/index.php?title=File:64-bit_lookahead_carry_unit.svg *License*: GNU Free Documentation License *Contributors*: en:User:Cburnett

Image:Cla4bitsPG.png *Source*: http://en.wikibooks.org/w/index.php?title=File:Cla4bitsPG.png *License*: GNU Free Documentation License *Contributors*: User:Poil

Image:Cla16bitsPG.png *Source*: http://en.wikibooks.org/w/index.php?title=File:Cla16bitsPG.png *License*: GNU Free Documentation License *Contributors*: User:Poil

Image:Rotate left logically.svg *Source*: http://en.wikibooks.org/w/index.php?title=File:Rotate_left_logically.svg *License*: Creative Commons Attribution-ShareAlike 3.0 Unported *Contributors*: en:User:Cburnett

Image:Rotate right logically.svg *Source*: http://en.wikibooks.org/w/index.php?title=File:Rotate_right_logically.svg *License*: Creative Commons Attribution-ShareAlike 3.0 Unported *Contributors*: en:User:Cburnett

Image:Rotate right arithmetically.svg *Source*: http://en.wikibooks.org/w/index.php?title=File:Rotate_right_arithmetically.svg *License*: Creative Commons Attribution-ShareAlike 3.0 Unported *Contributors*: en:User:Cburnett

Image:Rotate left.svg *Source*: http://en.wikibooks.org/w/index.php?title=File:Rotate_left.svg *License*: Creative Commons Attribution-ShareAlike 3.0 Unported *Contributors*: en:User:Cburnett

Image:Rotate right.svg *Source*: http://en.wikibooks.org/w/index.php?title=File:Rotate_right.svg *License*: Creative Commons Attribution-ShareAlike 3.0 Unported *Contributors*: en:User:Cburnett

Image:Nopipeline.png *Source*: http://en.wikibooks.org/w/index.php?title=File:Nopipeline.png *License*: GNU Free Documentation License *Contributors*: User:Poil

Image:Fivestagespipeline.png *Source*: http://en.wikibooks.org/w/index.php?title=File:Fivestagespipeline.png *License*: GNU Free Documentation License *Contributors*: User:Poil

Image:Pipeline-base.png *Source*: http://en.wikibooks.org/w/index.php?title=File:Pipeline-base.png *License*: Public Domain *Contributors*: Hellisp

Image:Pipeline MIPS.png *Source*: http://en.wikibooks.org/w/index.php?title=File:Pipeline_MIPS.png *License*: Public Domain *Contributors*: Hellisp

Image:Pipeline 3.png *Source*: http://en.wikibooks.org/w/index.php?title=File:Pipeline_3.png *License*: Public Domain *Contributors*: Hellisp

Image:Pentium4superpipeline.png *Source*: http://en.wikibooks.org/w/index.php?title=File:Pentium4superpipeline.png *License*: GNU Free Documentation License *Contributors*: User:Poil

Image:Superscalarpipeline.png *Source*: http://en.wikibooks.org/w/index.php?title=File:Superscalarpipeline.png *License*: GNU Free Documentation License *Contributors*: User:Poil

Image:Vliwpipeline.png *Source*: http://en.wikibooks.org/w/index.php?title=File:Vliwpipeline.png *License*: GNU Free Documentation License *Contributors*: User:Poil

Image:Vectorsimdpipeline.png *Source*: http://en.wikibooks.org/w/index.php?title=File:Vectorsimdpipeline.png *License*: GNU Free Documentation License *Contributors*: User:Poil

Image:IntelCore2DuoE6600.jpg *Source*: http://en.wikibooks.org/w/index.php?title=File:IntelCore2DuoE6600.jpg *License*: Creative Commons Attribution-ShareAlike 3.0 Unported *Contributors*: Jürgen Melzer

Image:SPE (cell).png *Source*: http://en.wikibooks.org/w/index.php?title=File:SPE_(cell).png *License*: Public Domain *Contributors*: Hellisp

Image:PPE (Cell).png *Source*: http://en.wikibooks.org/w/index.php?title=File:PPE_(Cell).png *License*: Public Domain *Contributors*: Hellisp

Image:IBM Cell Block Diagram.svg *Source*: http://en.wikibooks.org/w/index.php?title=File:IBM_Cell_Block_Diagram.svg *License*: GNU Free Documentation License *Contributors*: Whiteknight

Image:Interrupt.svg *Source*: http://en.wikibooks.org/w/index.php?title=File:Interrupt.svg *License*: GNU Free Documentation License *Contributors*: Whiteknight

Image:Pipeline 5.png *Source*: http://en.wikibooks.org/w/index.php?title=File:Pipeline_5.png *License*: Public Domain *Contributors*: Hellisp

Image:BranchPredictor.JPG *Source*: http://en.wikibooks.org/w/index.php?title=File:BranchPredictor.JPG *License*: Public Domain *Contributors*: Original uploader was Whiteknight at en.wikibooks

Image:Sharp LH0080A.jpg *Source*: http://en.wikibooks.org/w/index.php?title=File:Sharp_LH0080A.jpg *License*: GNU Free Documentation License *Contributors*: User Baz1521 on ja.wikipedia

Image:AMD Athlon 1.1Ghz pins.jpg *Source*: http://en.wikibooks.org/w/index.php?title=File:AMD_Athlon_1.1Ghz_pins.jpg *License*: Creative Commons Attribution 3.0 *Contributors*: Krdan

Image:Socket 479.jpg *Source*: http://en.wikibooks.org/w/index.php?title=File:Socket_479.jpg *License*: Creative Commons Attribution-Sharealike 2.5 *Contributors*: User:Rosco

Image:Intel-pentium-pro-CPU.jpg *Source*: http://en.wikibooks.org/w/index.php?title=File:Intel-pentium-pro-CPU.jpg *License*: Creative Commons Attribution-ShareAlike 3.0 Unported *Contributors*: Adamantios

Image:Cache,basic.svg *Source*: http://en.wikibooks.org/w/index.php?title=File:Cache,basic.svg *License*: GNU Free Documentation License *Contributors*: Traced by User:Stannered

Image:Cache Block Basic.svg *Source*: http://en.wikibooks.org/w/index.php?title=File:Cache_Block_Basic.svg *License*: GNU Free Documentation License *Contributors*: Whiteknight

Image:Cache Block Basic Conflict.svg *Source*: http://en.wikibooks.org/w/index.php?title=File:Cache_Block_Basic_Conflict.svg *License*: GNU Free Documentation License *Contributors*: Whiteknight

Image:Cache Block Conflict Compulsary.svg *Source*: http://en.wikibooks.org/w/index.php?title=File:Cache_Block_Conflict_Compulsary.svg *License*: GNU Free Documentation License *Contributors*: Whiteknight

License