Microprocessor Design

en.wikibooks.org

March 15, 2015

Contents

1 Introduction **3**
- 1.1 About This Book . . . 3
- 1.2 How Will This Book Be Organized? . . . 3
- 1.3 Prerequisites . . . 4
- 1.4 Who Is This Book For? . . . 4
- 1.5 What This Book Will Not Cover . . . 4
- 1.6 Terminology . . . 5

2 Microprocessors **7**
- 2.1 Microprocessors . . . 7
- 2.2 Abstraction Layers . . . 11
- 2.3 Operating System . . . 12
- 2.4 ISA . . . 12
- 2.5 Moore's Law . . . 12
- 2.6 Clock Rates . . . 14
- 2.7 Basic Elements of a Computer . . . 15

3 Computer Architecture **17**
- 3.1 Von Neumann Architecture . . . 17
- 3.2 Harvard Architecture . . . 18
- 3.3 Modern Computers . . . 18
- 3.4 RISC and CISC and DSP . . . 18
- 3.5 Microprocessor Components . . . 19
- 3.6 Endian . . . 22
- 3.7 Stack . . . 23
- 3.8 further reading . . . 24

4 Instruction Set Architectures **25**
- 4.1 ISAs . . . 25
- 4.2 Memory Arrangement . . . 26
- 4.3 Common Instructions . . . 26
- 4.4 Instruction Length . . . 28
- 4.5 Further reading . . . 29

5 Memory **31**
- 5.1 Memory Hierarchy . . . 31
- 5.2 Hard Disk Drives . . . 31
- 5.3 RAM . . . 31
- 5.4 Cache . . . 33
- 5.5 Registers . . . 33

6 Control and Datapath **35**
6.1 References 36

7 Performance **37**
7.1 Clock Cycles 37
7.2 Cycles per Instruction 37
7.3 Instruction count 38
7.4 CPU Time 38
7.5 Performance 38
7.6 Amdahls Law 38
7.7 Benchmarking 39

8 Assembly Language **41**
8.1 Assemblers 41
8.2 Assembly Language Constructs 41
8.3 Load and Store 42
8.4 Arithmetic 42
8.5 Jumping 42
8.6 Branching 42
8.7 Further reading 43

9 Design Steps **45**
9.1 Determine Machine Capabilities 45
9.2 Design the Datapath 46
9.3 Create ISA 46
9.4 Instruction Set Design 47
9.5 Build Control Logic 48
9.6 Design the Address Path 48
9.7 Verify the design 48
9.8 Further reading 49
9.9 References 49

10 Basic Components **51**
10.1 Basic Components 51
10.2 Registers 51
10.3 Multiplexers 52
10.4 Adder 53

11 Program Counter **55**
11.1 Updating the PC 55
11.2 Branching 57

12 Instruction Decoder **61**
12.1 RISC Instruction Decoder 61
12.2 CISC Instruction Decoder 61

13 Register File **63**
13.1 Register File 64
13.2 More registers than you can shake a stick at 66

13.3 Register Bank . . . 67
13.4 References . . . 69

14 Memory Unit **71**
14.1 Memory Unit . . . 71
14.2 Actions of the Memory Unit . . . 71
14.3 Timing Issues . . . 71

15 ALU **73**
15.1 Tasks of an ALU . . . 74
15.2 ALU Slice . . . 74
15.3 Example: 2-Bit ALU . . . 74
15.4 Example: 4-Bit ALU . . . 75
15.5 Additional Operations . . . 76
15.6 ALU Configurations . . . 76
15.7 References . . . 83

16 FPU **85**
16.1 Floating point numbers . . . 85
16.2 Floating Point Unit Design . . . 86
16.3 Further Reading . . . 87

17 Control Unit **89**
17.1 Simple Control Unit . . . 89
17.2 Complex Control Unit . . . 89

18 Add and Subtract Blocks **91**
18.1 Addition and Subtraction . . . 91
18.2 Bit Adders . . . 91
18.3 Serial Adder . . . 95
18.4 Parallel Adder . . . 95
18.5 Sources . . . 101

19 Shift and Rotate Blocks **103**
19.1 Shift and Rotate . . . 103
19.2 Logical Shift . . . 103
19.3 Arithmetic shift . . . 103
19.4 Rotations . . . 104
19.5 Fast Shift Implementations . . . 105
19.6 Further reading . . . 105

20 Multiply and Divide Blocks **107**
20.1 Multiply and Divide Problems . . . 107
20.2 Multiplication Algorithms . . . 107
20.3 Division Algorithm . . . 108
20.4 Multiply and Accumulate . . . 108

21 ALU Flags **109**
21.1 Comparisons . . . 109

21.2 Zero Flag . . . 109
21.3 Overflow Flag . . . 109
21.4 Carry/Borrow flag . . . 109
21.5 Comparisons . . . 110
21.6 Latch ALU flags or not? . . . 110

22 Single Cycle Processors 111
22.1 Cycle Times . . . 111
22.2 Redundant Hardware . . . 111
22.3 Single Cycle Designs . . . 112

23 Multi Cycle Processors 113
23.1 Multi-Cycle Stages . . . 113
23.2 Hardware Reuse . . . 114

24 Pipelined Processors 115
24.1 Pipelining Introduction . . . 115
24.2 Pipelining Hardware . . . 116
24.3 Superpipeline . . . 118
24.4 Resources . . . 119

25 Superscalar Processors 121

26 VLIW Processors 123
26.1 VLIW Vs Superscalar . . . 123
26.2 Multi-Issue . . . 123

27 Vector Processors 125
27.1 Parallel Execution . . . 125
27.2 Non-Parallel Execution . . . 126

28 Multicore Processors 127
28.1 Symmetric Multi-core . . . 127
28.2 Asymmetric Multi-core . . . 127
28.3 Symmetric Multicore . . . 128
28.4 Asymmetric Multi-core . . . 129
28.5 further reading . . . 130

29 Exceptions 131

30 Interrupts 133
30.1 Further Reading . . . 134

31 Hazards 135
31.1 Data Hazards . . . 135
31.2 Control Hazards . . . 136
31.3 Structural Hazards . . . 136
31.4 Fixing Hazards . . . 136

32 Performance Metrics 143

33 Performance Metrics **145**
33.1 Runtime . . . 145
33.2 Processor Time . . . 147
33.3 MIPS/$ . . . 147
33.4 Latency . . . 147
33.5 MIPS/mW . . . 147
33.6 Further reading . . . 148

34 Benchmarking **149**
34.1 Benchmarks . . . 149
34.2 Common Benchmarks . . . 149
34.3 Benchmark Problems . . . 150
34.4 Further reading . . . 150

35 Optimizations **151**

36 Multi-Core Systems **153**
36.1 Symmetric Multi-core . . . 153
36.2 Asymmetric Multi-core . . . 153
36.3 Symmetric Multicore . . . 154
36.4 Asymmetric Multi-core . . . 155
36.5 further reading . . . 156

37 Memory-Level Parallelism **157**
37.1 Memory-Level Parallelism . . . 157

38 Out Of Order Execution **159**
38.1 Hazards . . . 159
38.2 Example: Intel Hyperthreading . . . 159

39 Assembler **161**

40 Simulator **163**

41 Compiler **165**
41.1 Further reading . . . 165

42 FPGA **167**

43 Photolithography **169**
43.1 Wafers . . . 169
43.2 Basic Photolithography . . . 169
43.3 packaging . . . 169
43.4 further reading . . . 170

44 Sockets and interfacing **171**
44.1 Form Factors . . . 171
44.2 Connectors . . . 172
44.3 Sockets . . . 173

45 Microcodes **175**
45.1 Further Reading . . . 176
45.2 References . . . 176

46 Register Renaming **177**

47 Cache **179**
47.1 Cache . . . 179
47.2 No cache . . . 180
47.3 Single cache . . . 180
47.4 Hit or Miss . . . 181
47.5 Cache performance . . . 181
47.6 Cache Hierarchy . . . 182
47.7 Size of Cache . . . 183
47.8 Cache Tagging . . . 184
47.9 Memory Stall Cycles . . . 185
47.10 Associativity . . . 187
47.11 Cache Misses . . . 189
47.12 Cache Write Policy . . . 191
47.13 Stale Data . . . 192
47.14 Split cache . . . 193
47.15 Error detection . . . 194
47.16 Specialized cache features . . . 194
47.17 References . . . 195
47.18 Further reading . . . 195

48 Virtual Memory **197**
48.1 Implementation . . . 197
48.2 Memory Accessing . . . 198
48.3 Pages . . . 198
48.4 Page Table . . . 198
48.5 Further reading . . . 200

49 Power Dissipation **201**
49.1 Gene's Law . . . 201
49.2 Two reasons to reduce power . . . 201
49.3 Heat . . . 202
49.4 further reading . . . 203
49.5 Resources . . . 203

50 Resources **205**
50.1 Further Reading . . . 205

51 Contributors **209**

List of Figures **211**

52 Licenses **217**
52.1 GNU GENERAL PUBLIC LICENSE . . . 217

52.2 GNU Free Documentation License 218
52.3 GNU Lesser General Public License 219

This book serves as an introduction to the field of microprocessor design and implementation. It is intended for students in computer science or computer or electrical engineering who are in the third or fourth years of an undergraduate degree. While the focus of this book will be on Microprocessors, many of the concepts will apply to other ASIC design tasks as well.

The reader should have prior knowledge in Digital Circuits and possibly some background in Semiconductors although it isn't strictly necessary. The reader also should know at least one Assembly Language. Knowledge of higher-level languages such as C or C++ may be useful as well, but are not required. Sections about soft-core design will require prior knowledge of Programmable Logic, and a prior knowledge of at least one HDL.

1 Introduction

1.1 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.

1.2 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.

1.3 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[1]
- Programmable Logic[2]
- Embedded Systems[3]
- Assembly Language[4]

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.

1.4 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.

1.5 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:

1 http://en.wikibooks.org/wiki/Digital%20Circuits
2 http://en.wikibooks.org/wiki/Programmable%20Logic
3 http://en.wikibooks.org/wiki/Embedded%20Systems
4 http://en.wikibooks.org/wiki/Assembly%20Language

- Transistor mechanics, semiconductors[5], or integrated circuit fabrication (Microtechnology[6])
- Digital Circuit[7] Logic, Design or Layout (Programmable Logic[8])
- Design or interfacing with other computer components or peripherals (Embedded Systems[9])
- Design or implementation of communication protocols used to communicate between computer components (Serial Programming[10])
- Design or creation of computer software (Computer Programming[11])
- Design of System-on-a-Chip hardware or any device with an integrated micro-controller

1.6 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.

5 http://en.wikibooks.org/wiki/Semiconductors
6 http://en.wikibooks.org/wiki/Microtechnology
7 http://en.wikibooks.org/wiki/Digital%20Circuits
8 http://en.wikibooks.org/wiki/Programmable%20Logic
9 http://en.wikibooks.org/wiki/Embedded%20Systems
10 http://en.wikibooks.org/wiki/Serial%20Programming
11 http://en.wikibooks.org/wiki/Subject%3AComputer%20programming

2 Microprocessors

2.1 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.

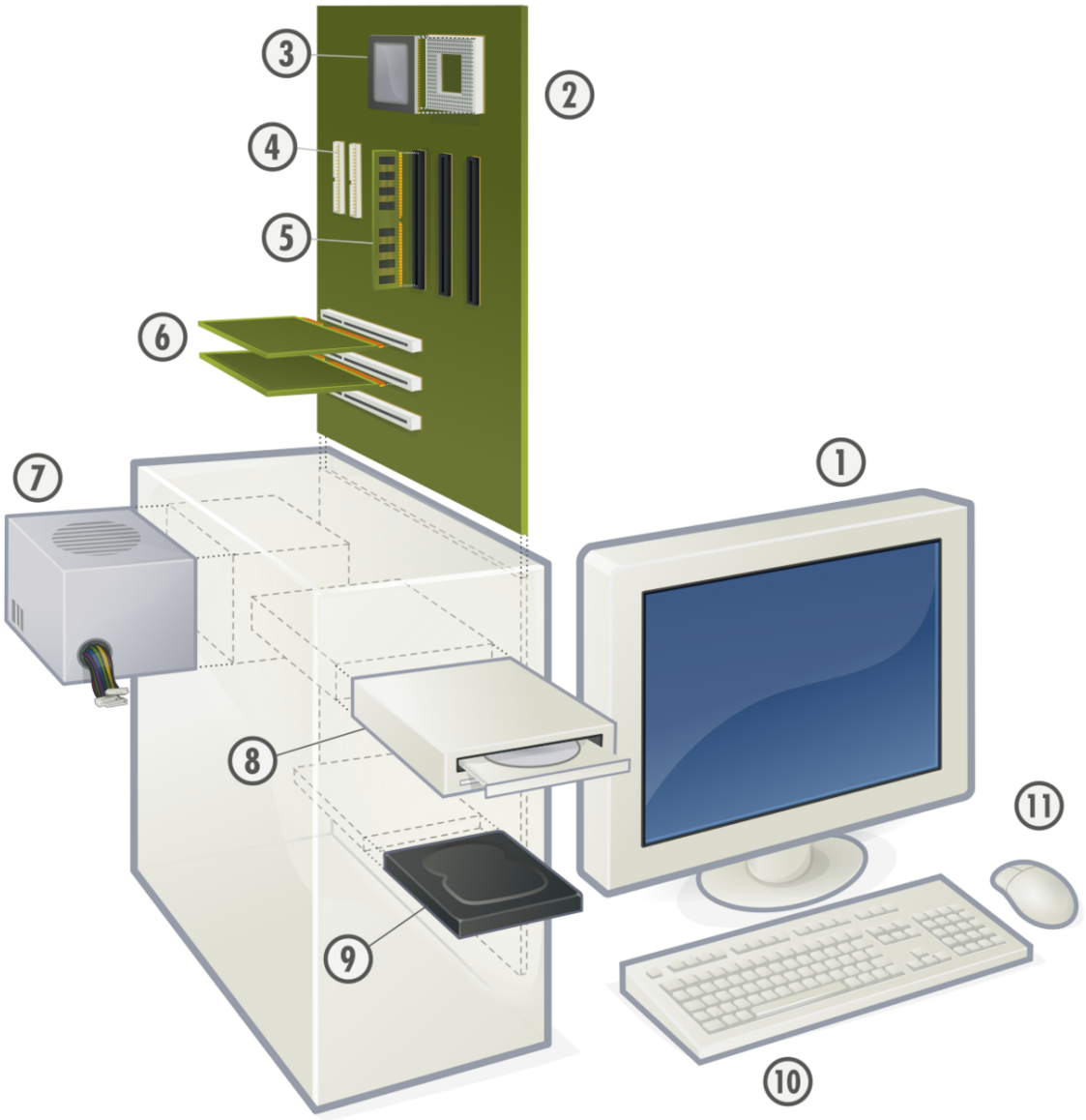

Figure 1 The components of a PC computer. Part number 3 is the CPU.

2.1.1 Types of Processors

w:Microprocessor[1]

The vast majority of microprocessors can be found in embedded microcontrollers. The second most common type of processors are common desktop processors, such as Intel's

1 `http://en.wikipedia.org/wiki/Microprocessor`

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)

2.1.2 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.

w:Digital signal processor[2]

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

2 http://en.wikipedia.org/wiki/Digital%20signal%20processor

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.

w:Graphics processing unit[3]

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.

2.1.3 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 utmost importance. An example of a real-time system is the anti-lock braking system (ABS) controller in modern automobiles.

Throughput and Routing

3 http://en.wikipedia.org/wiki/Graphics%20processing%20unit

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.

\mbox{Clock Time} = \frac{1}{\mbox{Clock Rate}}

2.2 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 but we will also describe the Operating System (OS) in brief. 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.

2.3 Operating System

Operating System is a program which acts as an interface between the system user and the computer hardware and controls the execution of application programs. It is the program running at all times on the computer, usually called the Kernel.

2.4 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.

2.4.1 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.

2.4.2 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[a].

a `http://en.wikibooks.org/wiki/Assembly%20Language`

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

2.5 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.*

Moore's Law has been used incorrectly to calculate the speed of an integrated circuit, or even to calculate its 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.

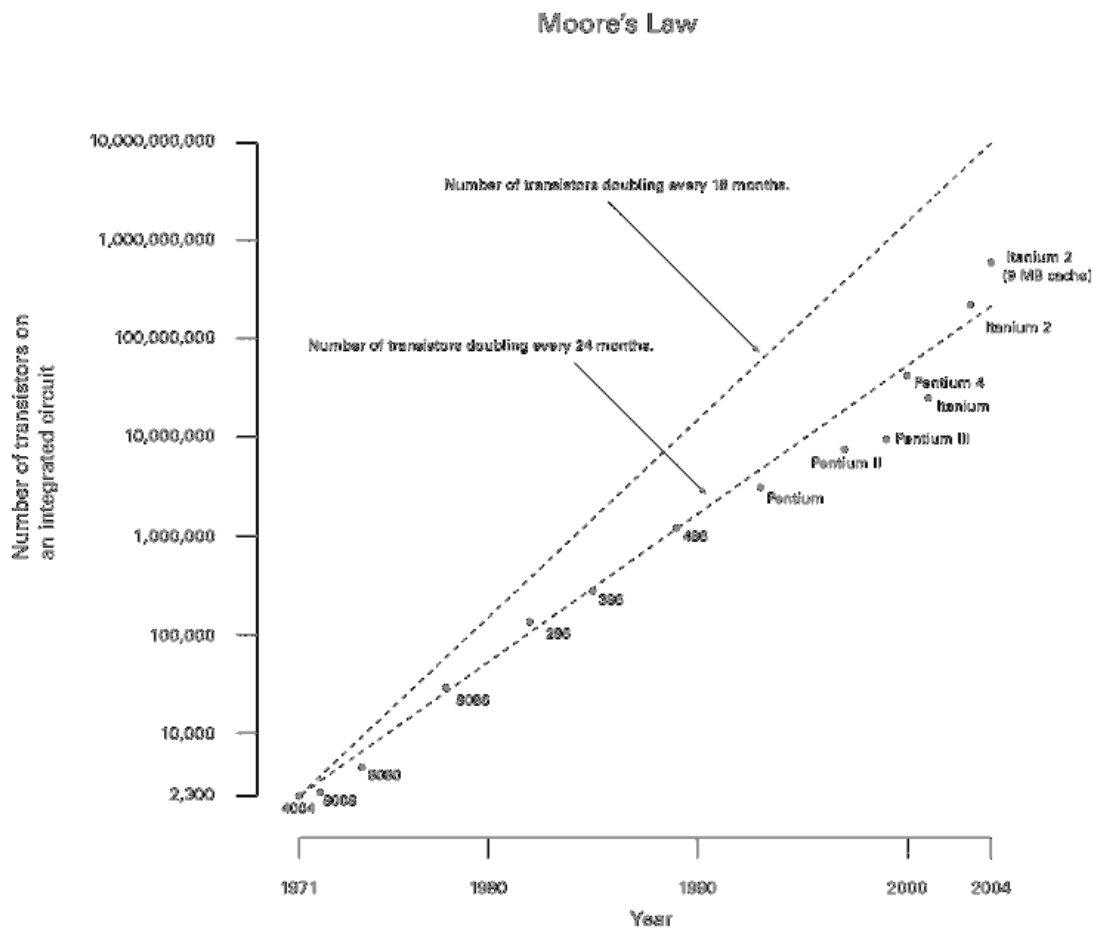

Figure 2

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).

2.6 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.

2.7 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.

3 Computer Architecture

3.1 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[1] suggested that data and programs should be stored together in memory. This is now called Von Neumann architecture[2]. Programs are fetched from memory for execution by a central unit that we call the CPU. Basically programs and data are represented in memory in the same way. The program is just data encoded with special meaning. The main criticism of this approach is, that security problems can arise when instructions can be manipulated as if they were data, and vice-versa.

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** .

3.1.1 Control

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

3.1.2 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

1 http://en.wikipedia.org/wiki/John%20von%20Neumann
2 http://en.wikipedia.org/wiki/Von%20Neumann%20architecture

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.

We discuss the control and datapath in far more detail in a later section, ../Control and Datapath/[3].

3.2 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.

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 Von Neumann architecture that is forced to read the instruction, the data sample, and the filter coefficient, one at a time.

3.3 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.

3.4 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[4] (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

3 Chapter 6 on page 35

4 http://en.wikipedia.org/wiki/Instruction_set%23Categories_of_ISA

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`http://www.cs.uiowa.edu/~jones/arch/cisc/` group stack machines with the CISC machines; some people `http://www.ultratechnology.com/ml0.htm`, `http://www.ece.cmu.edu/~koopman/stack_computers/sec6_1.html` 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[5].

3.5 Microprocessor Components

Some of the common components of a microprocessor are:

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

A brief introduction to these components is placed below.

3.5.1 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.

3.5.2 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.

5 Chapter 4.1 on page 25

3.5.3 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).

```
ALU will be discussed in far more detail in a later chapter, ../ALU/[6].
```

3.5.4 Registers

w:processor register[7] w:hardware register[8]

This book, includes data about 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.

Registers are mentioned in far more detail in a later chapter, ../Register File/[9].

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

7 http://en.wikipedia.org/wiki/processor%20register
8 http://en.wikipedia.org/wiki/hardware%20register
9 Chapter 13 on page 63

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** .

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

w:register renaming[10]

- extra physical registers to support register renaming

w:prefetch input queue[11]

- the prefetch input queue

10 http://en.wikipedia.org/wiki/register%20renaming
11 http://en.wikipedia.org/wiki/prefetch%20input%20queue

- writable control stores (We will discuss the control store in the Microprocessor Design/Control Unit[12] and Microprocessor Design/Microcode[13])
- 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.

3.5.5 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).

We discuss cache in far more detail in a later chapter, ../Cache/[14].

3.6 Endian

w:endianness[15] 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.

12 Chapter 17 on page 89
13 Chapter 45 on page 175
14 Chapter 47 on page 179
15 http://en.wikipedia.org/wiki/endianness

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.

3.7 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.

Some people claim that a processor must have a hardware stack in order to run C programs.[16]

Most computer architectures have hardware support for a recursive "call" instruction in their ../Assembly Language/[17]. Some architectures (such as the ARM, the Freescale RS08, etc.) implement "call" like this:

- the "call" instruction pushes a return address into a link register and jumps to the subroutine. A separate instruction near the beginning of the subroutine pushes the contents of the link register to a stack in main memory, to free up the link register so that subroutine can then recursively call other subroutines.

Some architectures (such as the 6502, the x86, etc.) implement "call" like this:

- the "call" instruction pushes a return address onto the stack in main memory and jumps to the subroutine.

A few architectures (such as the PIC16, the RISC I processor, the Novix NC4016, many LISP machines, etc.) implement "call" like this:

- The "call" instruction pushes a return address into a dedicated return stack, separate from main memory, and jumps to the subroutine.

3.8 further reading

w: system bus[18]

- Wikipedia: writable control stores[19]

Category:Microprocessor Design[20]

16 Walter Banks. "The Stack Controversy" ^{http://www.bytecraft.com/stack_controversy} . 2009.
17 Chapter 8 on page 41
18 http://en.wikipedia.org/wiki/%20system%20bus
19 http://en.wikipedia.org/wiki/Microcode%23Writable_control_stores%20
20 http://en.wikibooks.org/wiki/Category%3AMicroprocessor%20Design

4 Instruction Set Architectures

4.1 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.

4.1.1 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.

4.1.2 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.

4.1.3 VLIW

We will discuss VLIW Processors[1] in a later section.

4.1.4 Vector processors

We will discuss Vector Processors[2] in a later section.

4.1.5 Computational RAM

4.2 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.[3] 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.

4.3 Common Instructions

4.3.1 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.

1 Chapter 26 on page 123

2 Chapter 27 on page 125

3 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.

Instructions that move (or copy) data from one place to another are the #1 most-frequently-used instructions in most programs.[4]

4.3.2 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").[5]

4.3.3 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[6].

4.3.4 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.

4 Peter Kankowski. "x86 Machine Code Statistics" ^{http://www.strchr.com/x86_machine_code_statistics}

5 Peter Kankowski. "x86 Machine Code Statistics" ^{http://www.strchr.com/x86_machine_code_statistics}

6 http://en.wikibooks.org/wiki/Microprocessor_Design%23ALU_Design

4.3.5 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** .

4.4 Instruction Length

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

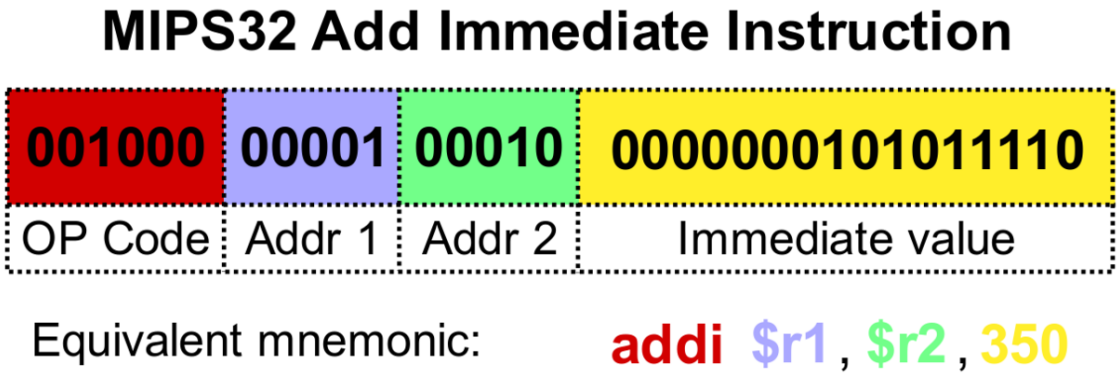

Figure 3 A MIPS "add immediate" instruction includes the opcode (logical operation), the destination register specifier, the source register specifier, and a constant value, all in the same 32 bits that are used for every MIPS instruction.

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 whether each instruction straddles a cache line or virtual memory page boundary; CPUs with fixed-length instructions can skip all that. [7]

There simply are not enough bits in a 16 bit instruction to accommodate 32 general-purpose registers, and also do "Ra = Rb (op) 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[8] and use longer fixed-width instructions, typically 32 bit, such as the MIPS and DLX and ARM.

7 The evolution of RISC technology at IBM by John Cocke ^{http://domino.watson.ibm.com/tchjr/journalindex.nsf/0/22d06c5aa961e78085256bfa0067fa93?OpenDocument} – IBM Journal of R&D, Volume 44, Numbers 1/2, p.48 (2000)

8 http://en.wikibooks.org/wiki/..%2FCode%20Density%20

- 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. 3-operand instructions allow better reuse of data[9]; 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[10] -- 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.

4.5 Further reading

w:instruction set[11] w:addressing mode[12]

9 The evolution of RISC technology at IBM by John Cocke ^{http://domino.watson.ibm.com/tchjr/journalindex.nsf/0/22d06c5aa961e78085256bfa0067fa93?OpenDocument} – IBM Journal of R&D, Volume 44, Numbers 1/2, p.48 (2000)

10 Chapter 3.5.4 on page 20

11 http://en.wikipedia.org/wiki/instruction%20set

12 http://en.wikipedia.org/wiki/addressing%20mode

5 Memory

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

5.1 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.

5.2 Hard Disk Drives

w:Hard disk drive[1] Hard Disk Drives (HDD) and solid-state drives (SSD) 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.

5.3 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.

1 `http://en.wikipedia.org/wiki/Hard%20disk%20drive`

5.3.1 SRAM

w:SRAM[2] 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.

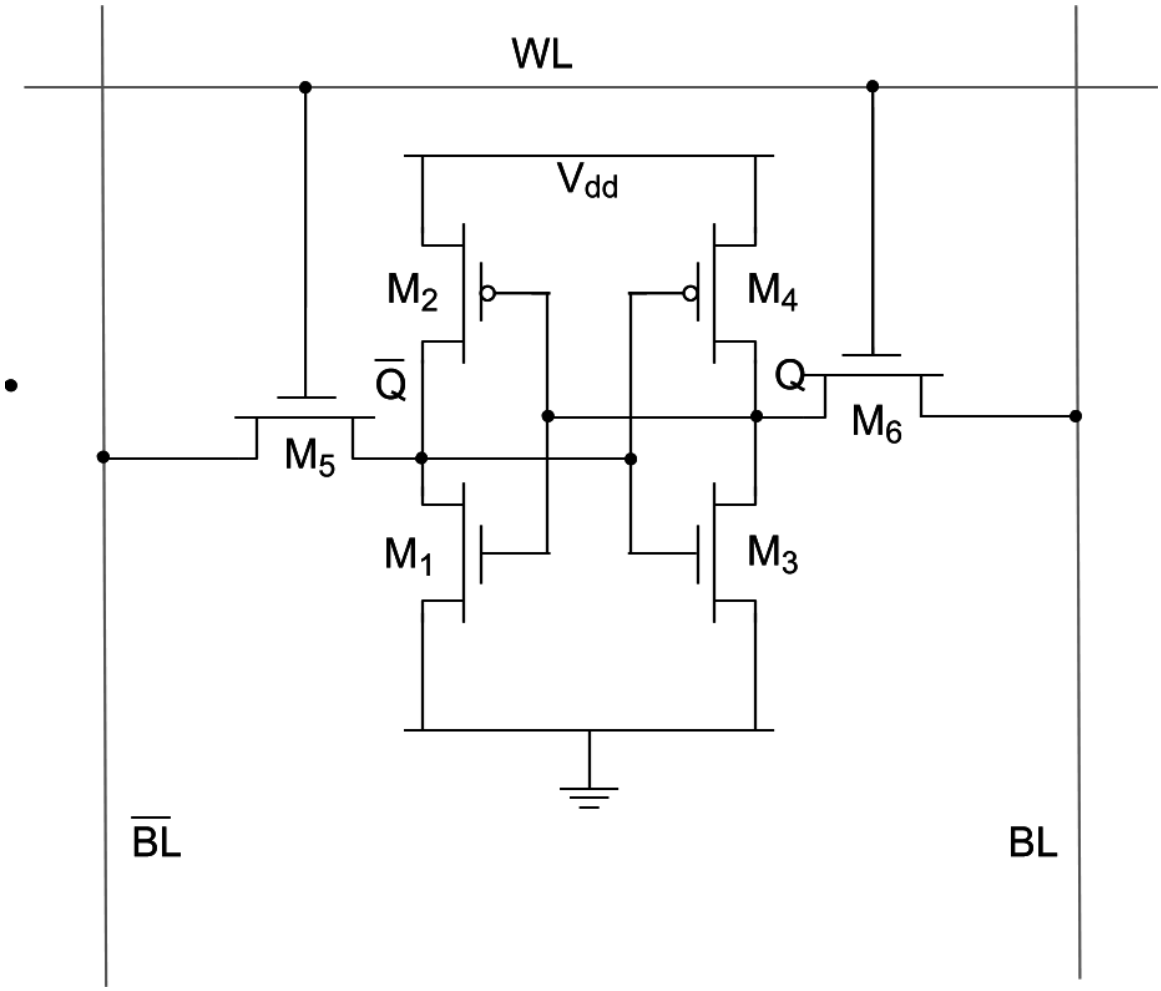

Figure 4 A single bit of storage in SRAM, showing 6 transistors.

SRAM is typically used in processor caches because of its faster speed, but not in main memory because it takes up more space.

5.3.2 DRAM

w:DRAM[3] 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

2 http://en.wikipedia.org/wiki/SRAM
3 http://en.wikipedia.org/wiki/DRAM

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.

5.3.3 Other RAM

5.4 Cache

w:CPU cache[4] 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.

5.5 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[5].

4 http://en.wikipedia.org/wiki/CPU%20cache

5 Chapter 13 on page 63

6 Control and Datapath

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 ...

For good code density[1], 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]

After a person has designed the data path, that person finds all the control signal inputs to that datapath -- all the control signals that are needed to specify how data flows through that datapath.

- Each general-purpose register needs at least one control signal to control whether it maintains the current value or loads a new value from elsewhere.
- The ALU needs some control signals to tell it whether to add, subtract, etc.
- The program counter section needs control signals to tell it whether the program counter gets reloaded with an incremented version of the previous value, or with some completely different branch value.
- etc.

1 `http://en.wikibooks.org/wiki/..%2FCode%20Density%20`

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}`

Once we know what control signals we need to generate, we need to design an ../Instruction Decoder[4] to generate those signals.

6.1 References

4 Chapter 12 on page 61

7 Performance

7.1 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* .

7.2 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.

7.3 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.

7.4 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.

7.5 Performance

7.6 Amdahls Law

w:Amdahl's Law[1]

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:

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

1 http://en.wikipedia.org/wiki/Amdahl%27s%20Law

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.

7.7 Benchmarking

- SpecInt
- SpecFP
- "Maxim/Dallas APPLICATION NOTE 3593"[2] benchmarking
- "Mod51 Benchmarks"[3]
- EEMBC, the Embedded Microprocessor Benchmark Consortium[4]

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

8 Assembly Language

8.1 Assemblers

Assemblers take in human-readable assembly code and produce machine code.

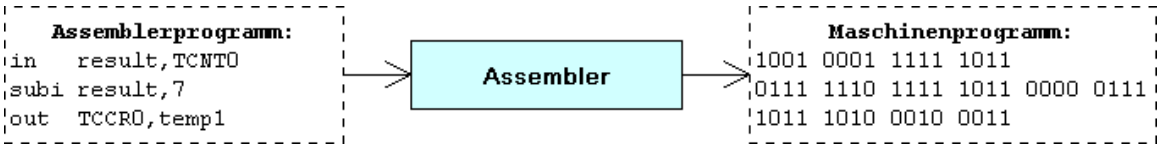

Figure 5

8.2 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 instruction 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

8.3 Load and Store

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

register

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

8.4 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

8.5 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.

8.6 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.

8.7 Further reading

- Assembly Language[1]

1 http://en.wikibooks.org/wiki/Assembly%20Language

9 Design Steps

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:

9.1 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

9.2 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.

9.3 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.

9.4 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[1]" 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".

w: non-blocking synchronization [2]

- 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.

1 http://en.wikibooks.org/wiki/..%2FCode%20Density%20
2 http://en.wikipedia.org/wiki/%20non-blocking%20synchronization%20

- CPUs dedicated to a single application (ASICs or FPGAs) led to the idea of customizing the CPU for one particular application[3]

[4] w: Popek and Goldberg virtualization requirements [5]

- The rise of viruses and other malware led to the recognition of the Popek and Goldberg virtualization requirements.

9.5 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.

We go into much more detail on control unit design in the following sections, ../Control and Datapath[6] and ../Instruction Decoder[7].

9.6 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. This enables virtual memory, memory protection, and other desirable features.

We talk more about the benefits and drawbacks of a MMU, and how to implement it, in Microprocessor Design/Virtual Memory[8].

9.7 Verify the design

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

3

4 "Generating instruction sets and microarchitectures from applications" ^{http://portal.acm.org/citation.cfm?id=191326.191501} by Ing-Jer Huang, and Alvin M. Despain

5 http://en.wikipedia.org/wiki/%20Popek%20and%20Goldberg%20virtualization%20requirements%20

6 Chapter 6 on page 35

7 Chapter 12 on page 61

8 Chapter 48 on page 197

9.8 Further reading

w:functional verification[9]

- Kong and Patterson. "Instruction set design". 1995.http://www.cs.berkeley.edu/~pattrsn/152/lec3.ps

9.9 References

9 http://en.wikipedia.org/wiki/functional%20verification

10 Basic Components

10.1 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[1].

10.2 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.

10.2.1 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.

1 http://en.wikibooks.org/wiki/Digital%20Circuits

10.3 Multiplexers

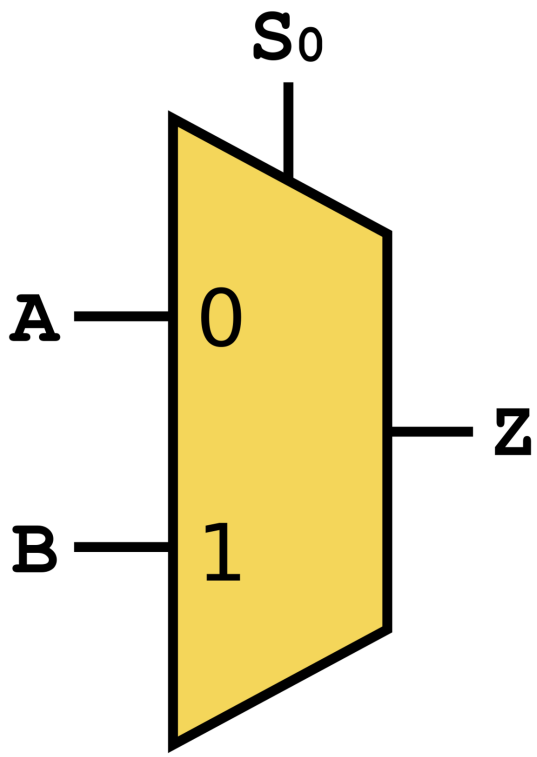

Figure 6

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.

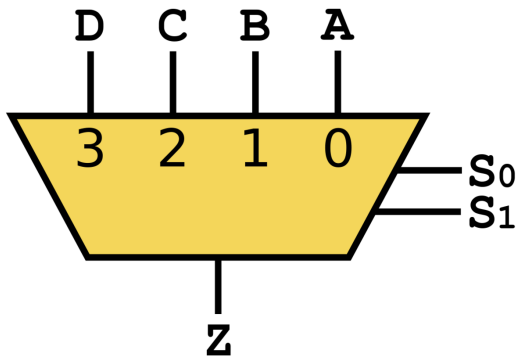

Figure 7
A 4 input Multiplexer with 2 control signal wires

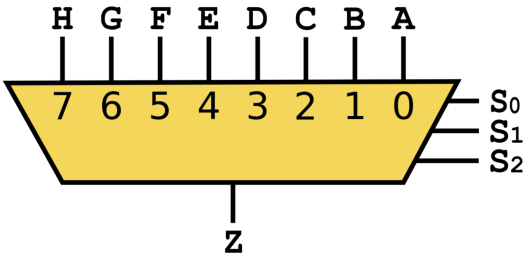

Figure 8
An 8 input Multiplexer with 3 control signal wires

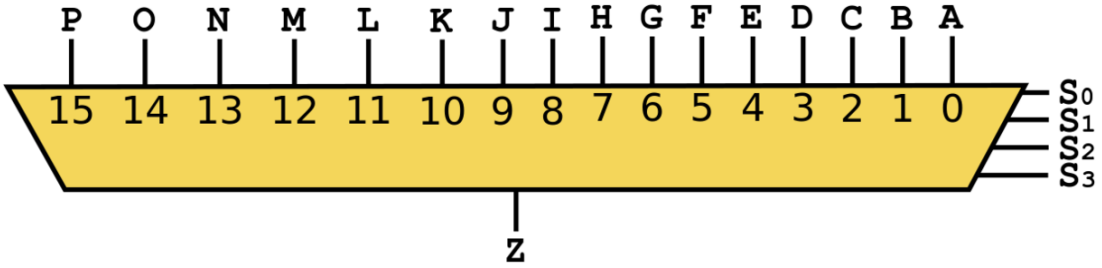

Figure 9
A 16 input Multiplexer with 4 control wires

There can be decoders implemented in the components.

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 O0  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.
```

10.4 Adder

11 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.

11.1 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.

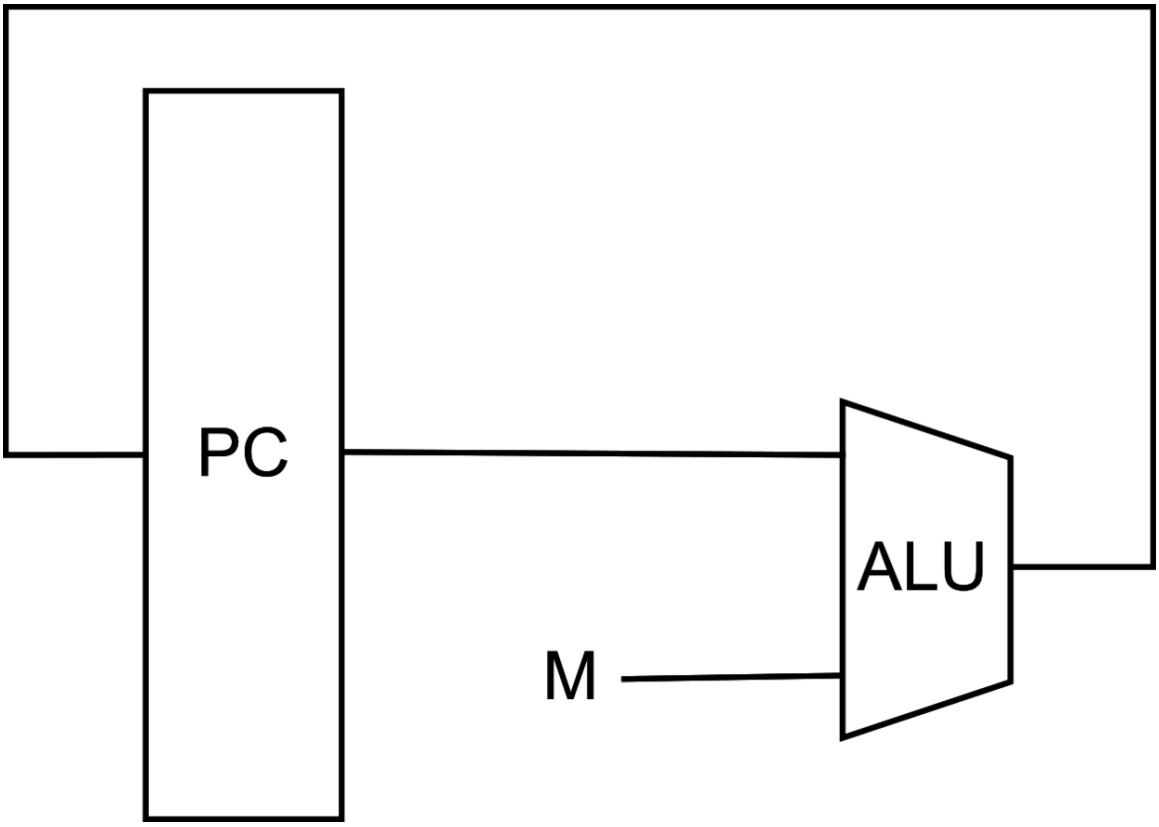

Figure 10

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.

11.2 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.

11.2.1 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.

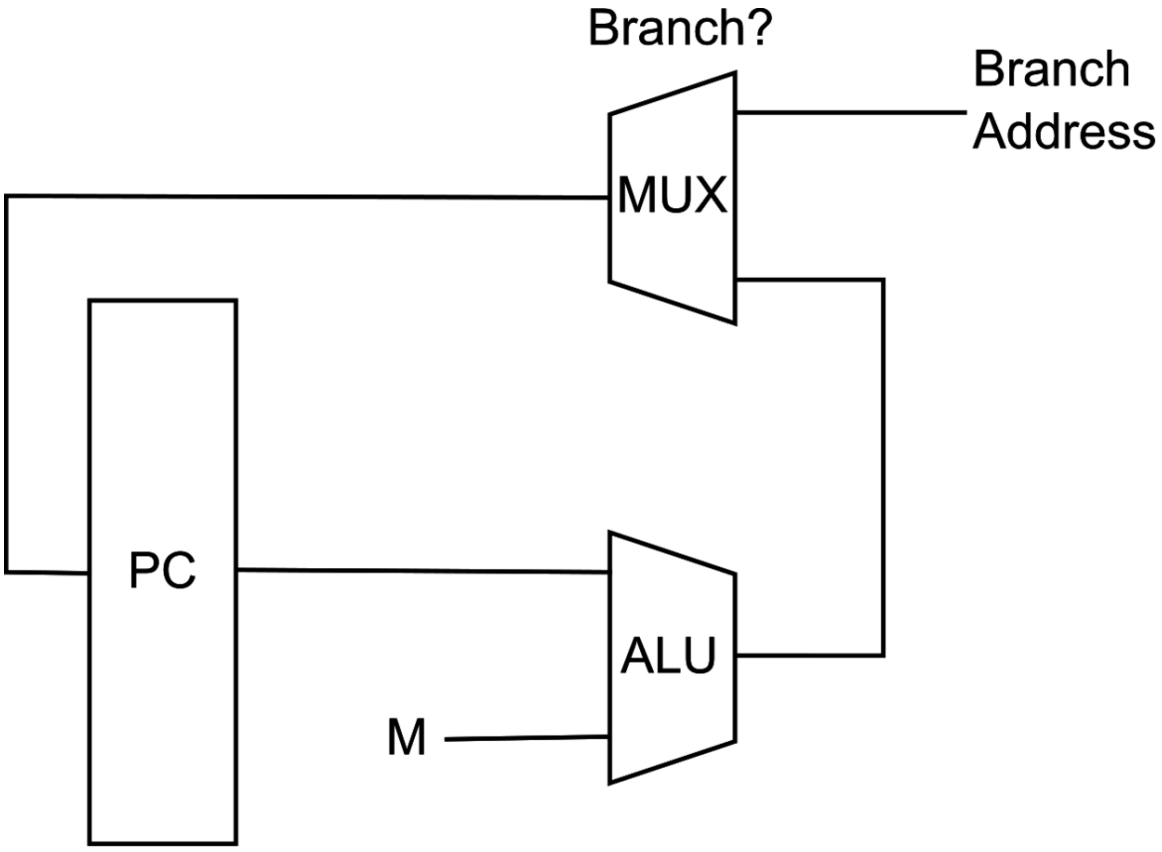

Figure 11

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.

11.2.2 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.

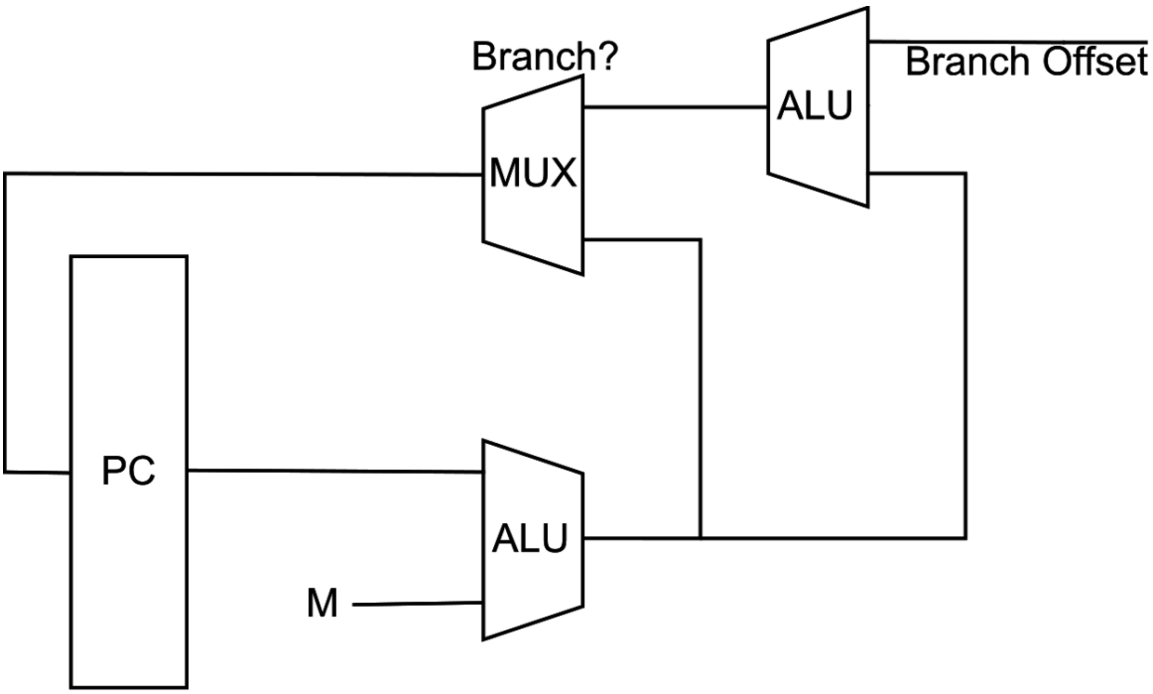

Figure 12

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:

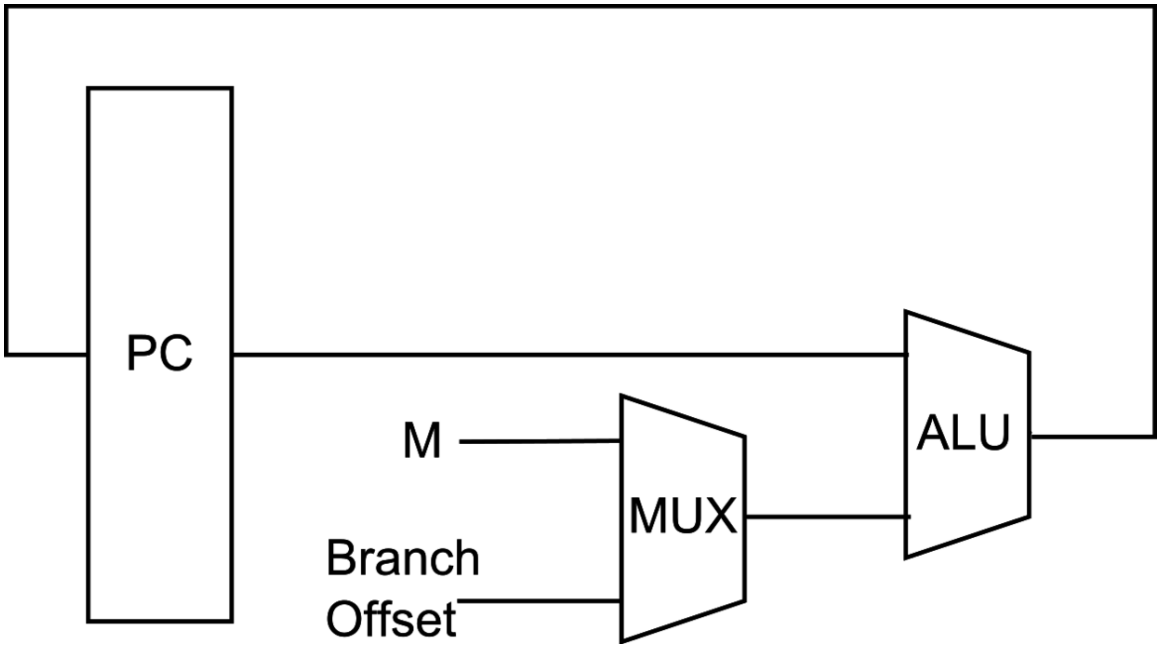

Figure 13

These are just two possible configurations for this circuit.

11.2.3 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.

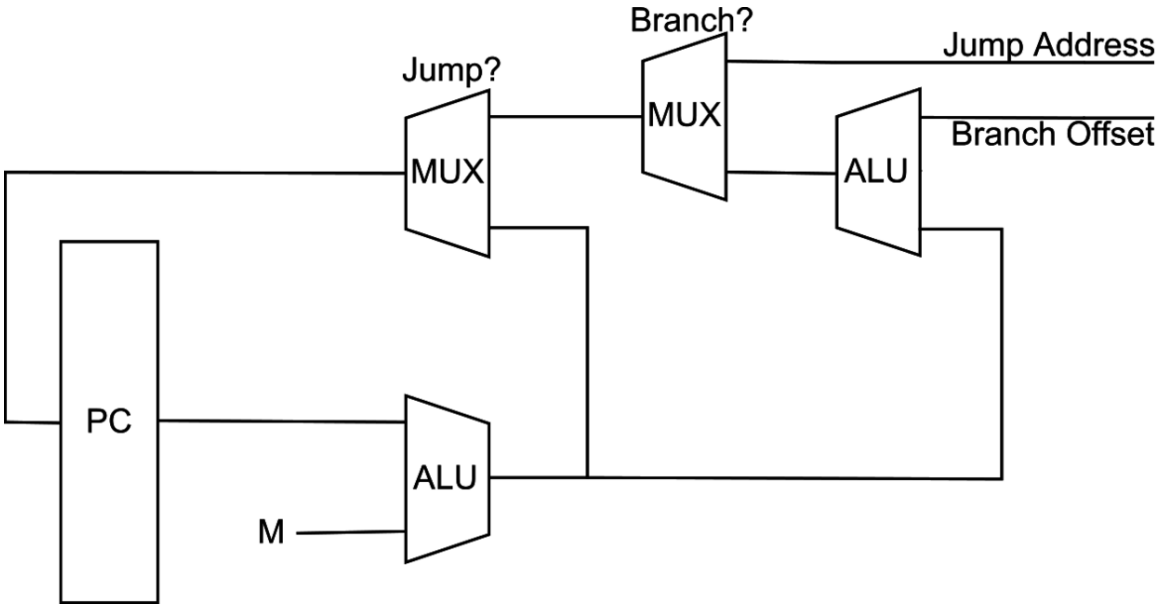

Figure 14

12 Instruction Decoder

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

For each machine-language instruction, the control unit produces the sequence of pulses on each control signal line required to implement that instruction (and to fetch the next instruction).

If you are lucky, when you design a processor you will find that many of those control signals can be "directly decoded" from the instruction register. For example, sometimes a few output bits from the instruction register IR can be directly wired to the "which function" inputs of the ALU. Even if those bits mean something completely unrelated in non-ALU instructions, it's OK if the ALU performs, say, a bogus SUBTRACT, while the rest of the processor is executing a STORE instruction.

The remaining control signals that cannot be decoded from the instruction register -- if you are unlucky, *all* the control signals -- are generated by the control unit, which is implemented as a [Moore machine][2] or a [Mealy machine][3]. There are many different ways to implement the control unit.

If you design a processor with a Princeton architecture -- your processor normally pulls instructions from the same single-ported memory used to read and write data -- then you are forced to have at least LOAD and STORE take more than one clock cycle to execute. (One cycle for the data, and another cycle to read the next instruction). (Many processors are designed with "single-cycle execution", either very simple Harvard architecture processors, or complicated high-performance processors with a separate instruction cache).

12.1 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, and processor reads in the entire instruction into the instruction register. We can decode an instruction, therefore, by simply separating the machine word in the instruction register into small parts using wire slices.

12.2 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.

w:control store[1] w:microprogram[2] Perhaps the conceptually simplest and most general-purpose approach is to implement the control unit with a very wide control store ROM holding the microprogram. A pipeline register latches all the output bits of the control store ROM every clock cycle.

Each clock cycle the pipeline register latches a new set of bits.

The output of the pipeline register has 2 sections: Control bits that go out to all the other bits and pieces of the processor. The "microPC" that feeds back to some of the address inputs of the control store ROM. Some people hardwire the carry flag to one of the address inputs of the control store ROM.

Every time a new opcode is fetched from main memory, typically the high bits of the microPC are loaded with the opcode, and the low bits of the microPC reset to zero. (To make things easier to debug, some designers load the opcode into both a separate instruction register IR as well as the microPC register, at least in the initial prototypes. Once the design is debugged, it might turn out that some or all the bits from the IR or the microPC register or both are never used, and so can be left out of the final design). During execution of the instruction, each clock cycle the pipeline register loads a new microPC address from the control store and a new set of control bits from the control store. Typically the person who writes the microprogram -- burned into the control store ROM -- designs the next-address output bits of that ROM to sequentially increment for the first few cycles of the implementation of that opcode. Then the microprogram for every "normal" opcode eventually jumps to one common section of the control store ROM that handles fetch-next-instruction.

1 http://en.wikipedia.org/wiki/control%20store
2 http://en.wikipedia.org/wiki/microprogram

13 Register File

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]

---

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.

13.1 Register File

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

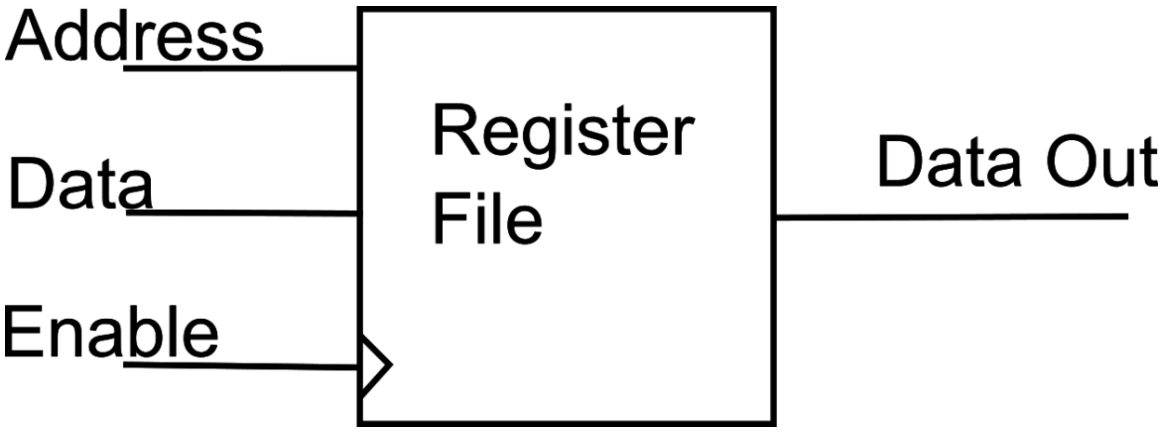

Figure 15

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.

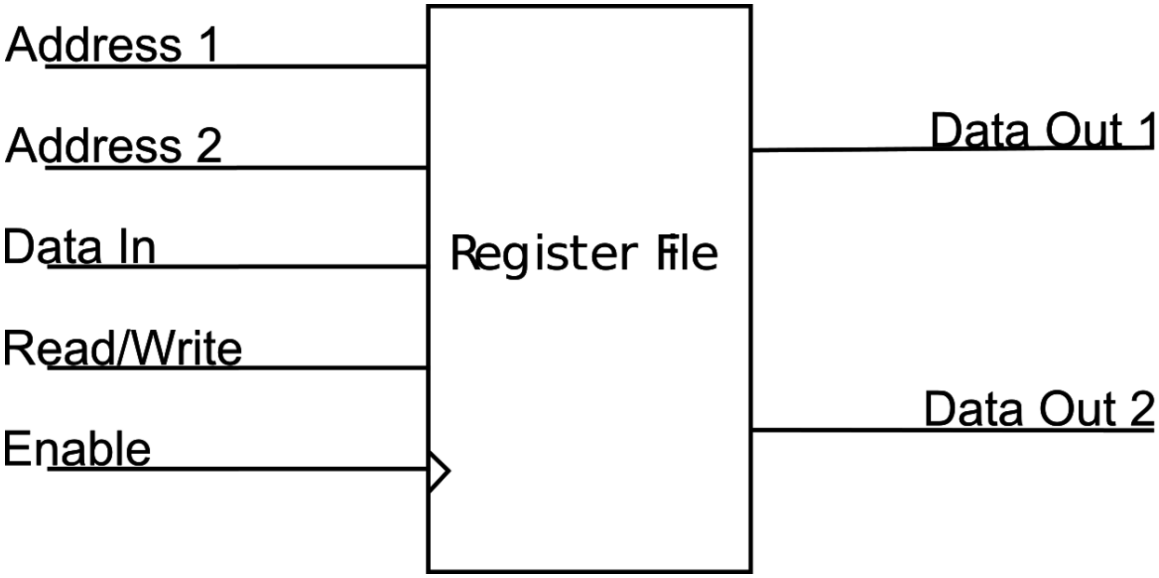

Figure 16

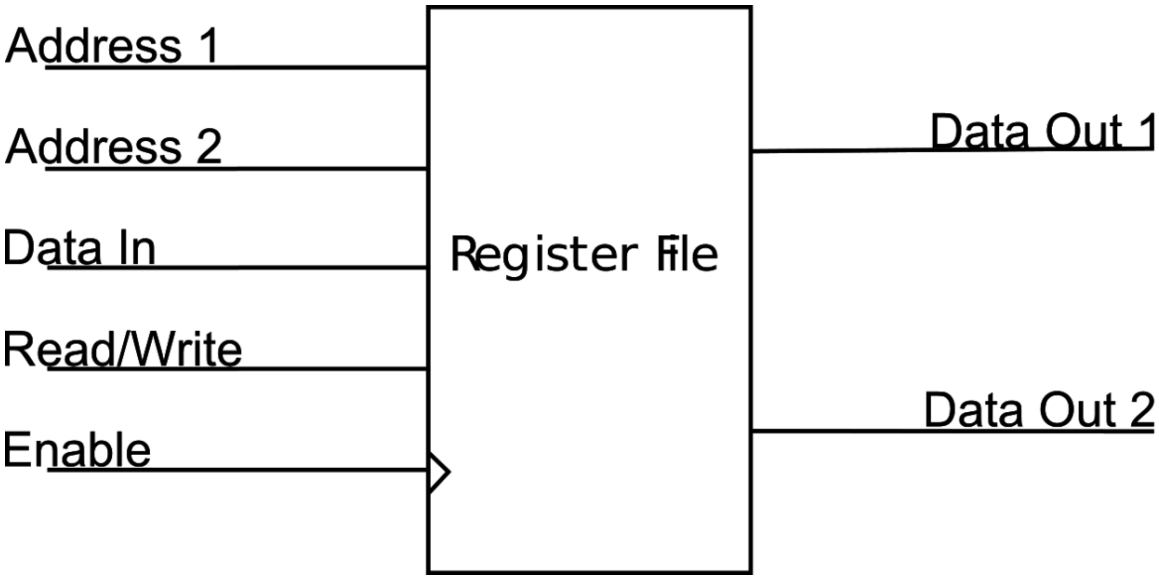

Figure 17

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

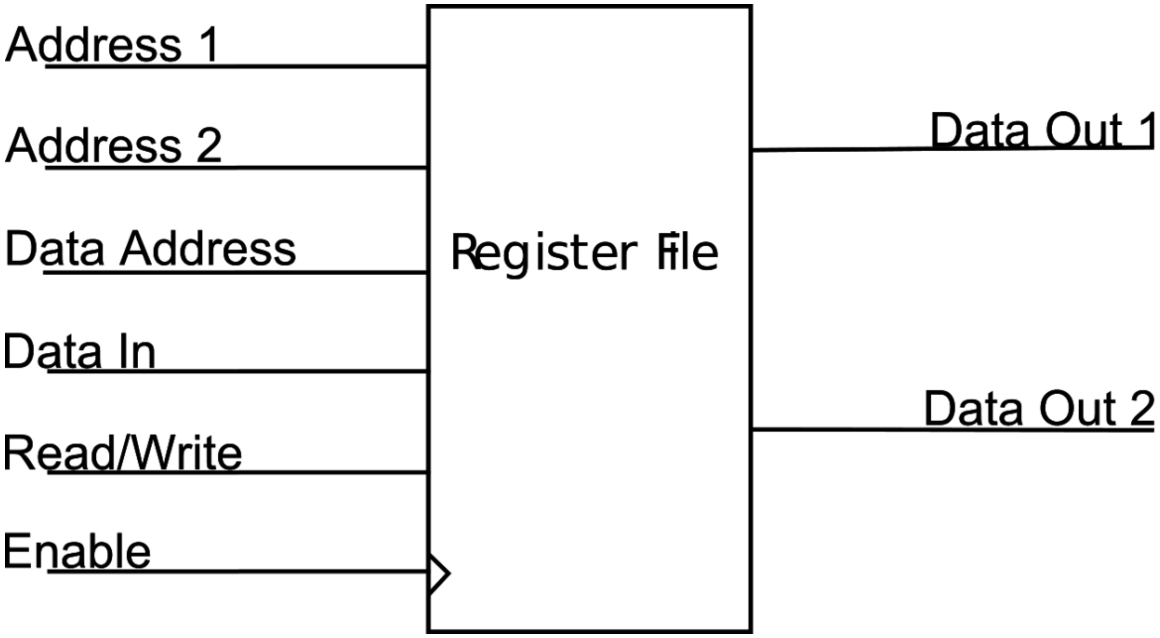

Figure 18

13.2 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

[3] and

3 "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.

- using a ”register bank”.

13.3 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.

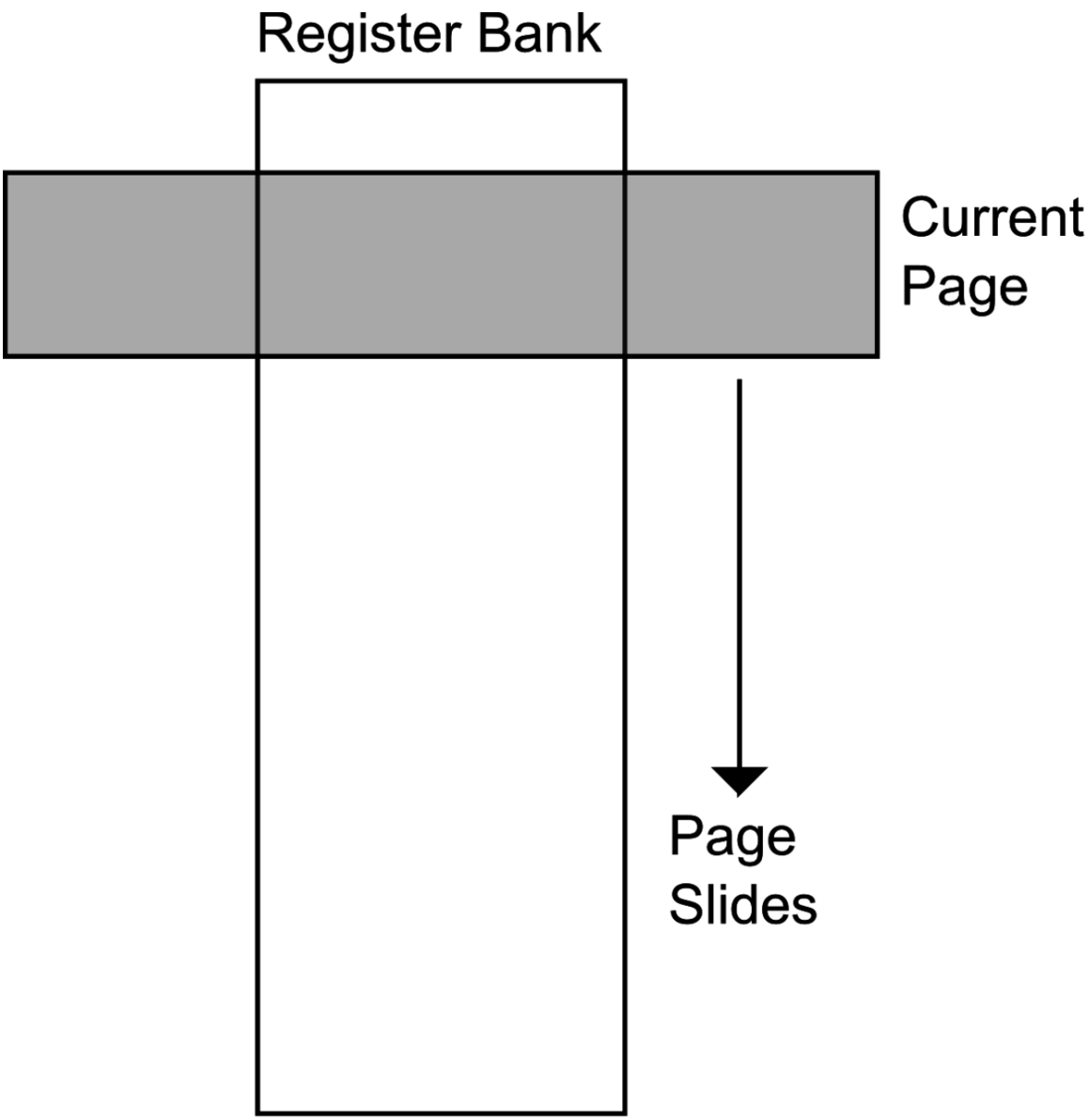

Figure 19

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:

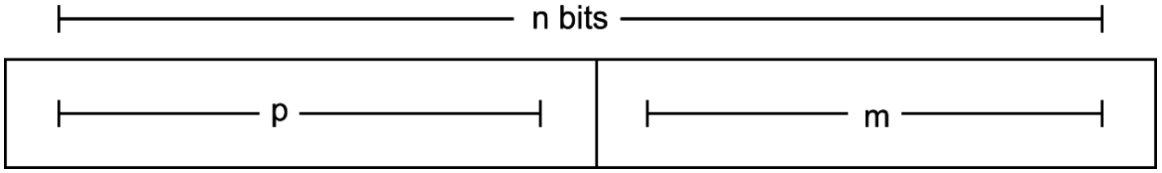

Figure 20

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.

13.4 References

14 Memory Unit

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.

14.1 Memory Unit

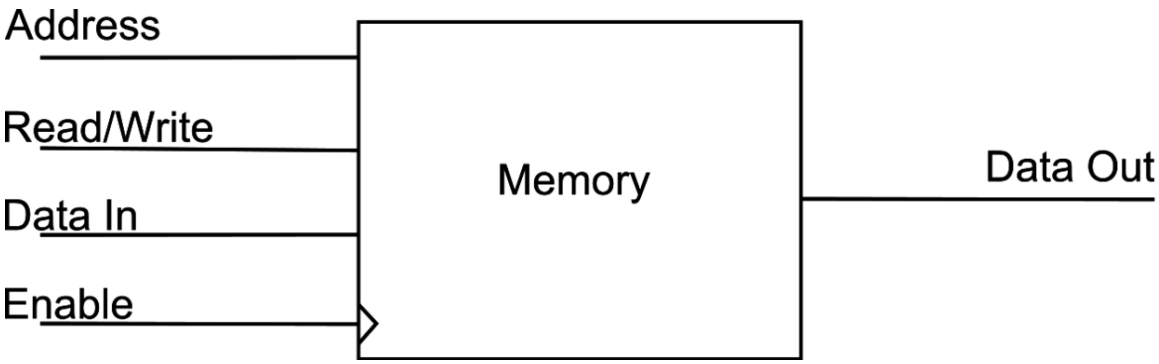

Figure 21

14.2 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.

14.3 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.

15 ALU

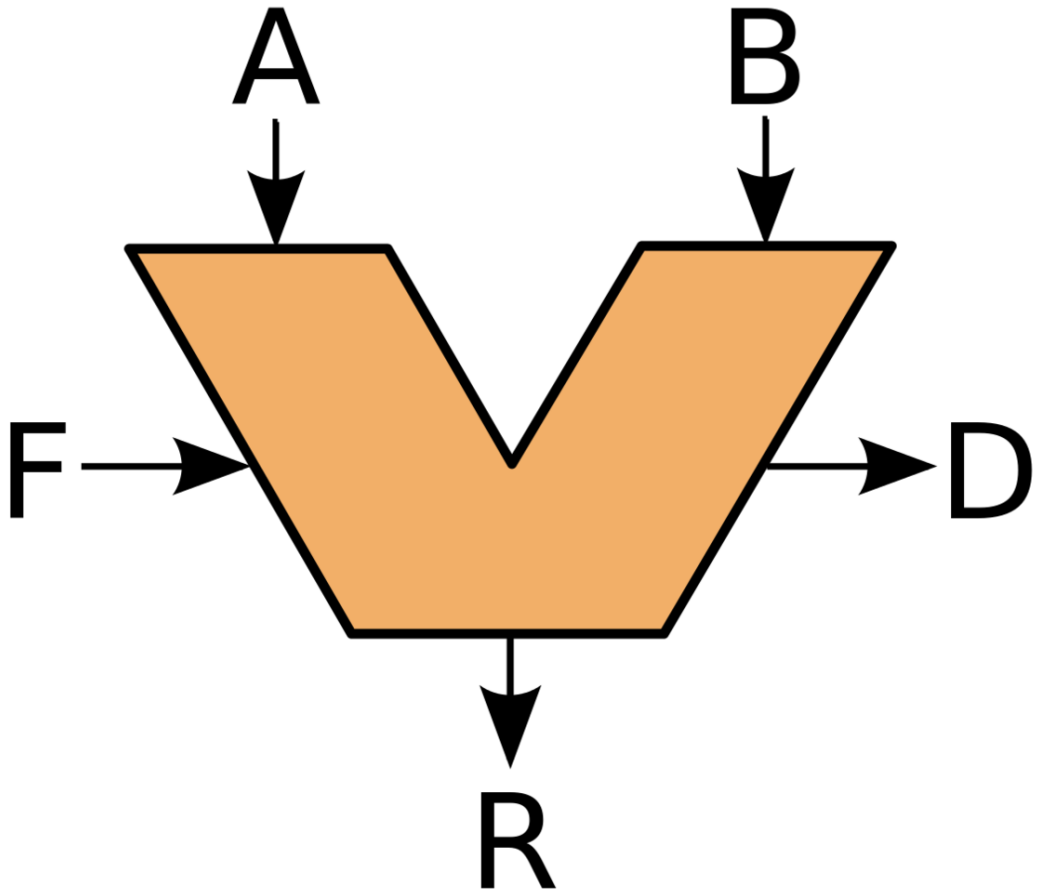

Figure 22

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. [1]

1 CPU designers have used a variety of names for the arithmetic logic unit, including "ALU", "integer execution unit", and "E-box". Paul V. Bolotoff. "Functional Principles of Cache Memory" ^{http://alasir.com/articles/cache_principles/cache_hierarchy.html} 2007.

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.

15.1 Tasks of an ALU

ALU units typically need to be able to perform the basic logical operations (AND, OR) and 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.

15.2 ALU Slice

15.3 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.

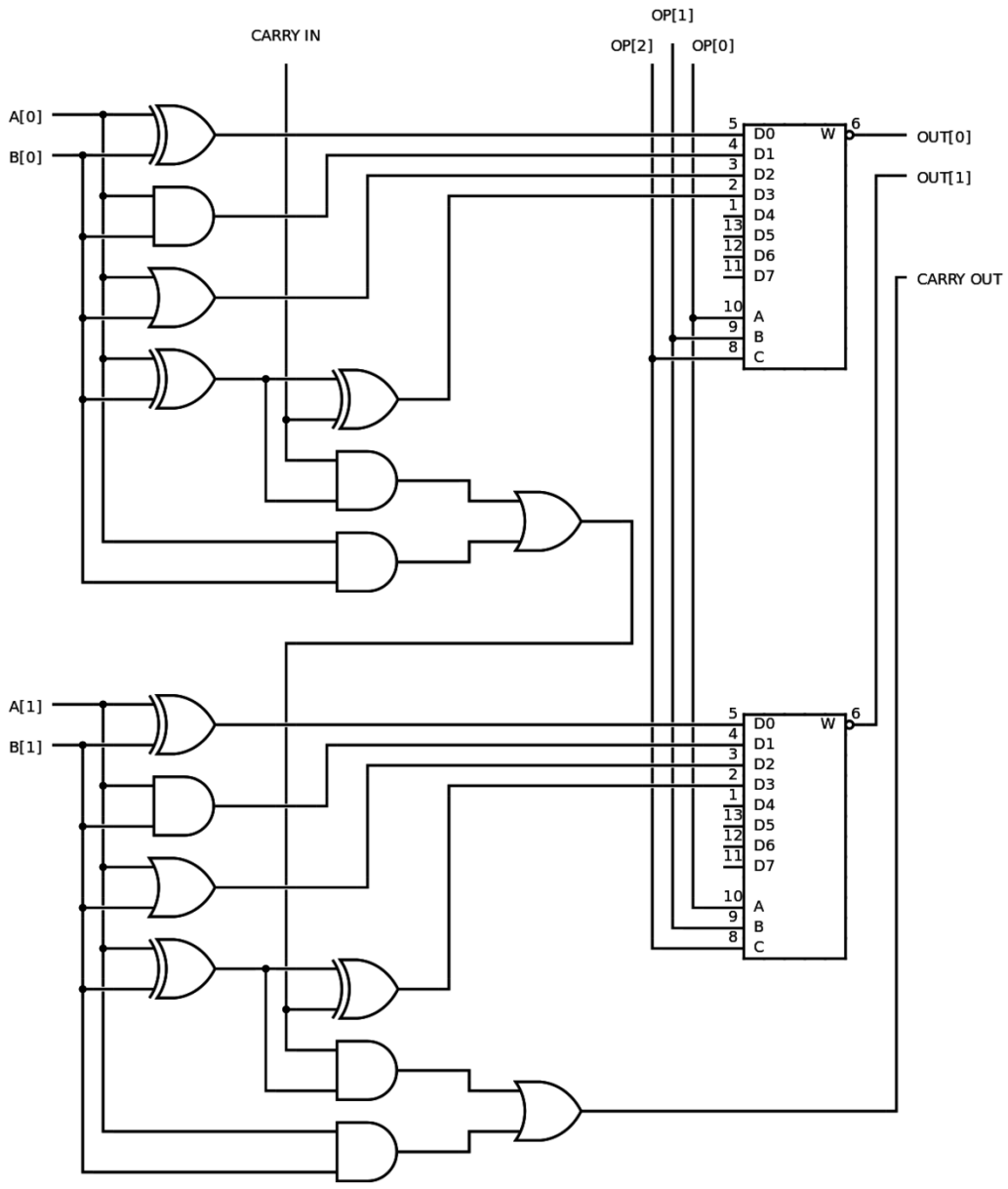

Figure 23

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.

15.4 Example: 4-Bit ALU

Here is a circuit diagram of a 4 bit ALU.

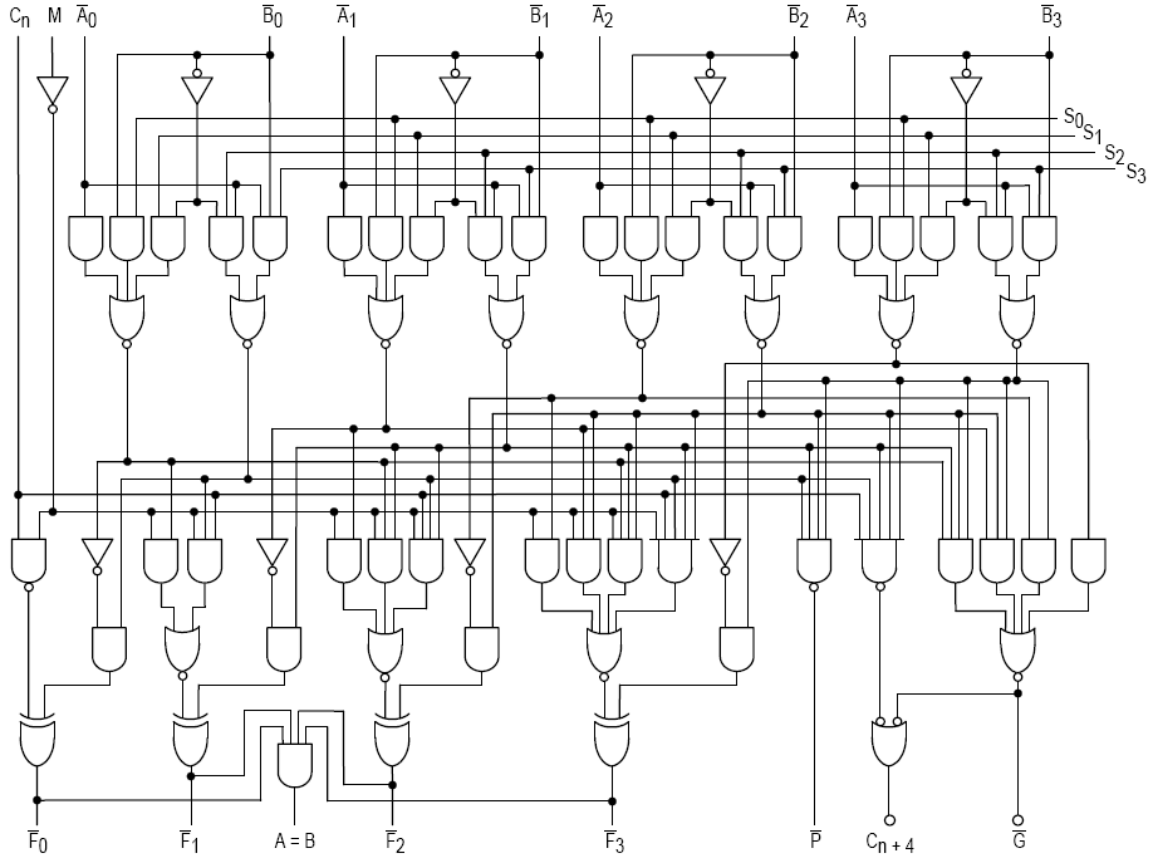

Figure 24

15.5 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).

15.6 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. [2] In all images below, the orange represents memory structures internal to the CPU (registers), and the purple represents external memory (RAM).

15.6.1 Accumulator

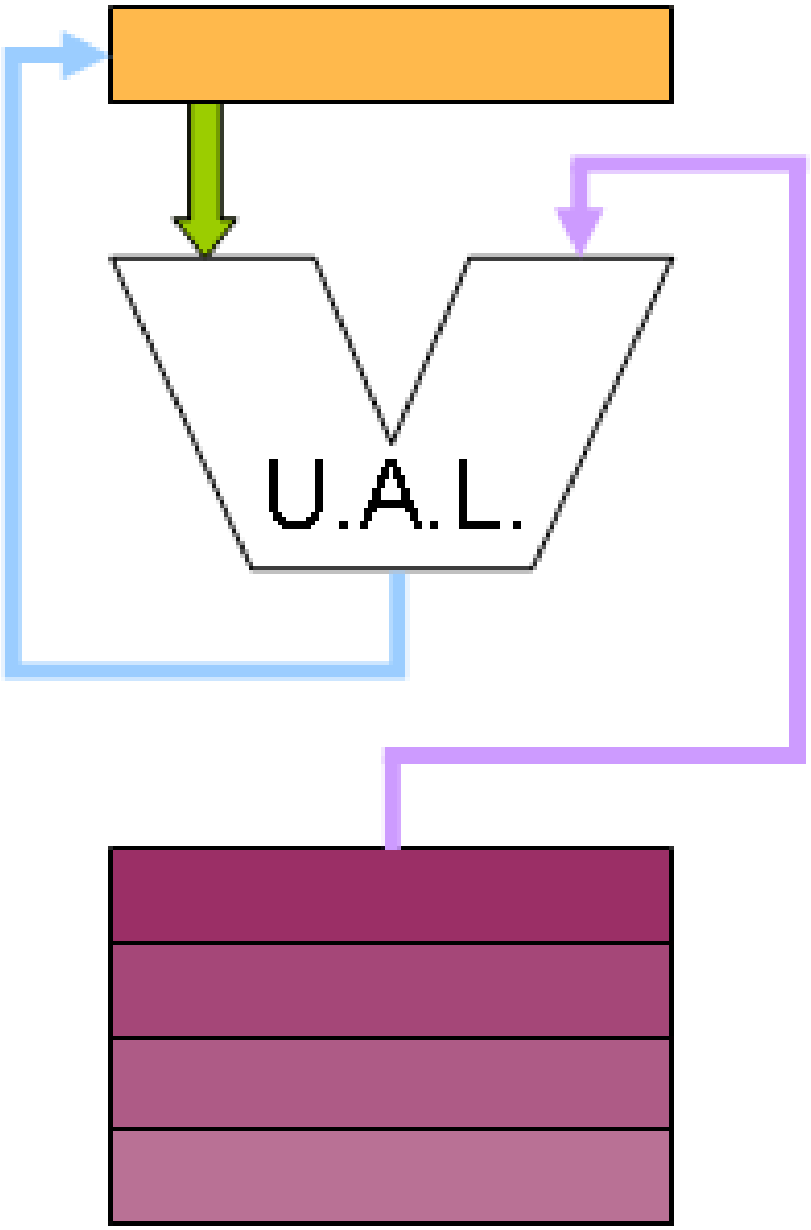

Figure 25

2 "Instruction Set Principles: Basic ISA Classes" ^{http://users.encs.concordia.ca/~tahar/coen6741/notes/Chapter2-4p.pdf} by Dr. Sofiène Tahar

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.

15.6.2 Register-to-Register

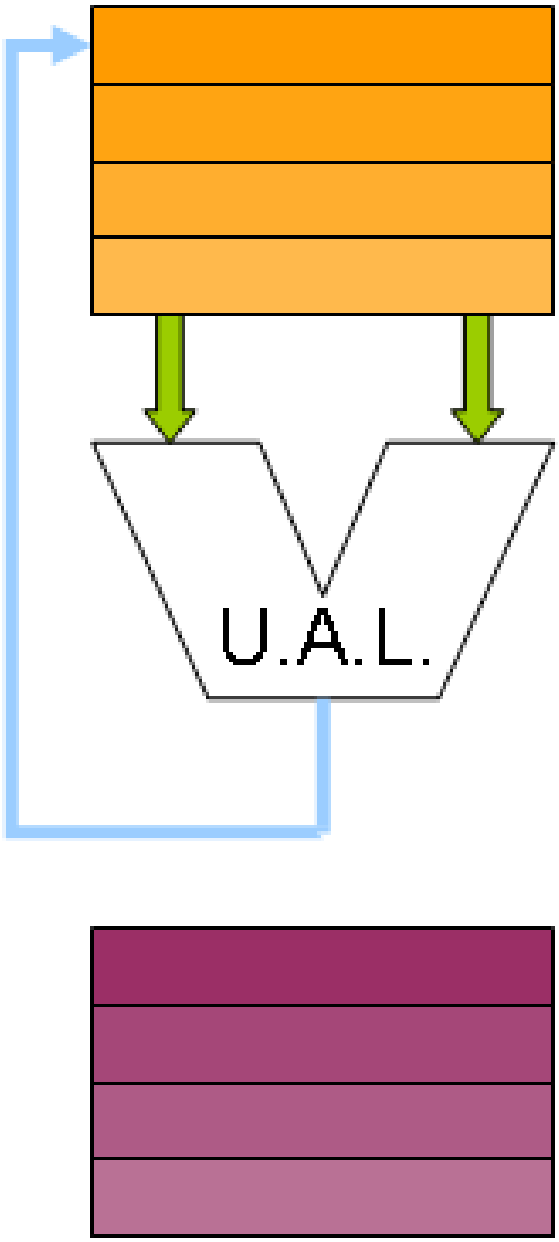

Figure 26

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).

15.6.3 Register Stack

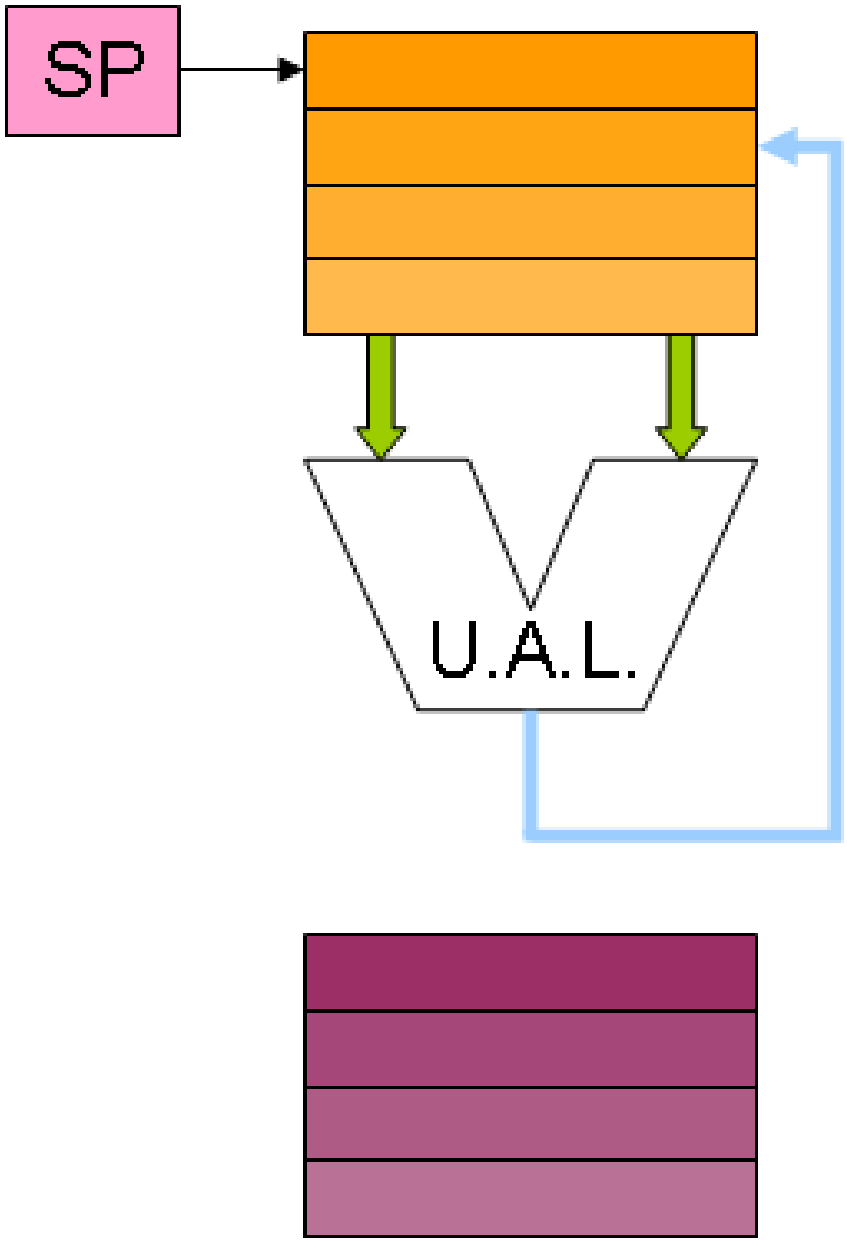

Figure 27

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.

15.6.4 Register-and-Memory

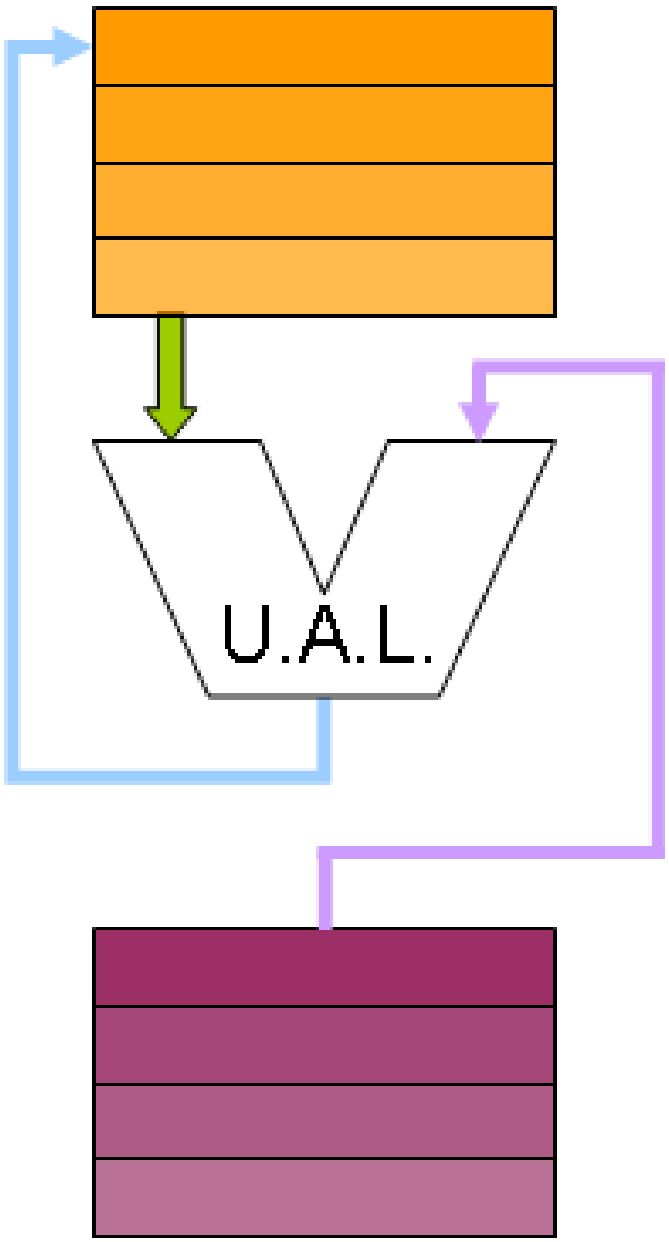

Figure 28

One complicated structure is a Register-and-Memory structure, like that shown at left. 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.

15.6.5 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.

15.6.6 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** .

15.6.7 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** .

15.7 References

- Digital Circuits/ALU[3]
- Electronics/ALU[4]

3 http://en.wikibooks.org/wiki/Digital%20Circuits%2FALU
4 http://en.wikibooks.org/wiki/Electronics%2FALU

16 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.

16.1 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[1] 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:

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

16.1.1 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.

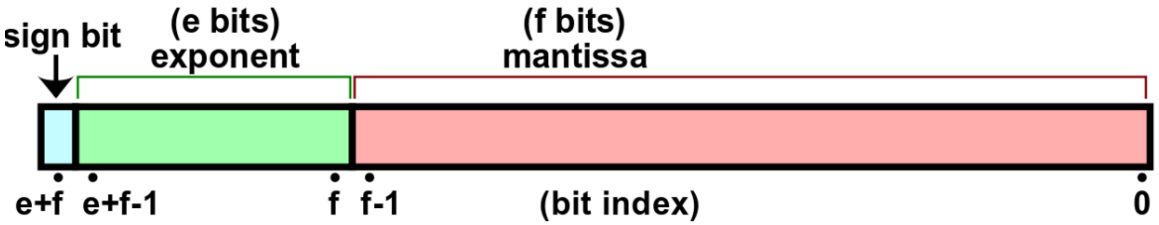

Figure 29

1 http://en.wikibooks.org/wiki/Floating%20Point

16.1.2 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).

16.1.3 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).

16.2 Floating Point Unit Design

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

For addition/Subtraction

- A comparator (subtractor) 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.

16.3 Further Reading

- Floating Point[2]

2 http://en.wikibooks.org/wiki/Floating%20Point

17 Control Unit

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.

17.1 Simple Control Unit

In its 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.

17.2 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.

18 Add and Subtract Blocks

18.1 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.

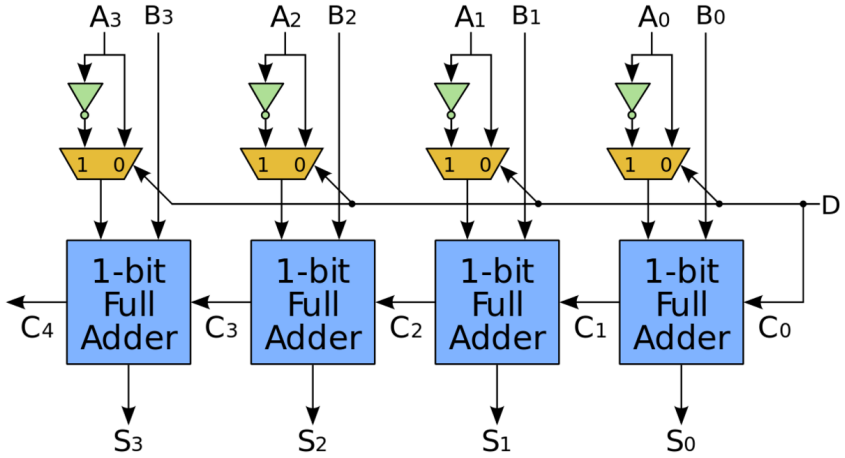

Figure 30

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

18.2 Bit Adders

18.2.1 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.

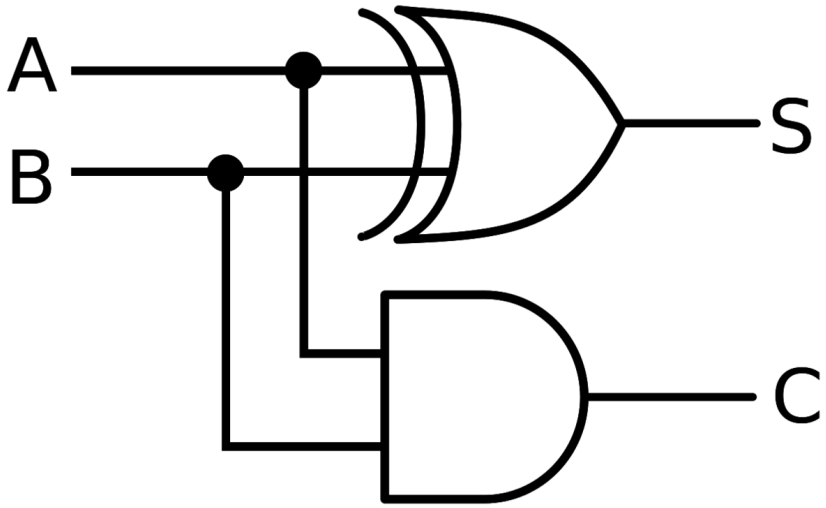

Figure 31

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
```

18.2.2 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.

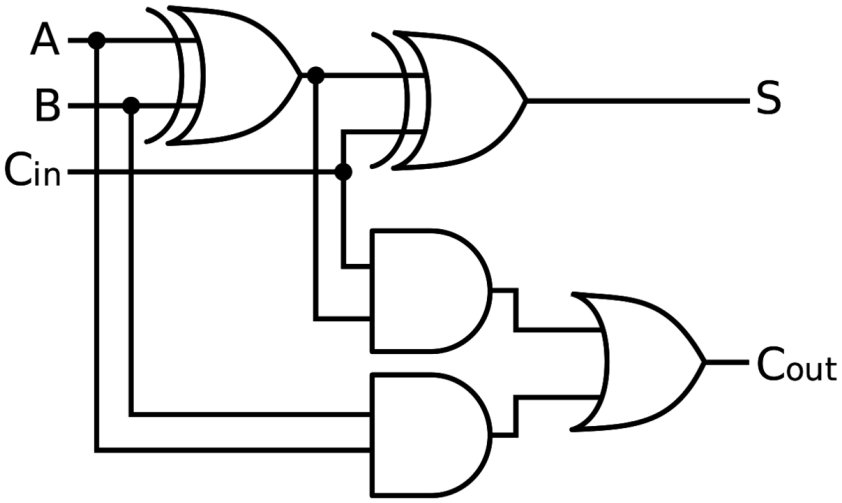

Figure 32

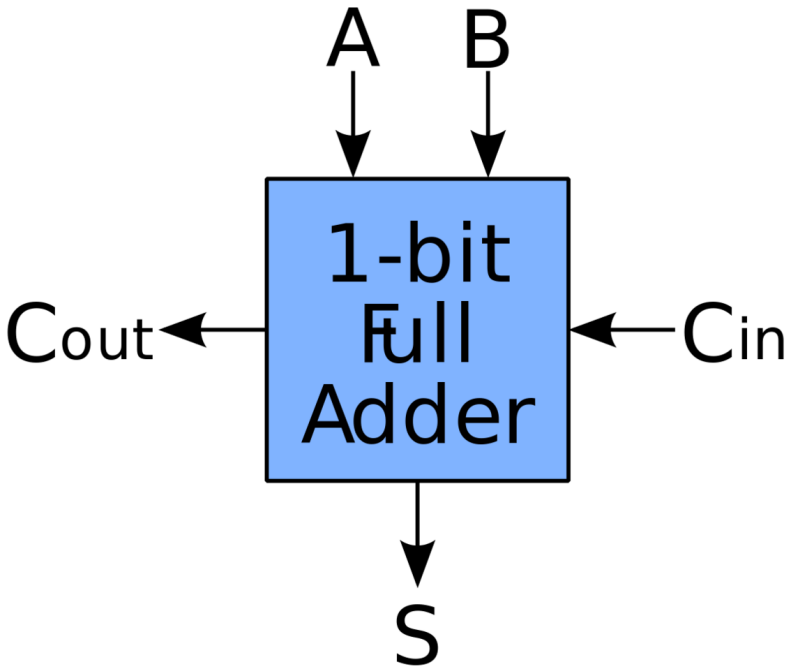

Figure 33

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

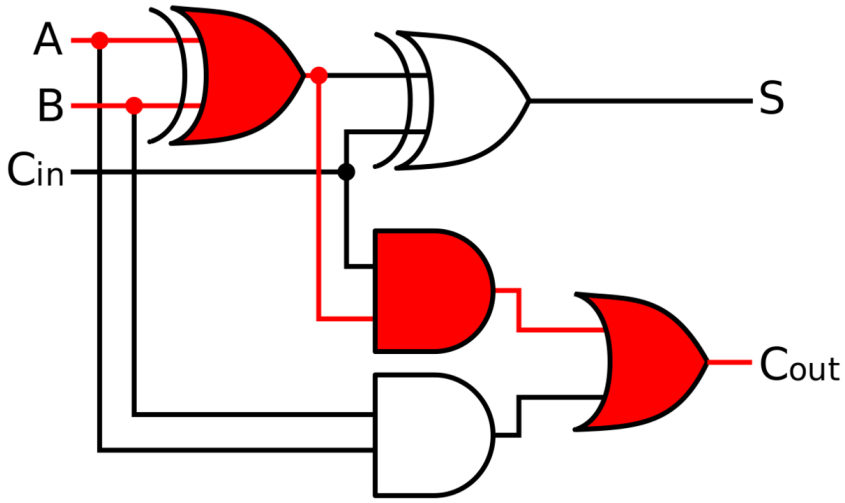

Figure 34

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
```

18.3 Serial Adder

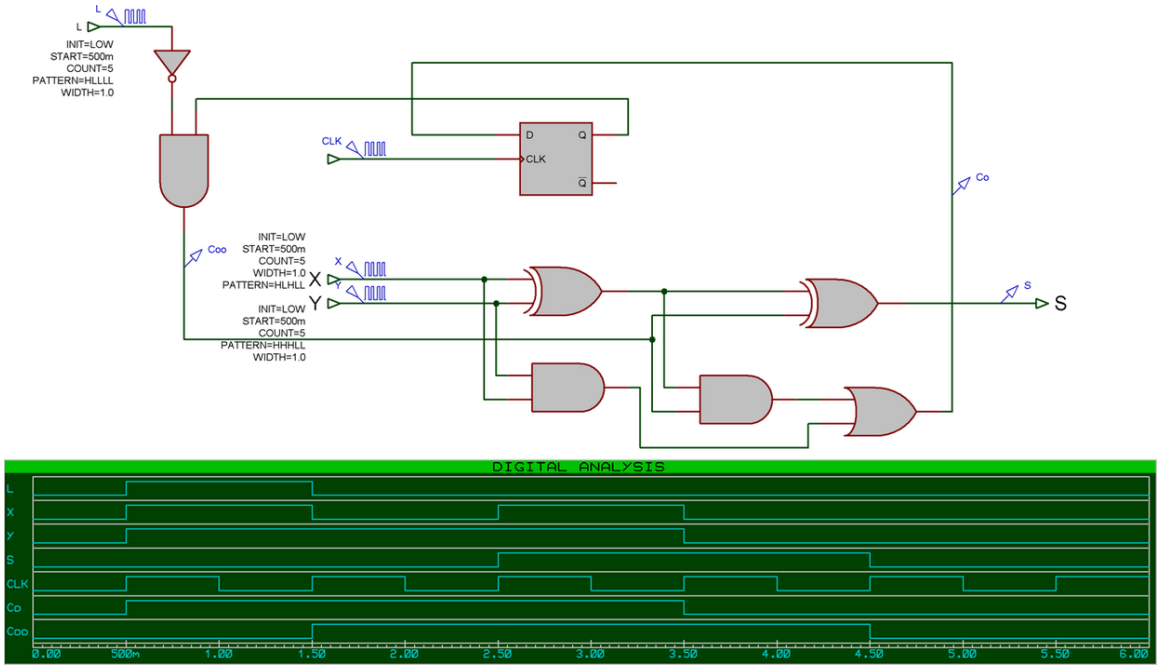

Figure 35

A serial adder is a kind of ../ALU/[1] 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.

18.4 Parallel Adder

A parallel adder is a kind of ../ALU/[2] 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).

1 Chapter 15 on page 73
2 Chapter 15 on page 73

18.4.1 Ripple Carry Adder

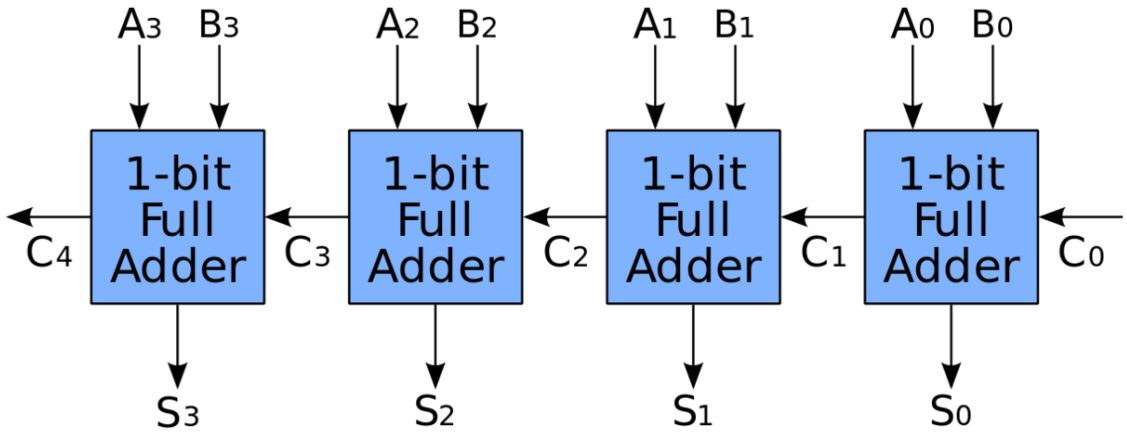

Figure 36

Numbers of more than 1 bit long require more then just a single full adder to manipulate using arithmetic and bitwise logic instructions. 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 fa2(a[1], b[1], c[0], c[1], s[1]);
full_adder fa3(a[2], b[2], c[1], c[2], s[2]);
full_adder fa4(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.[3] 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.

18.4.2 Carry Skip Adder

18.4.3 Carry Lookahead Adder

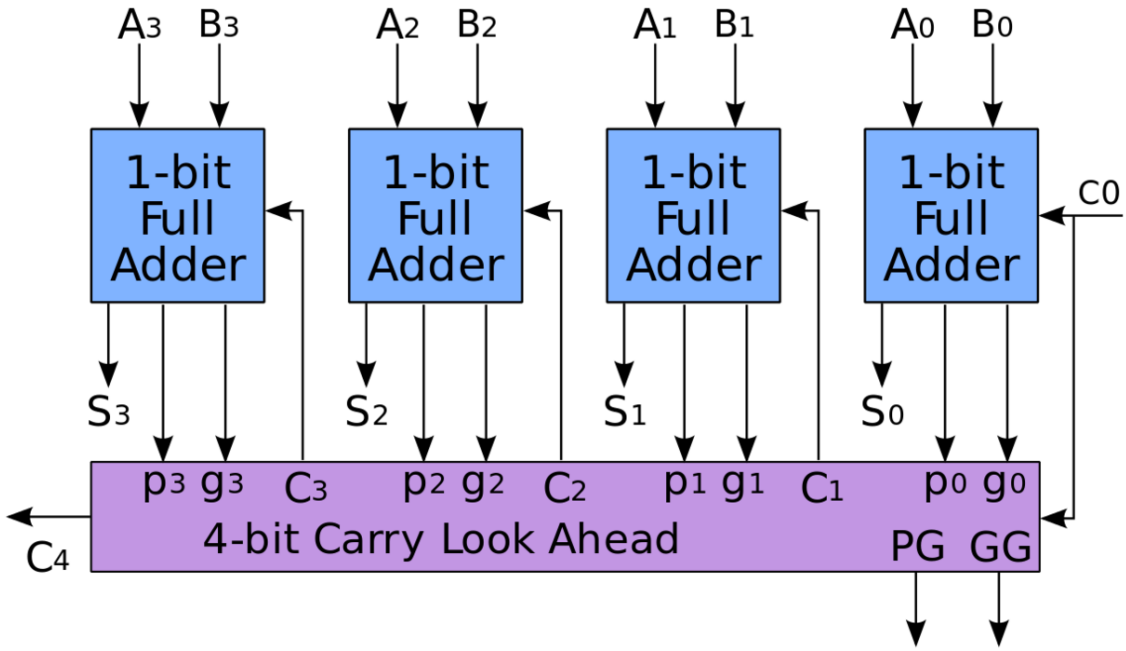

Figure 37

w:Carry look-ahead adder[4]

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.

3 "MuP21 Machine Forth": "Ripple Carry on + and +*" ^{http://www.ultratechnology.com/mfp21.htm}

4 http://en.wikipedia.org/wiki/Carry%20look-ahead%20adder

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:

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

18.4.4 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:

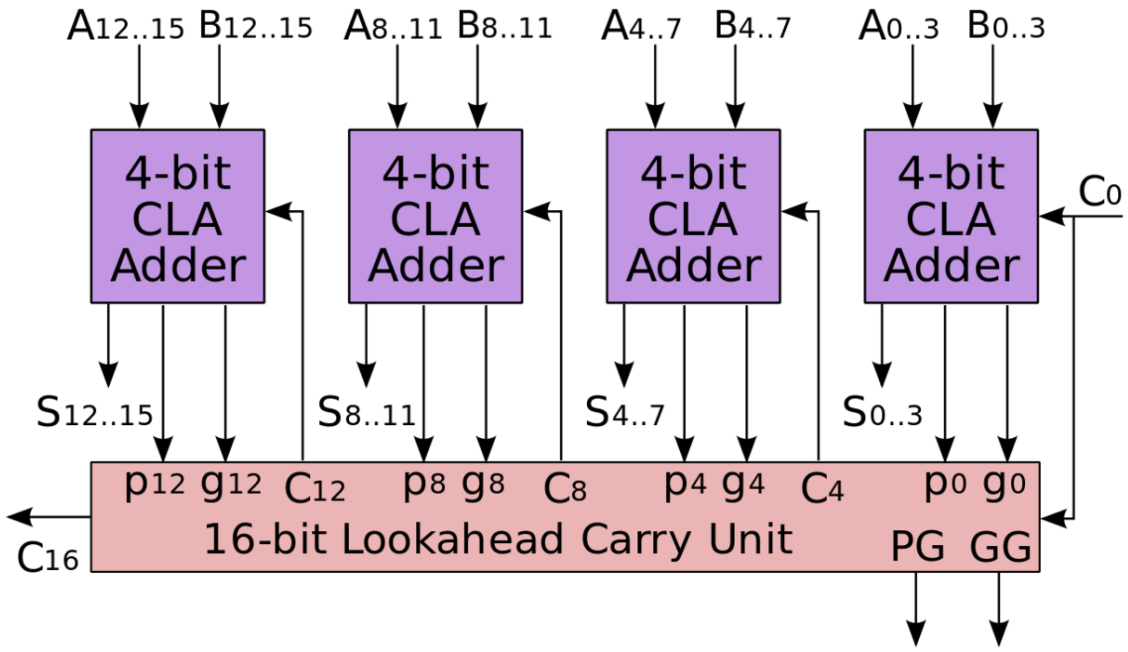

Figure 38 The 16-Bit carry lookahead unit is exactly the same as the 4-bit carry lookahead adder.

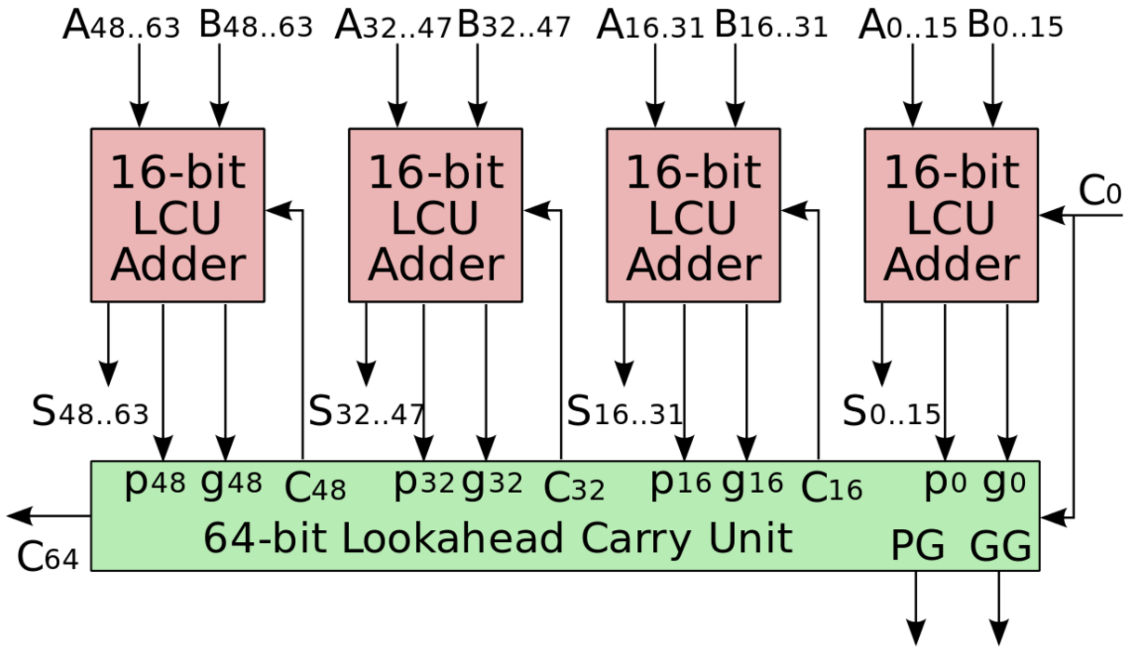

Figure 39 the 64-bit carry lookahead unit is exactly the same as the 4-bit and 16-bit units. This means that once we have designed one carry lookahead module, we can cascade it to any large size.

18.4.5 Generalized Cascading

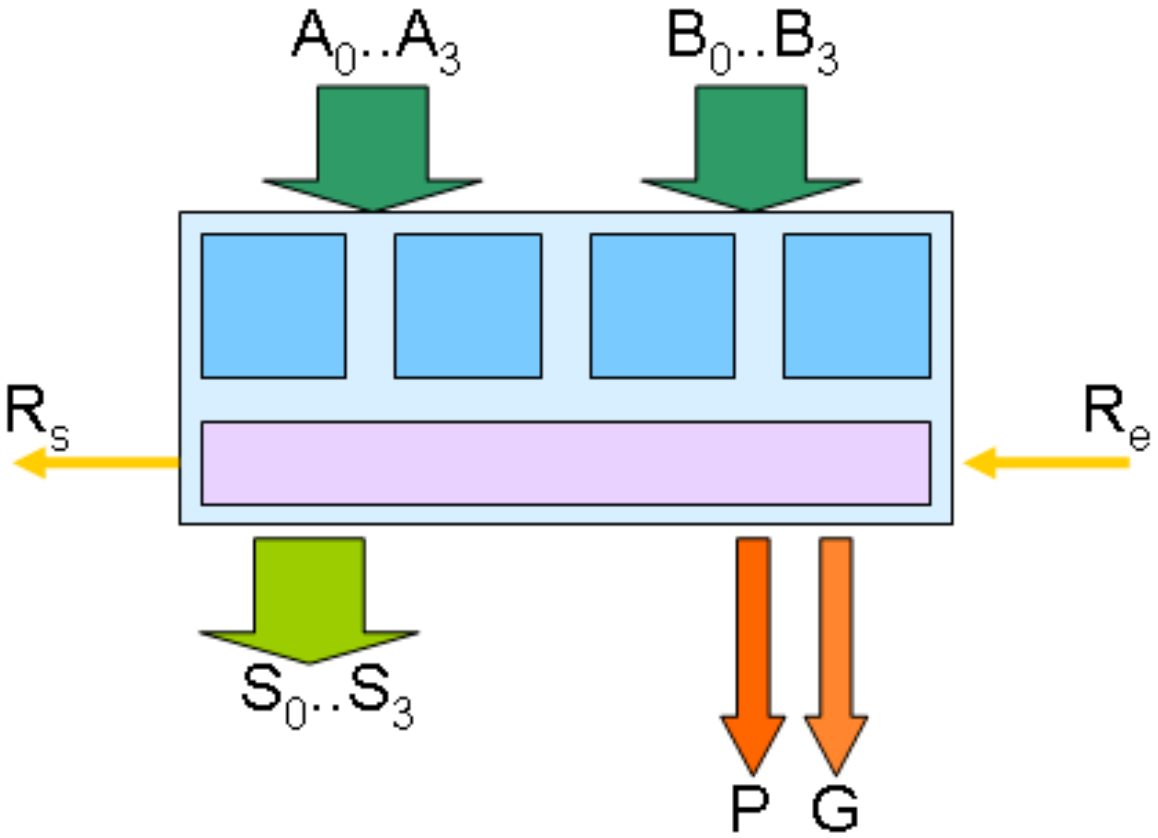

Figure 40 A generalized CLA block diagram. Each of the turquoise blocks represents a smaller CLA adder.

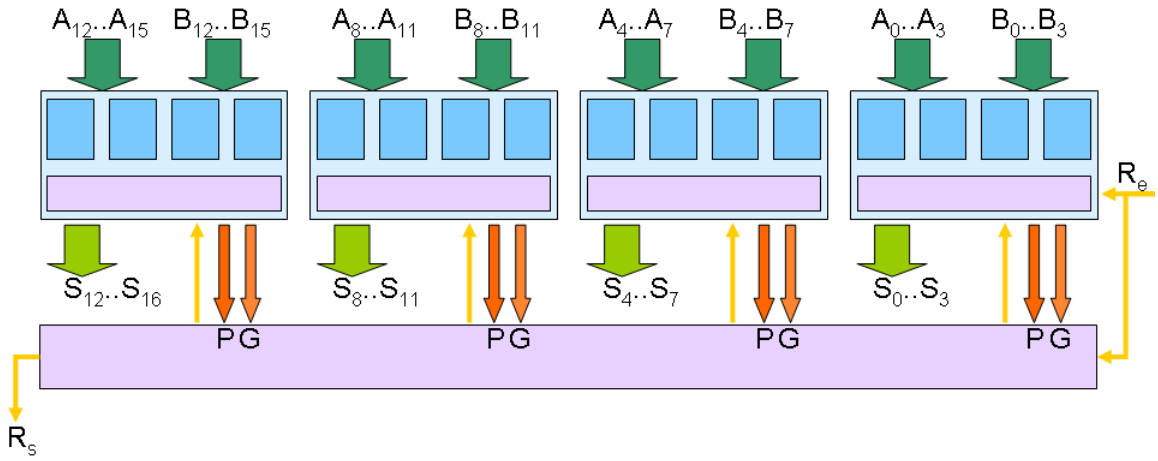

Figure 41 We can cascade the generalized CLA block above to form a larger CLA block. This larger block can then be cascaded into a larger CLA block using the same method.

18.5 Sources

19 Shift and Rotate Blocks

19.1 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.

19.2 Logical Shift

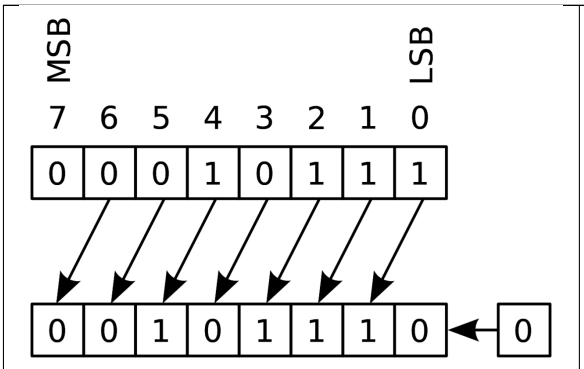

Figure 42
A left logical shift

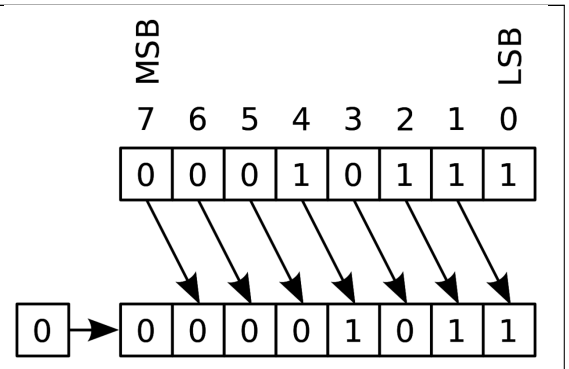

Figure 43
A right logical shift

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

19.3 Arithmetic shift

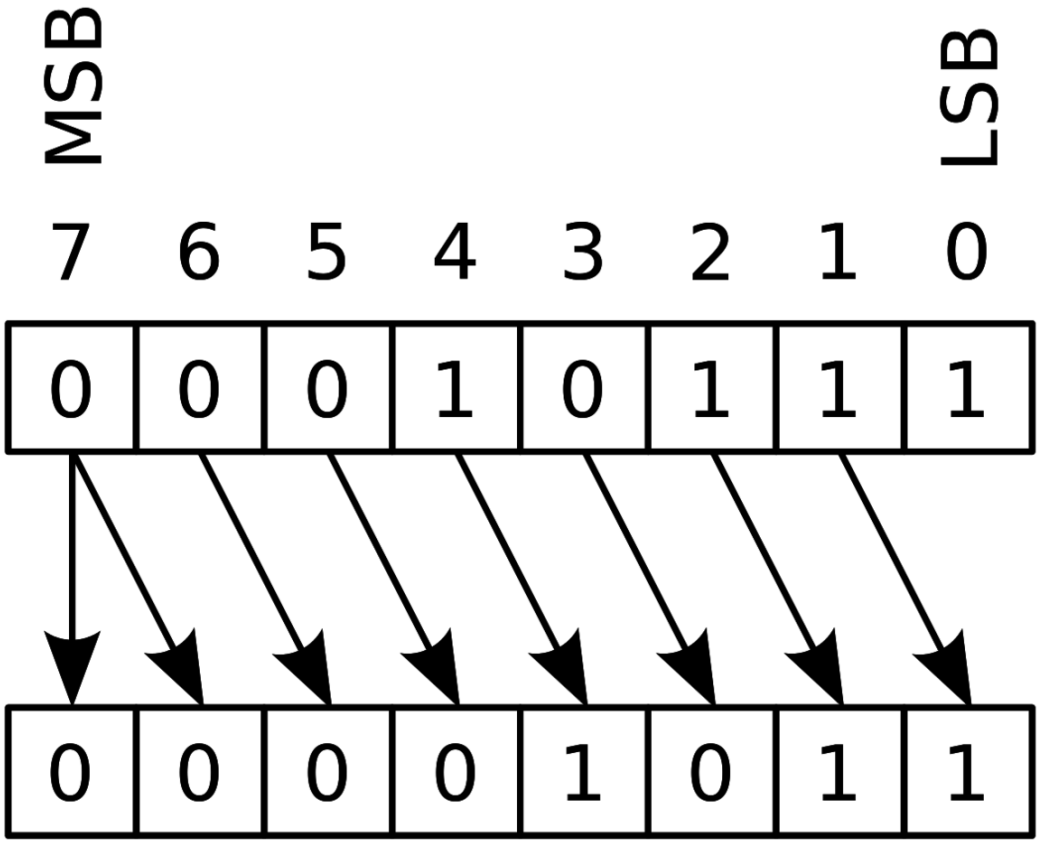

Figure 44
A right arithmetic shift

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.

19.4 Rotations

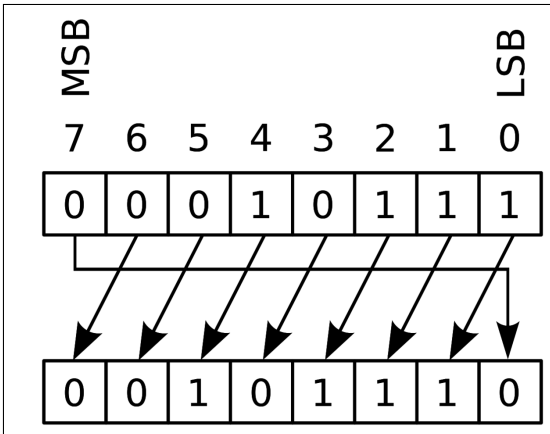

Figure 45
A left rotation

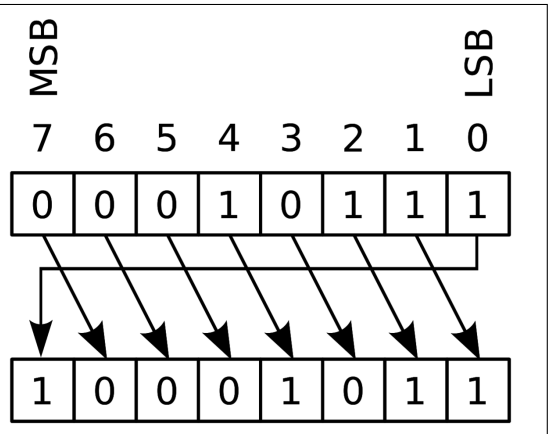

Figure 46
A right rotation

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

19.5 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.

19.6 Further reading

w:barrel shifter[1] 32-Bit Barrel Shifter Implementation Using 74-Series Integrated Circuits[2]

1 http://en.wikipedia.org/wiki/barrel%20shifter
2 http://ixeelectronics.com/Chipset/CPU/V3264/BarrelShifter32.html

20 Multiply and Divide Blocks

20.1 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.

20.2 Multiplication Algorithms

20.2.1 Booth's Algorithm

20.2.2 Cascaded Multiplication

20.2.3 Wallace tree

w: Wallace tree[1] w: Dadda multiplier[2]

The Wallace tree, a specialized structure for performing multiplication, has been called one of the most important advances in computing.[3]

A Wallace tree using many identical 3:2 compressors (aka full adders), such as the TI 74x275 chip, or the TI 74x183 chip, is one popular way to implement single-cycle multiplication. The datasheets for the TI 74x261 and 74x284 describe some practical details of implementing multiplication with a Wallace tree. The Dadda multiplier uses the same 3:2 compressors in a slightly more efficient arrangement.

---

1 http://en.wikipedia.org/wiki/%20Wallace%20tree

2 http://en.wikipedia.org/wiki/%20Dadda%20multiplier

3 DTACK Grounded, The Journal of Simple 68000/16081 Systems Issue # 29 - March 1984 ^{http://www.easy68k.com/paulrsm/dg/dg29.htm} p. 6.

20.3 Division Algorithm

20.4 Multiply and Accumulate

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)
```

20.4.1 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[4].

4 http://en.wikibooks.org/wiki/Floating%20Point

21 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.

21.1 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.

21.2 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.

21.3 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.

21.4 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.

21.5 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.

21.6 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.

22 Single Cycle Processors

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.

22.1 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.

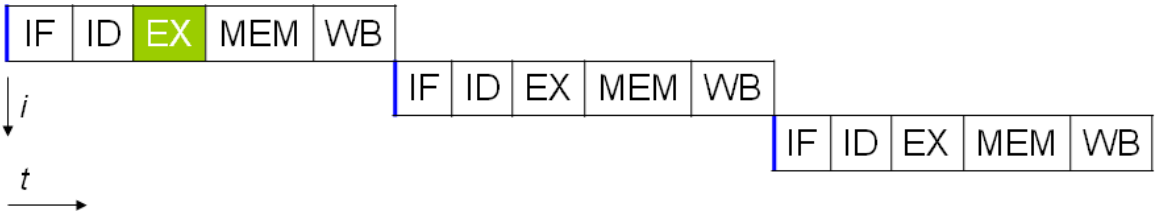

Figure 47

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.

22.2 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.

22.3 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.

23 Multi Cycle Processors

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.

23.1 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.

23.2 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.

24 Pipelined Processors

24.1 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:

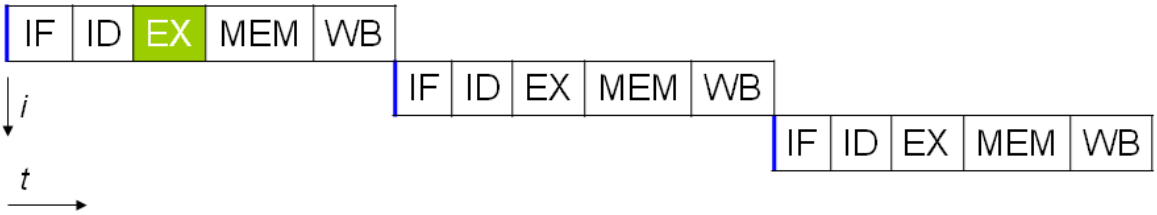

Figure 48

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.

24.1.1 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.

24.2 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:

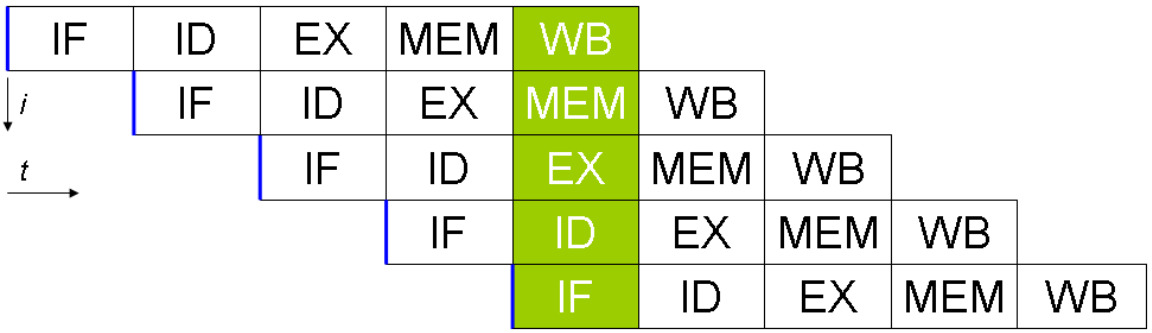

Figure 49

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.

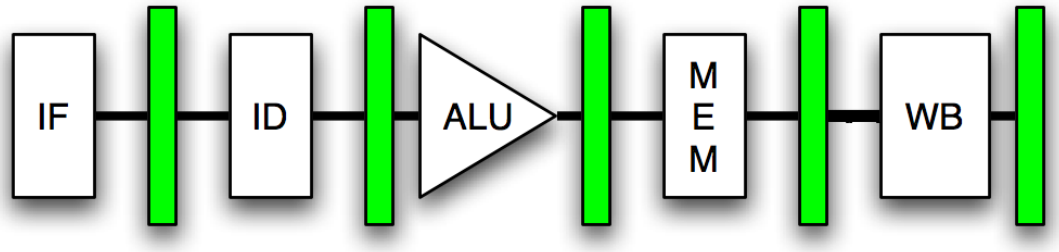

Figure 50

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:

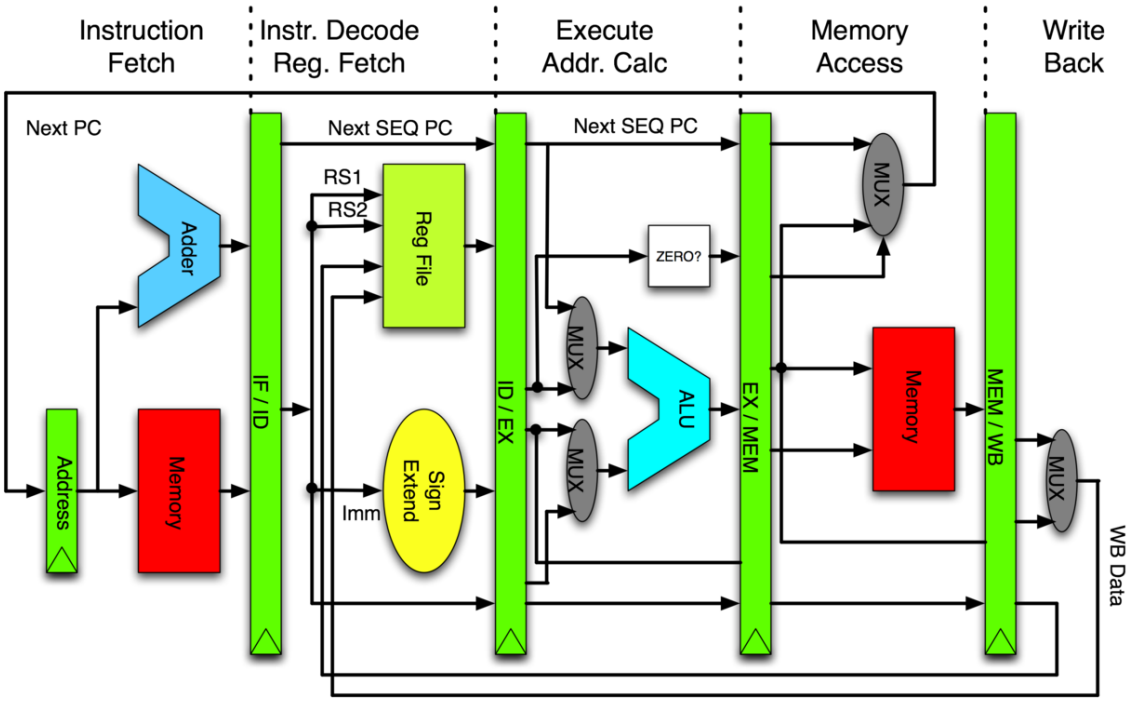

Figure 51

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.

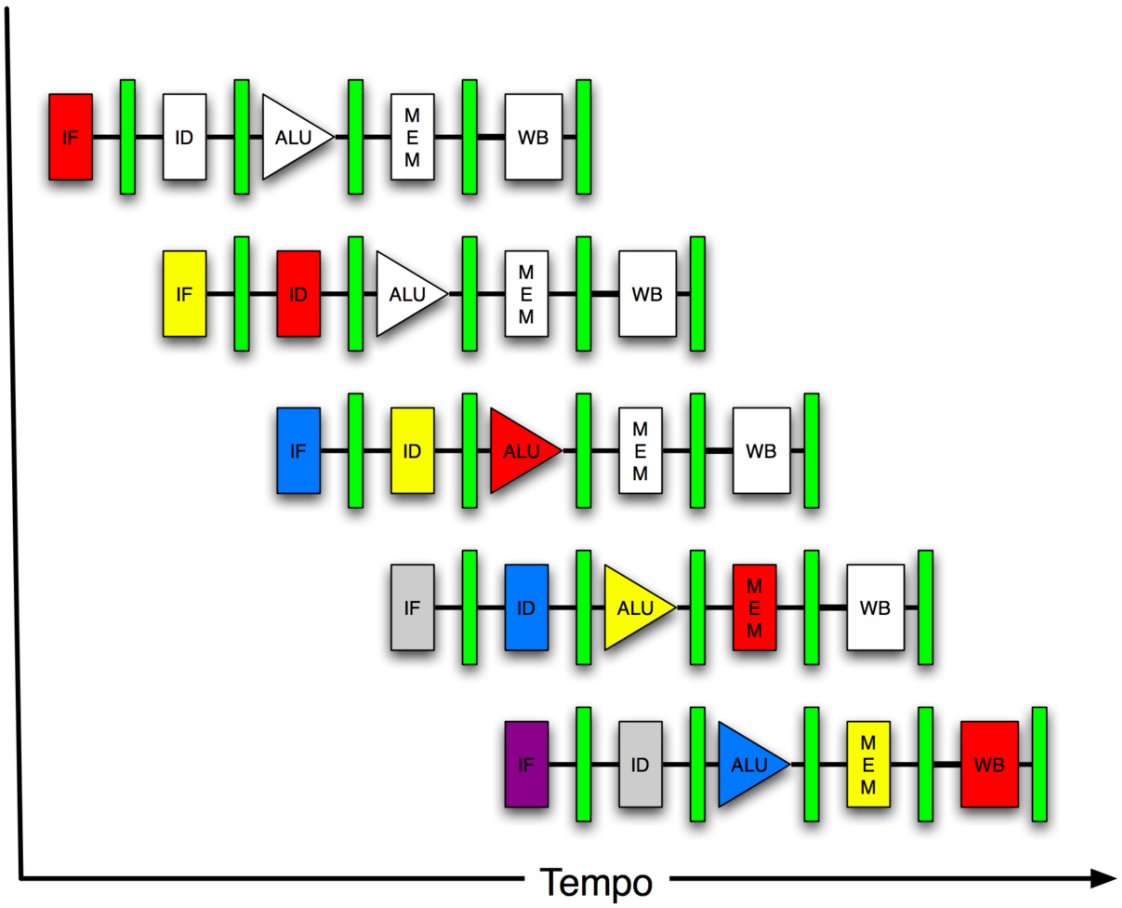

Figure 52

Pipelined processors generate the same results as a one-instruction-at-a-time processor does when running the same software -- they just generate those results much more quickly. People who build pipelined processors sometimes add special hardware -- operand forwarding; pipeline interlocks; etc. -- in order to get the same results "as if" each instruction is fetched, evaluated, and its results committed before the next instruction is fetched (non-overlapped) -- even though pipelined processors actually overlap instructions.

The **throughput** of a processor is the number of instructions that complete in a span of time. Many processors are designed to have a typical throughput of one instruction per clock cycle, even though any one particular instruction requires many cycles -- one cycle per pipeline stage -- from the time it is fetched to the time it completes.

24.3 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 than 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.

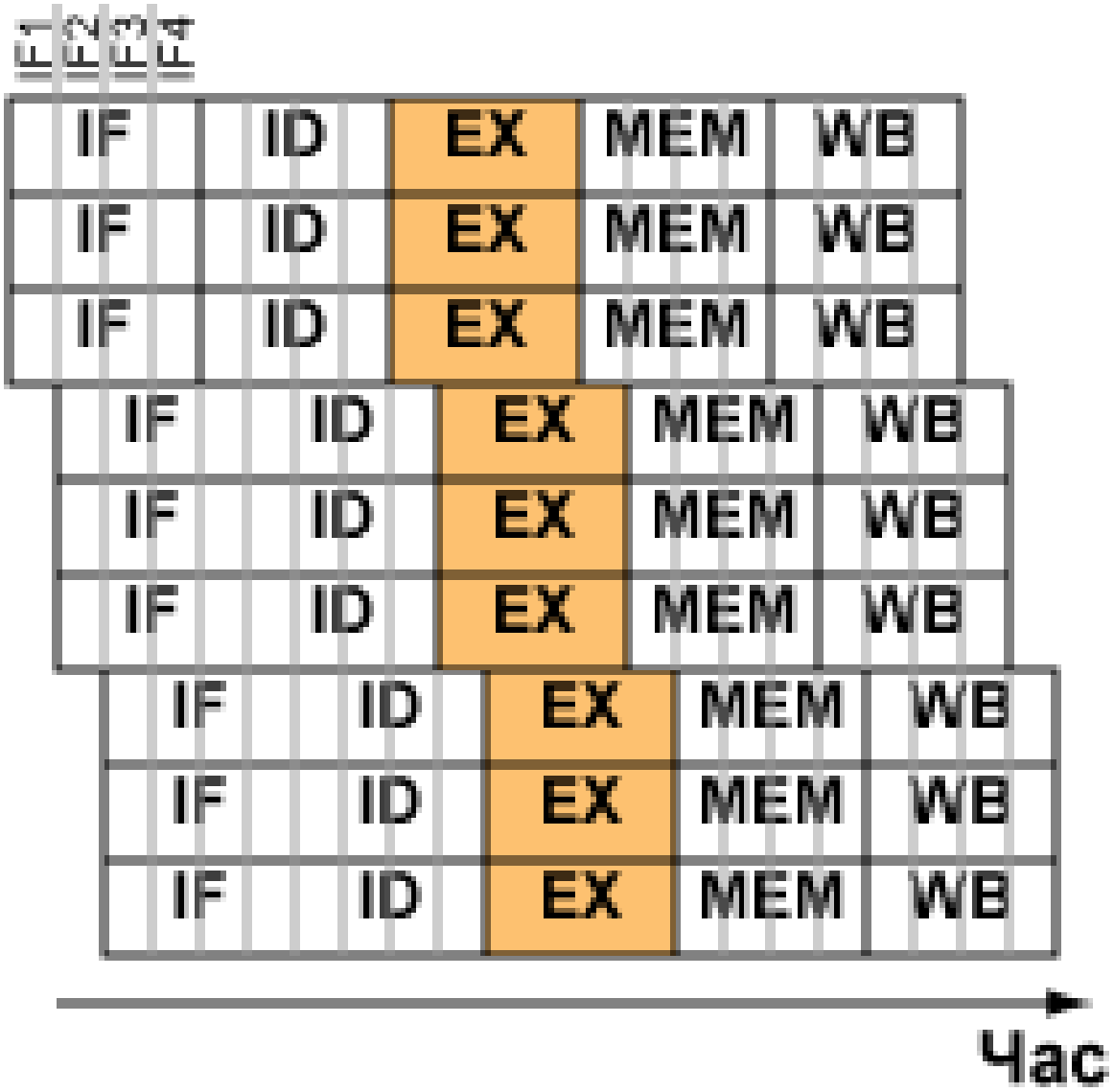

Figure 53

24.4 Resources

w: instruction pipeline[1]

1 http://en.wikipedia.org/wiki/%20instruction%20pipeline

Commons:Pipeline (computer)[2]

- Jim Plusquellic. "CMSC 611: Advanced Computer Architecture". "Introduction to Pipelining"[3].
- Jon "Hannibal" Stokes. "Pipelining: An Overview (Part I)"[4].
- Jon Stokes. "Pipelining: An Overview (Part II)"[5].

2 http://en.commons.org/wiki/Pipeline%20%28computer%29
3 http://www.ece.unm.edu/~jimp/611/slides/chap3_1.html
4 http://archive.arstechnica.com/paedia/p/pipelining-1/m-pipelining-1-1.html
5 http://arstechnica.com/features/2004/09/pipelining-2/

25 Superscalar Processors

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.

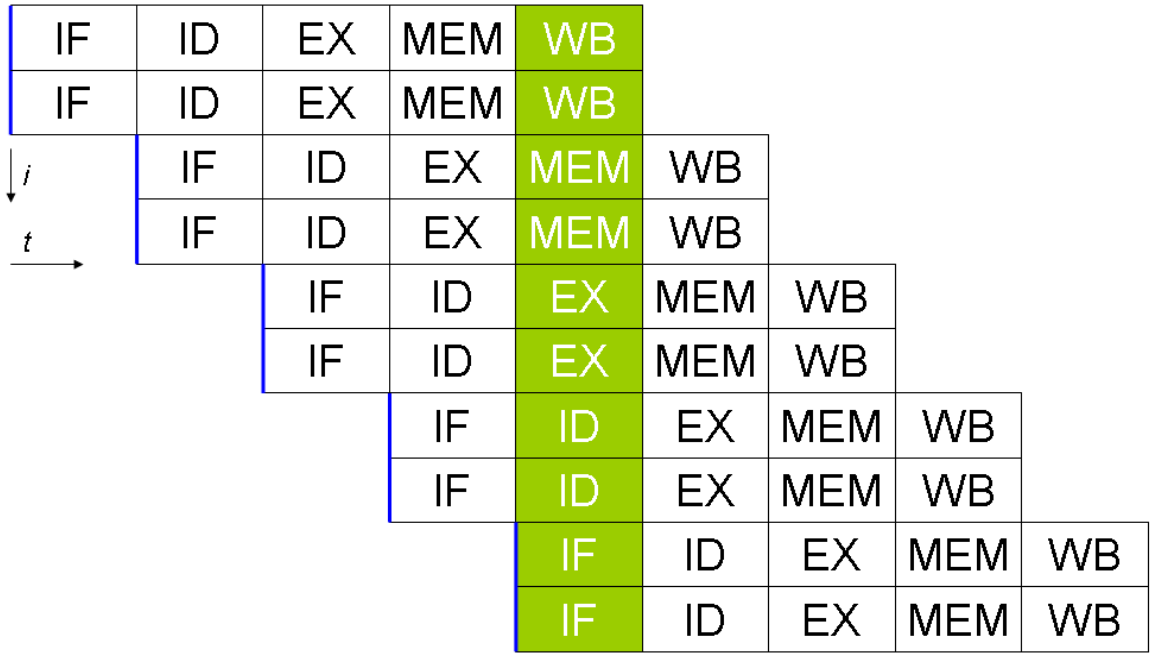

Figure 54

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.

26 VLIW Processors

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

26.1 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!

26.2 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.

27 Vector Processors

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.

27.1 Parallel Execution

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

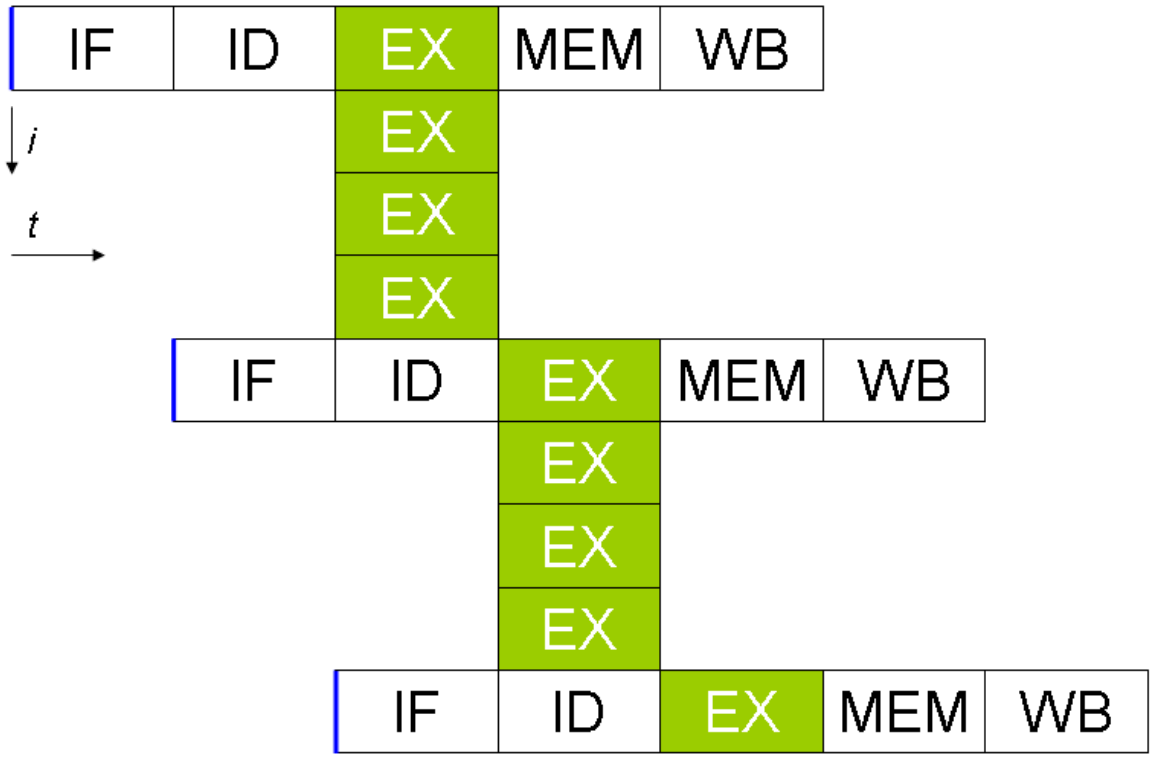

Figure 55

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

27.2 Non-Parallel Execution

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

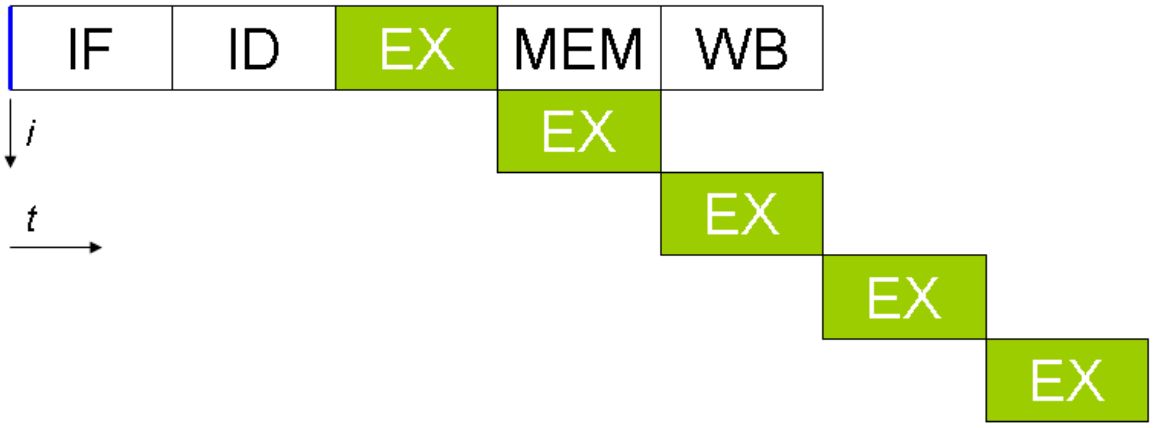

Figure 56

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.

28 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.

28.1 Symmetric Multi-core

A symmetric multi-core 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 multi-core 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 independently 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 multi-core 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.

28.2 Asymmetric Multi-core

An asymmetric multi-core 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 multi-core 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 multi-core 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].

28.3 Symmetric Multicore

Figure 57 The Intel Core 2 Duo

A symmetric multi-core 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 multi-core 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.

28.4 Asymmetric Multi-core

In an asymmetric multi-core processor, the chip has multiple cores on-board, but the cores might be different designs. Each core will have different capabilities.

28.4.1 Example: IBM Cell Processor

An example of an asymmetric multi-core processor is the IBM Cell processor.

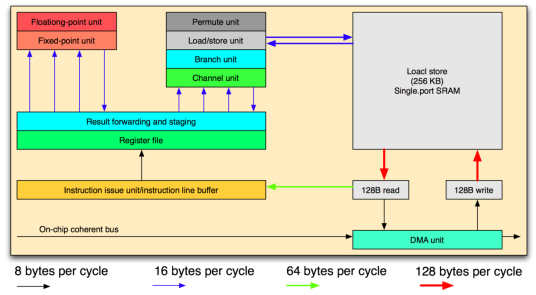

Figure 58

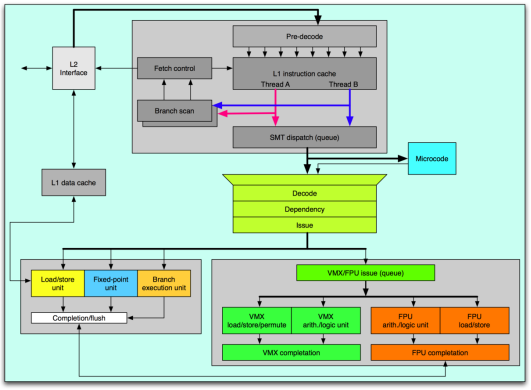

Figure 59

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:

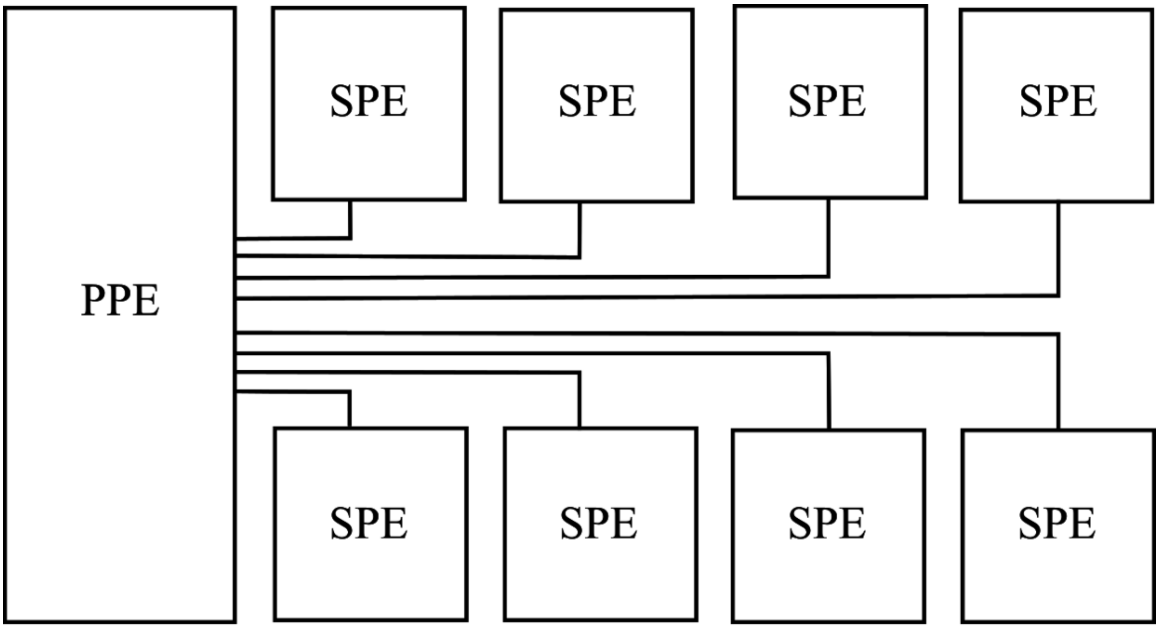

Figure 60

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.

28.5 further reading

w:Multi-core (computing)[2]

2 http://en.wikipedia.org/wiki/Multi-core%20%28computing%29

29 Exceptions

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.

30 Interrupts

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:

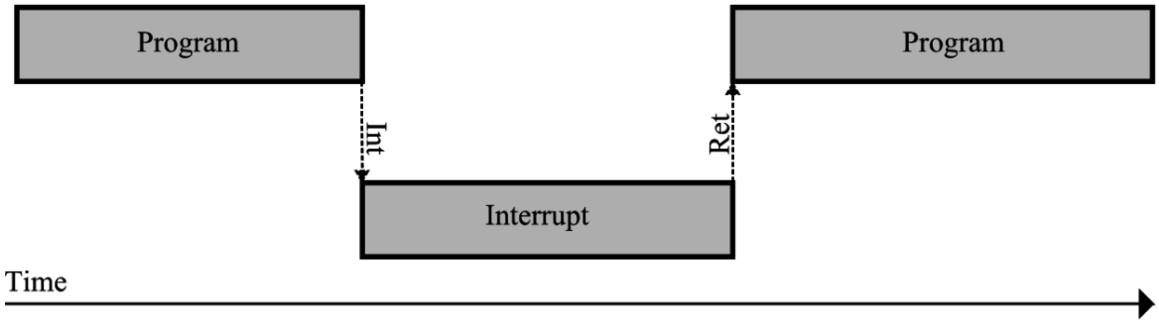

Figure 61

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

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”.

30.1 Further Reading

w:interrupt[1] w:interrupt storm[2] w:interrupt vector[3]

- Operating System Design/Processes/Interrupt[4]

1 http://en.wikipedia.org/wiki/interrupt
2 http://en.wikipedia.org/wiki/interrupt%20storm
3 http://en.wikipedia.org/wiki/interrupt%20vector
4 http://en.wikibooks.org/wiki/Operating%20System%20Design%2FProcesses%2FInterrupt

31 Hazards

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.

31.1 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.

31.1.1 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.

31.2 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.

31.3 Structural Hazards

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

31.4 Fixing Hazards

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

31.4.1 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.

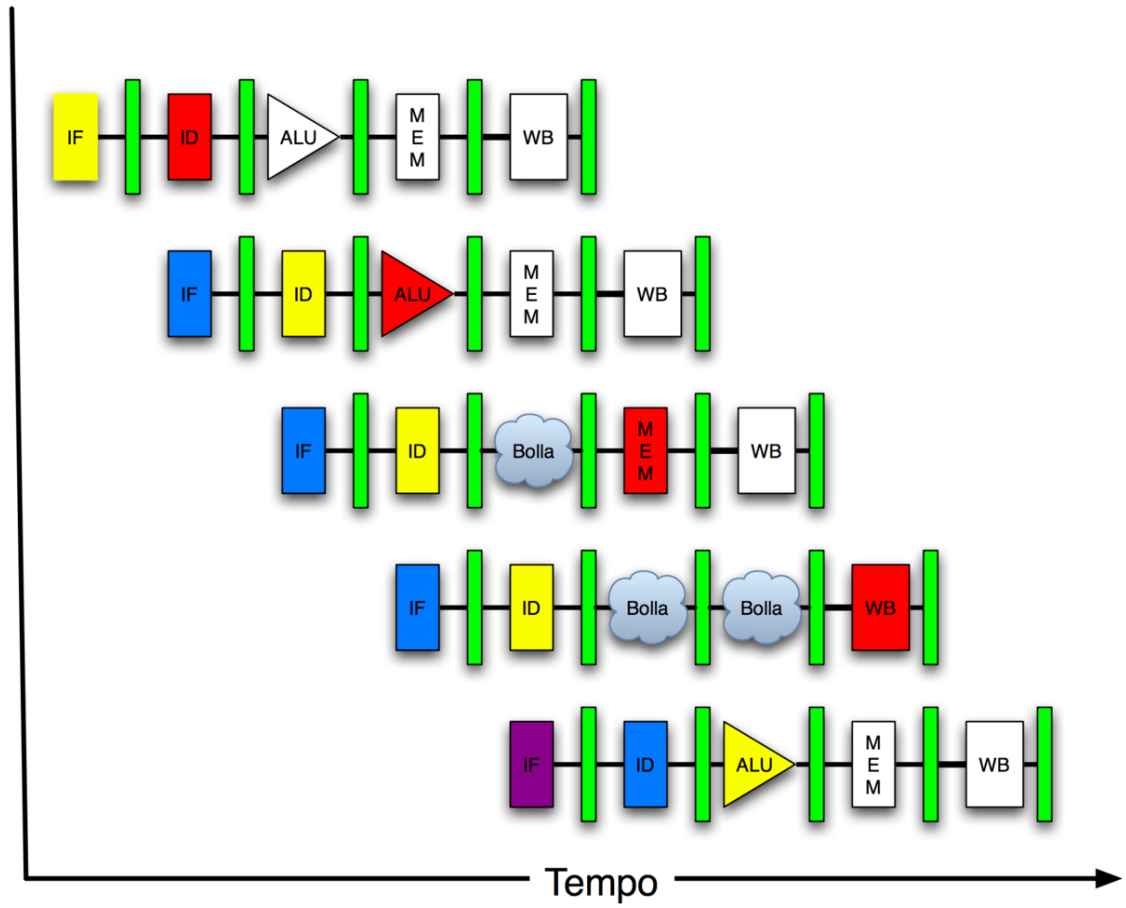

Figure 62

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.

31.4.2 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.

31.4.3 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).

31.4.4 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.

31.4.5 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.

31.4.6 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.

31.4.7 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 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.

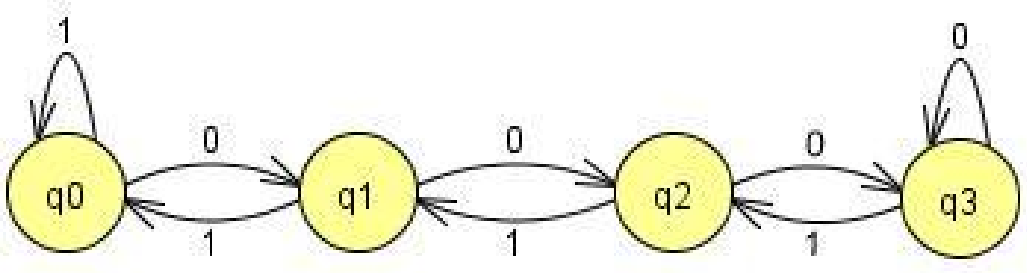

Figure 63 A 2-bit branch predictor with 4 stages.

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.

32 Performance Metrics

33 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.

33.1 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[1] 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.

33.1.1 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).

1 Chapter 34 on page 149

33.1.2 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.

33.1.3 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.

33.2 Processor Time

33.3 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.

33.4 Latency

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

33.5 MIPS/mW

[2]

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.

2 "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?"

33.6 Further reading

34 Benchmarking

34.1 Benchmarks

34.2 Common Benchmarks

- block move
- Eratosthenes sieve
- matrix multiply

- MINC/Benchmarks[1]
- "AN910: ST7 and ST9 performance benchmarking"[2] 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"[3] describes some very frequently used, very short code fragments.
- Wikipedia: benchmark (computing)[4]
- Wikipedia: EEMBC[5] Embedded Microprocessor Benchmark Consortium
- MIPS/Watt benchmarks; "Philips Challenges 8-bit MCUs"[6]; "Innovative Techniques for Extremely Low Power Consumption with 8-bit Microcontrollers"[7]
- 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 `http://www.nabble.com/Re:-Intellasys-question-for-Jeff-Fox-p17450680.html`

1 `http://en.wikibooks.org/wiki/MINC%2FBenchmarks`
2 `http://www.st.com/stonline/books/pdf/docs/5039.pdf`
3 `http://www.bradrodriguez.com/papers/moving2.htm`
4 `http://en.wikipedia.org/wiki/%20benchmark%20%28computing%29`
5 `http://en.wikipedia.org/wiki/%20EEMBC`
6 `http://www.standardics.nxp.com/support/documents/microcontrollers/pdf/article.challenge.8-bit.mcu.pdf`
7 `http://atmel.com/dyn/resources/prod_documents/doc7903.pdf`

34.3 Benchmark Problems

34.4 Further reading

- Cyberbotics' Robot Curriculum/Cognitive Benchmarks[8]

8 http://en.wikibooks.org/wiki/Cyberbotics%27%20Robot%20Curriculum%2FCognitive%20Benchmarks

35 Optimizations

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.

36 Multi-Core Systems

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.

36.1 Symmetric Multi-core

A symmetric multi-core 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 multi-core 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 independently 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 multi-core 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.

36.2 Asymmetric Multi-core

An asymmetric multi-core 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 multi-core 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 multi-core 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].

36.3 Symmetric Multicore

Figure 64 The Intel Core 2 Duo

A symmetric multi-core 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 multi-core 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.

36.4 Asymmetric Multi-core

In an asymmetric multi-core processor, the chip has multiple cores on-board, but the cores might be different designs. Each core will have different capabilities.

36.4.1 Example: IBM Cell Processor

An example of an asymmetric multi-core processor is the IBM Cell processor.

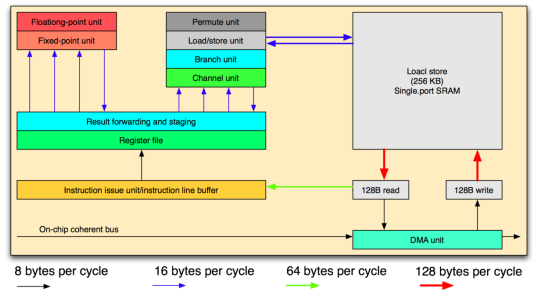

Figure 65

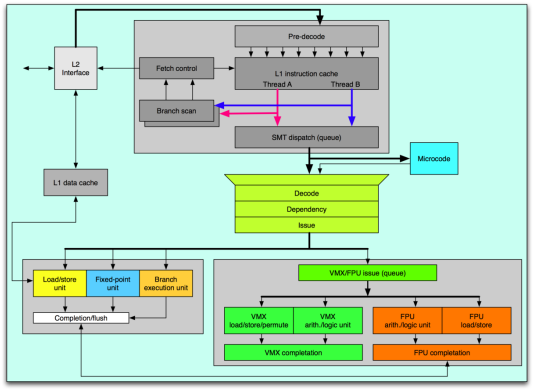

Figure 66

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:

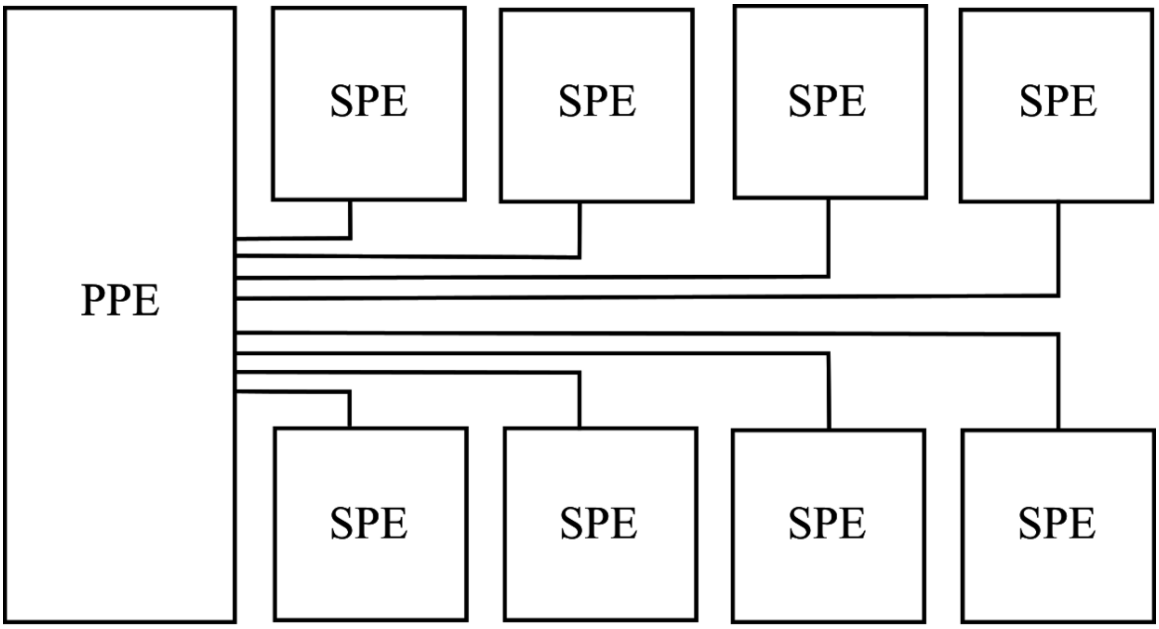

Figure 67

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.

36.5 further reading

w:Multi-core (computing)[2]

2 http://en.wikipedia.org/wiki/Multi-core%20%28computing%29

37 Memory-Level Parallelism

37.1 Memory-Level Parallelism

w:Memory level parallelism[1]

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.

1 http://en.wikipedia.org/wiki/Memory%20level%20parallelism

38 Out Of Order Execution

w:Out-of-order execution[1]

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.

38.1 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.

38.2 Example: Intel Hyperthreading

w:Hyper-threading[2]

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.

1 http://en.wikipedia.org/wiki/Out-of-order%20execution
2 http://en.wikipedia.org/wiki/Hyper-threading

39 Assembler

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.

40 Simulator

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* .

41 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 built in 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.

41.1 Further reading

- Compiler Construction[1] discusses how to write a compiler from scratch

1 http://en.wikibooks.org/wiki/Compiler%20Construction

42 FPGA

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

a http://en.wikibooks.org/wiki/Programmable%20Logic

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[1], lists details on how to get them.

1 Chapter 50 on page 205

43 Photolithography

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.

43.1 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** .

43.2 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)[1] (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.

43.3 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? ...

1 `http://en.wikipedia.org/wiki/doping%20%28semiconductor%29`

43.4 further reading

w:semiconductor fabrication plant[2] w:semiconductor device fabrication[3] w:Integrated circuit packaging[4] w:MOSIS[5]

- MOSIS (Metal Oxide Semiconductor Implementation Service[6]) is probably the oldest (1981) integrated circuit (IC) foundry service. Many VLSI students have sent their chips to MOSIS for fabrication.

2 http://en.wikipedia.org/wiki/semiconductor%20fabrication%20plant
3 http://en.wikipedia.org/wiki/semiconductor%20device%20fabrication
4 http://en.wikipedia.org/wiki/Integrated%20circuit%20packaging
5 http://en.wikipedia.org/wiki/MOSIS
6 http://www.mosis.com/

44 Sockets and interfacing

44.1 Form Factors

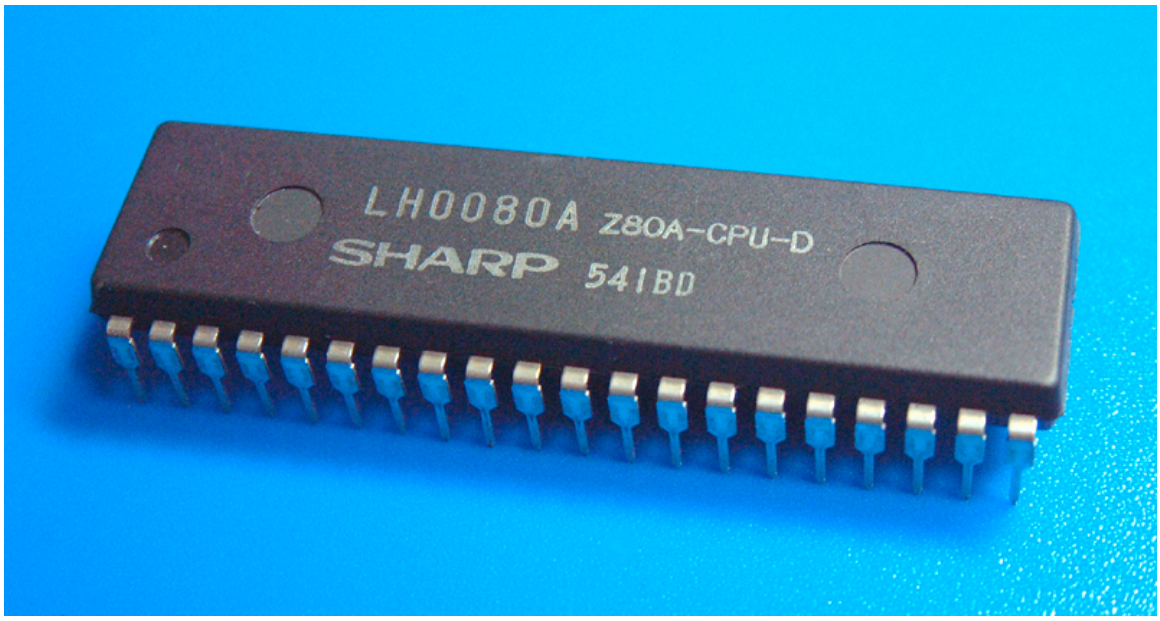

Figure 68 A Sharp Z80 microprocessor, with a Dual-Inline Package (DIP) with 40 pins.

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 Athlon 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** .

44.2 Connectors

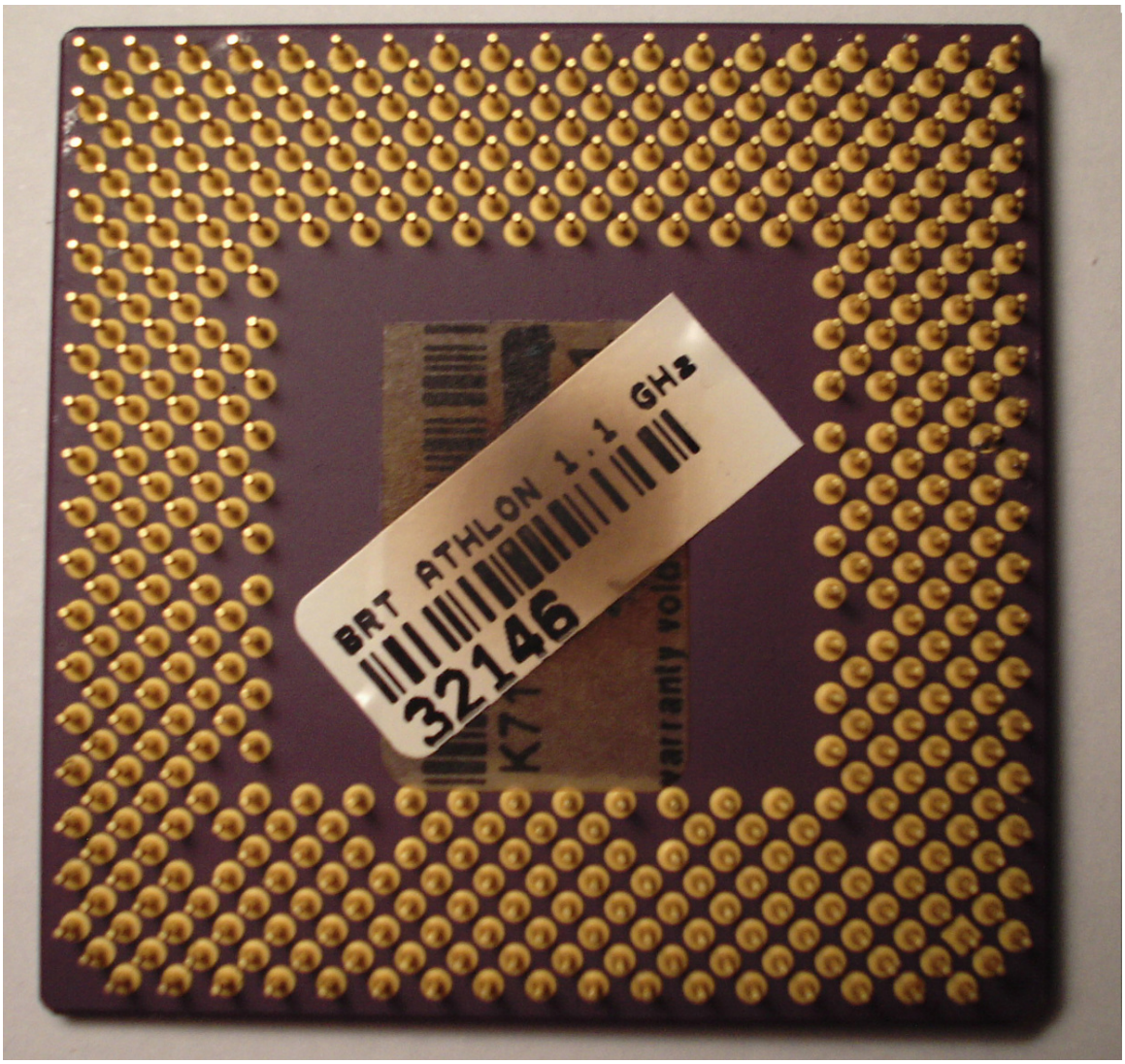

Figure 69 The underside of an AMD Athlon processor, showing the connector pins.

44.3 Sockets

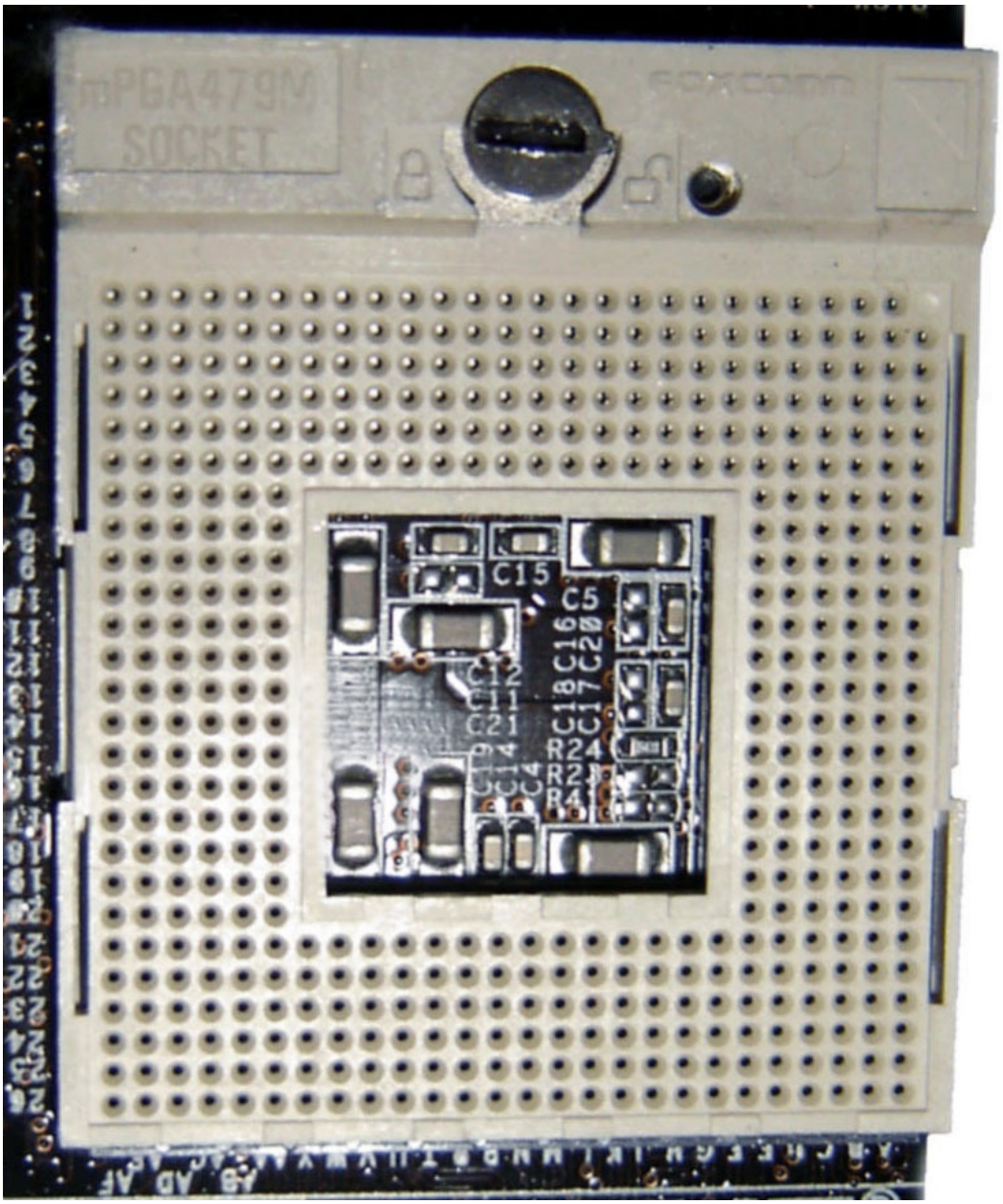

Figure 70 A Socket 479, one of many different and incompatible socket types.

45 Microcodes

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.

w:control store[1] w:microcode[2] The most common way to implement memory-memory architecture CPUs (even with single-chip microprocessors, not just wire-wrapped machines[3]) uses a small "control store" ROM. The output data bits of the control store are latched in the microinstruction register[4][5][6][7][8] (reminiscent of the way instructions fetched from RAM are latched in the instruction register). The clock signal determining the cycle time of the system primarily clocks the microinstruction register. The bits stored in the microinstruction register directly control everything that goes on in the CPU. (In some processors, the microinstruction register is the only thing connected to the clock signal. Later we will discuss "pipelining", a technique involving pipeline registers connected to the clock signal).

Some of the bits in the microinstruction 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.

---

1 http://en.wikipedia.org/wiki/control%20store

2 http://en.wikipedia.org/wiki/microcode

3 http://en.wikibooks.org/wiki/Microprocessor%20Design%2FWire%20Wrap%20

4 US Patent 5050073. "Microinstruction execution system for reducing execution time for calculating microinstruction" ^{http://www.google.com/patents/US5050073} . 1987.

5 Jonathan G. Campbell. "The Central Processing Unit (CPU)" ^{http://www.johnloomis.org/ece314/notes/carch/node6.html} 2000.

6 Patrick R. Schaumont. "A Practical Introduction to Hardware/Software Codesign" ^{http://books.google.com/books?id=ngENR5O6fusC&pg=PA157&lpg=PA157&dq=%22microinstruction+register%22&source=bl&ots=Zlow6_H8fU&sig=_YOLw9liHRxU8h2nWaxG45C9Qr4&hl=en&sa=X&ei=4WOEUu6ZFZOWyAHmlYDABQ&ved=0CCGOQ6AEwCA#v=onepage&q=%22microinstruction%20register%22&f=false} . 2010.

7 Govindarajalu. "Computer Architecture and Organization: Design Principles and Applications" ^{http://books.google.com/books?id=YT74AkSrj4sC&pg=PA237&lpg=PA237&dq=%22microinstruction+register%22&source=bl&ots=4rrwlDKNZc&sig=ir8J4PAyxoV3GPtYsr-k0LPT6BA&hl=en&sa=X&ei=Gm4EUqKZOoK6yQHJpoGACQ&ved=0CEwQ6AEwBTgK#v=onepage&q=%22microinstruction%20register%22&f=false} . 2004.

8 B. Govindarajalu. "Computer Architecture and Organization, 2E" ^{http://books.google.com/books?id=zzGoVXQ0GzsC&pg=PA278&lpg=PA278&dq=%22microinstruction+register%22&source=bl&ots=wjoc4xyWjS&sig=TaexyoCUfv6b6WL4ZmZw9Z_lqUY&hl=en&sa=X&ei=Gm4EUqKZOoK6yQHJpoGACQ&ved=0CE8Q6AEwBjgK#v=onepage&q=%22microinstruction%20register%22&f=false} 2010.

Some CPUs, such as the ECOMIPS[9], the Intel Core 2 and the Intel Xeon,[10] 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.

45.1 Further Reading

w: microsequencer [11]

- "Viktor's Amazing 4-bit Processor"[12] 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 uses transistors instead of diodes in a big AND-OR PLA (programmable logic array) matrix to implement the microcode.

45.2 References

9 "ECOMIPS: An Economic MIPS CPU Design on FPGA" ^{http://www.lixizhi.net/download/xizhil_ecomips.pdf} by Xizhi Li and Tiecai Li

10 Wikipedia: microcode#Writable_control_stores ^{http://en.wikipedia.org/wiki/%20microcode%23Writable_control_stores}

11 http://en.wikipedia.org/wiki/%20microsequencer%20

12 http://www.vttoth.com/vicproc.htm

46 Register Renaming

47 Cache

47.1 Cache

w:Cache[1]

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.

The cache memory performance is the most significant factor in achieving high processor performance.[2]

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 **principle of locality** .

In this chapter, we will discuss several possible cache arrangements, in increasing order of complexity:

- No cache, single-CPU, physical addressing
- Single cache, single-CPU, physical addressing
- Cache hierarchy: L1, L2, L3, etc.
- cache replacement policies: associativity, random replacement, LRU, etc.
- Split cache: I-cache and D-cache, on top of a unified cache hierarchy
- caching with multiple CPUs
- cache hardware that supports virtual memory addressing
- the TLB as a kind of cache
- how single-address-space virtual memory addressing interacts with cache hardware
- how per-process virtual memory addressing interacts with cache hardware

1 http://en.wikipedia.org/wiki/Cache

2 Alan Jay Smith. "Design of CPU Cache Memories". Proc. IEEE TENCON, 1987. http://www.eecs.berkeley.edu/Pubs/TechRpts/1987/5288.html http://www.eecs.berkeley.edu/Pubs/TechRpts/1987/CSD-87-357.pdf

47.2 No cache

Most processors today, such as the processors inside standard keyboards and mice, don't have any cache. Many historically important computers, such as Cray supercomputers, don't have any cache.[3] The vast majority of software neither knows nor cares about the specific details of the cache, or if there is even a cache at all.

Processors without a cache are usually limited in performance by the main memory access time. Without a cache, the processor fetches each instruction, one at a time, from main memory, and every LOAD or STORE goes to main memory before executing the next instruction.

One way to improve performance is to substitute faster main memory. Alas, that usually has a financial limit: hardly anyone is willing to pay a penny a bit for a gigabyte of really fast main memory.

Even if money is no object, eventually one reaches physical limits to main memory access time. Even with the fastest possible memory money can buy, the memory access time for a unified 1 gigabyte main memory is limited by the time it takes a signal to get from the CPU to the most distant part of the memory and back.

47.3 Single cache

Using exactly the same technology, it takes less time for a signal to traverse a small block of memory than a large block of memory.

The performance of a processor with a cache is no longer limited by the main memory access time. The performance of a processor with a cache is usually limited in performance by the (much faster) cache memory access time: if the cache access time of a processor could be decreased, the processor would have higher performance. [4] However, cache memory is generally much easier to speed up than main memory: really fast memory is much more affordable when we only buy small amounts of it. If it will improve the performance of a system significantly, lots of people are willing to pay a penny a bit for a kilobyte of really fast cache memory.

47.3.1 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

3 Alan Jay Smith. "Design of CPU Cache Memories". Proc. IEEE TENCON, 1987. `http://www.eecs.berkeley.edu/Pubs/TechRpts/1987/5288.html` `http://www.eecs.berkeley.edu/Pubs/TechRpts/1987/CSD-87-357.pdf`

4 Alan Jay Smith. "Design of CPU Cache Memories". Proc. IEEE TENCON, 1987. `http://www.eecs.berkeley.edu/Pubs/TechRpts/1987/5288.html` `http://www.eecs.berkeley.edu/Pubs/TechRpts/1987/CSD-87-357.pdf`

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. It is a good idea to keep recently used items in the cache, and not over-write data that has been recently used.

47.4 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.

Measurements of "the hit ratio" are typically performed on benchmark[5] applications. The actual hit ratio varies widely from one application to another. In particular, video and audio streaming applications often have a hit ratio close to zero, because each bit of data in the stream is read once for the first time (a compulsory miss), used, and then never read or written again. Even worse, many cache algorithms (in particular, LRU) allow this streaming data fill the cache, pushing out of the cache information that will be used again soon (cache pollution).[6]

47.5 Cache performance

A processor with a cache first looks in the cache for data (or instructions). On a miss, the processor then fetches the data (or instructions) from main memory. On a miss, this process takes *longer* than an equivalent processor without a cache.

There are three ways a cache gives better net performance than a processor without a cache:

- A hit (read from the cache) is faster than the time it takes a processor without a cache to fetch from main memory. The trick is to design the cache so we get hits often enough that their increase in performance more than makes up for the loss in performance on the occasional miss. (This requires a cache that is faster than main memory).
- Multiprocessor computers with a shared main memory often have a bottleneck accessing main memory. When a local cache succeeds in satisfying memory operations without going all the way to main memory, main memory bandwidth is freed up for the other

5 http://en.wikibooks.org/wiki/benchmark%20%28computing%29

6 Paul V. Bolotoff. "Functional Principles of Cache Memory" ^{http://alasir.com/articles/cache_principles/} . 2007.

processors, and the local processor doesn't need to wait for the other processors to finish their memory operations.[7]
- Many systems are designed so the processor often read multiple items from cache simultaneously -- either 3 separate caches for instruction, data, and TLB; or a multiported cache; or both -- which takes less time than reading the same items from main memory one at a time.

The last two ways improve overall performance even if the cache is no faster than main memory.

A processor without a cache has a constant memory reference time T of

$$T = Tm + E$$

A processor with a cache has an average memory access time of[8]

$$T = m * Tm + Th + E$$

where

- m is the miss ratio
- Tm is the time to make a main memory reference
- Th is the time to make a cache reference on a hit
- E accounts for various secondary factors (memory refresh time, multiprocessor contention, etc.)

47.5.1 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.

47.6 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

7 Alan Jay Smith. "Design of CPU Cache Memories". Proc. IEEE TENCON, 1987. `http://www.eecs.berkeley.edu/Pubs/TechRpts/1987/5288.html` `http://www.eecs.berkeley.edu/Pubs/TechRpts/1987/CSD-87-357.pdf`

8 Alan Jay Smith. "Design of CPU Cache Memories". Proc. IEEE TENCON, 1987. `http://www.eecs.berkeley.edu/Pubs/TechRpts/1987/5288.html` `http://www.eecs.berkeley.edu/Pubs/TechRpts/1987/CSD-87-357.pdf`

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.

47.6.1 Inclusive, exclusive, and other cache hierarchy

When there are several levels of cache, and a copy of the data in some location in main memory has been cached in the L1 cache, is there another copy of that data in the L2 cache?

- No. Some systems are designed to have strictly exclusive cache levels: any particular location in main memory is cached in at most one cache level.
- Yes. Other systems are designed to have a strictly inclusive cache levels: whenever some location in main memory is cached in any one level, the same location is also cached in all higher levels. All the data in the L2 cache can also be found in L3 (and also in main memory).

All the data in a L1 cache can also be found in L2 and L3 (and also in main memory).

- Maybe. In some systems, such as the Intel Pentium 4, some data in the L1 cache is also in the L2 cache, while other data in the L1 cache is not in the L2 cache. This kind of cache policy does not yet have a popular name.

47.7 Size of Cache

Figure 71 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.

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.

47.8 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 (a "block in the cache", also called a "line in the cache"[9])

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.

9 Alan Jay Smith. "Design of CPU Cache Memories". Proc. IEEE TENCON, 1987. `http://www.eecs.berkeley.edu/Pubs/TechRpts/1987/5288.html` `http://www.eecs.berkeley.edu/Pubs/TechRpts/1987/CSD-87-357.pdf`

The cache controller splits an effective memory address (MSB[10] to LSB[11]) into the tag, the index, and the block offset.[12][13] Some authors refer to the block offset as simply the "offset"[14] or the "displacement".[15][16]

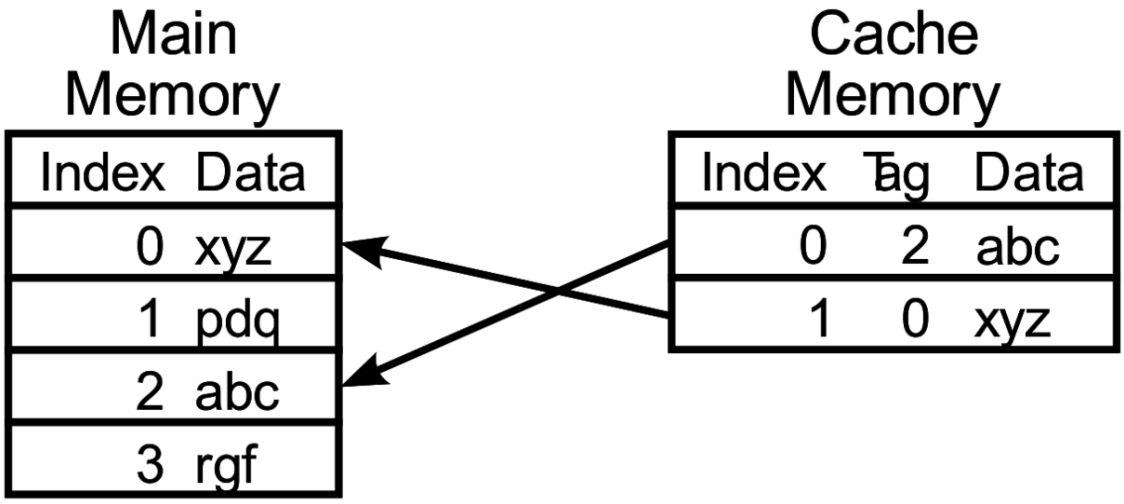

Figure 72 A diagram of cache showing non-sequential data

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

47.9 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

10 http://en.wikibooks.org/wiki/Most%20significant%20bit

11 http://en.wikibooks.org/wiki/Least%20significant%20bit

12 John L. Hennessy, David A. Patterson. "Computer Architecture: A Quantitative Approach". 2011. ISBN 012383872X, ISBN 9780123838728. page B-9. http://books.google.com/books?id=v3-1hVwHnHwC&pg=PA120&lpg=PA120&dq=Hennessey+%22block+offset%22&source=bl&ots=H0RmJ057vE&sig=H5fFbBYAxvTyCeUv2yooeOKxnlM&hl=en&sa=X&ei=NHrwTsPsOtHMsQK-poW-AQ&ved=0CCUQ6AEwAQ#v=onepage&q=%22block%20offset%22&f=false

13 David A. Patterson, John L. Hennessy. "Computer organization and design: the hardware/software interface". 2009. ISBN 0123744938, ISBN 9780123744937 "Chapter 5: Large and Fast: Exploiting the Memory Hierarchy". p. 484. http://books.google.com/books?id=3b63x-0P3_UC&pg=PA484&lpg=PA484&dq=Hennessey+%22block+offset%22&source=bl&ots=Nyek04rcQ5&sig=w7tLCmRZDfyAZ5T8tG3xmfQeDAo&hl=en&sa=X&ei=NHrwTsPsOtHMsQK-poW-AQ&ved=0CCEQ6AEwAA#v=onepage&q=Hennessey%20%22block%20offset%22&f=false

14 Gene Cooperman. "Cache Basics". 2003. http://www.ccs.neu.edu/course/com3200/parent/NOTES/cache-basics.html

15 Ben Dugan. "Concerning Caches". 2002. http://www.cs.washington.edu/education/courses/cse378/02sp/sections/section9-1.html

16 Harvey G. Cragon. "Memory systems and pipelined processors". 1996. ISBN 0867204745, ISBN 9780867204742. "Chapter 4.1: Cache Addressing, Virtual or Real" p. 209 http://books.google.com/books?id=q2w3JSFD714C&pg=PA209&lpg=PA209&dq=displacement+tag+cache&source=bl&ots=i3HOLDymZk&sig=VOnTozBRVPb8BTcphIPSPvvFNSU&hl=en&sa=X&ei=spTwTsm0KtHMsQK-poW-AQ&ved=0CEkQ6AEwBQ#v=onepage&q=displacement%20tag%20cache&f=false

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.

47.9.1 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.

47.9.2 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.

47.9.3 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.

47.9.4 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.

block size

A very small cache block size increases the miss ratio, since a miss will fetch less data at a time. A very large cache block size also increases the miss ratio, since it causes the system to fetch a bunch of extra information that is used less than the data it displaces in the cache. [17]

47.10 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.

47.10.1 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

17 Alan Jay Smith. "Design of CPU Cache Memories". Proc. IEEE TENCON, 1987. `http://www.eecs.berkeley.edu/Pubs/TechRpts/1987/5288.html` `http://www.eecs.berkeley.edu/Pubs/TechRpts/1987/CSD-87-357.pdf`

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.

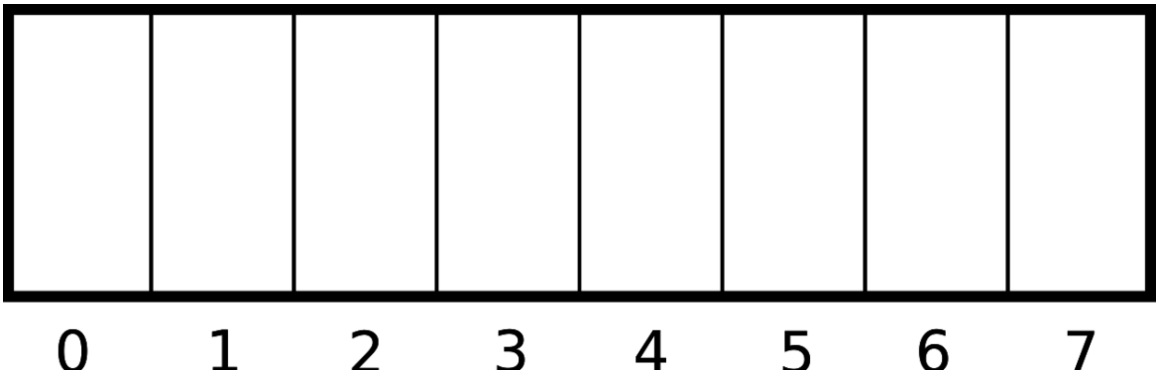

Figure 73

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).

Most external caches (located on the motherboard, but external to the CPU) are direct-mapped or occasionally 2-way set associative, because it's complicated to build higher-associativity caches out of standard components.[18] If there is such a cache, typically there is only one external cache on the motherboard, shared between all CPUs.

The replacement policy for a direct-mapped cache is the simplest possible replacement policy: the new data must go in the one and only one place in the cache it corresponds to. (The old data at the location in the cache, if its dirty bit is set, must be written to main memory first).

47.10.2 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

18 Paul V. Bolotoff. "Functional Principles of Cache Memory" ^{`http://alasir.com/articles/cache_principles/cache_hierarchy.html`} . 2007.

are quick because the cache controller can immediately narrow down its search area to the block that matches the address hash value.

The LRU replacement policy for a 2-way set associative cache is one of the simplest replacement policies: The new data must go in one of a set of 2 possible locations. Those 2 locations share a LRU bit that is updated whenever either one is read or written, indicating which one of the two entries in the set was the most-recently used. The new data goes in the *other* location (the least-recently used location). (The old data at that LRU location in the cache, if its dirty bit is set, must be written to main memory first).

47.10.3 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 SeznecA Case for Two-Way Skewed-Associative Caches [19]. . Retrieved 2007-12-13 [20]

47.10.4 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.

47.11 Cache Misses

There are three basic types of misses in a cache:

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

19

20 Micro-Architecture ^{`http://www.irisa.fr/caps/PROJECTS/Architecture/`} "Skewed-associative caches have ... major advantages over conventional set-associative caches."

47.11.1 Conflict Misses

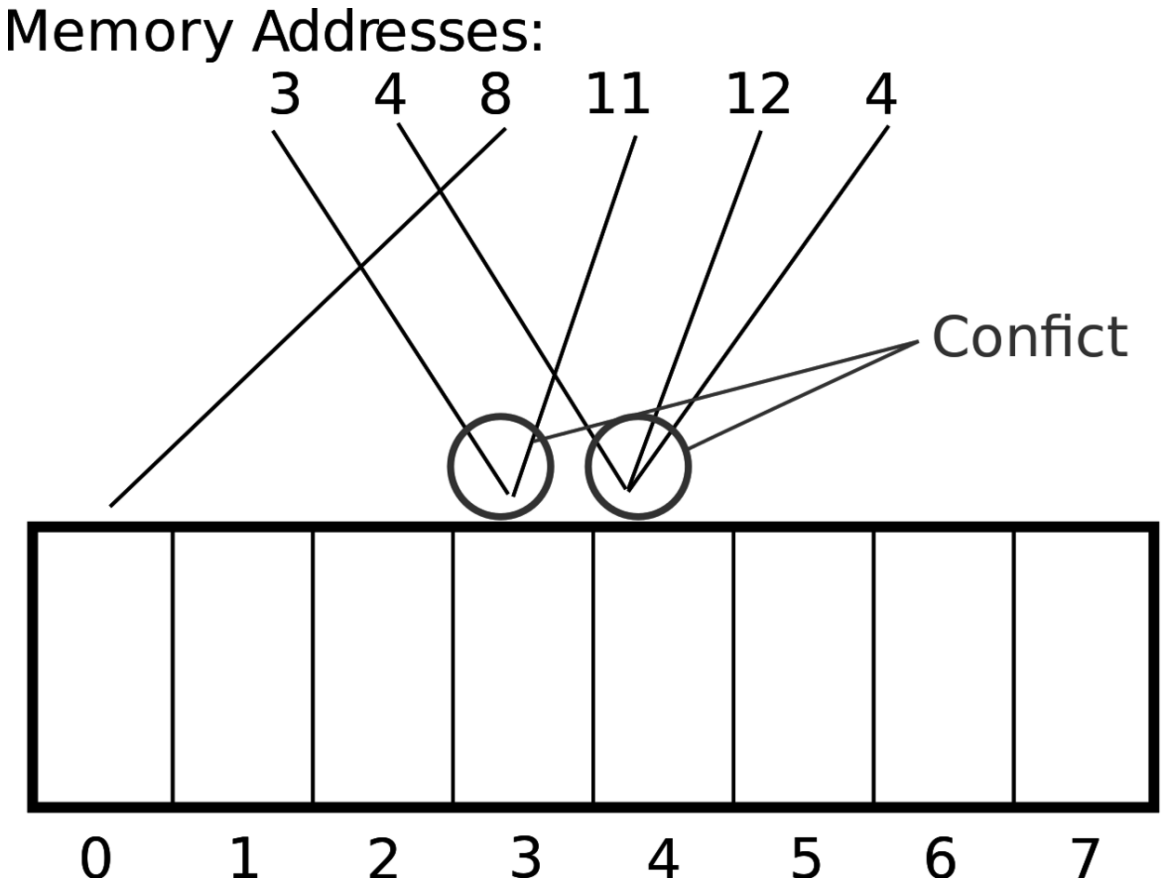

Figure 74

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.

47.11.2 Compulsory Misses

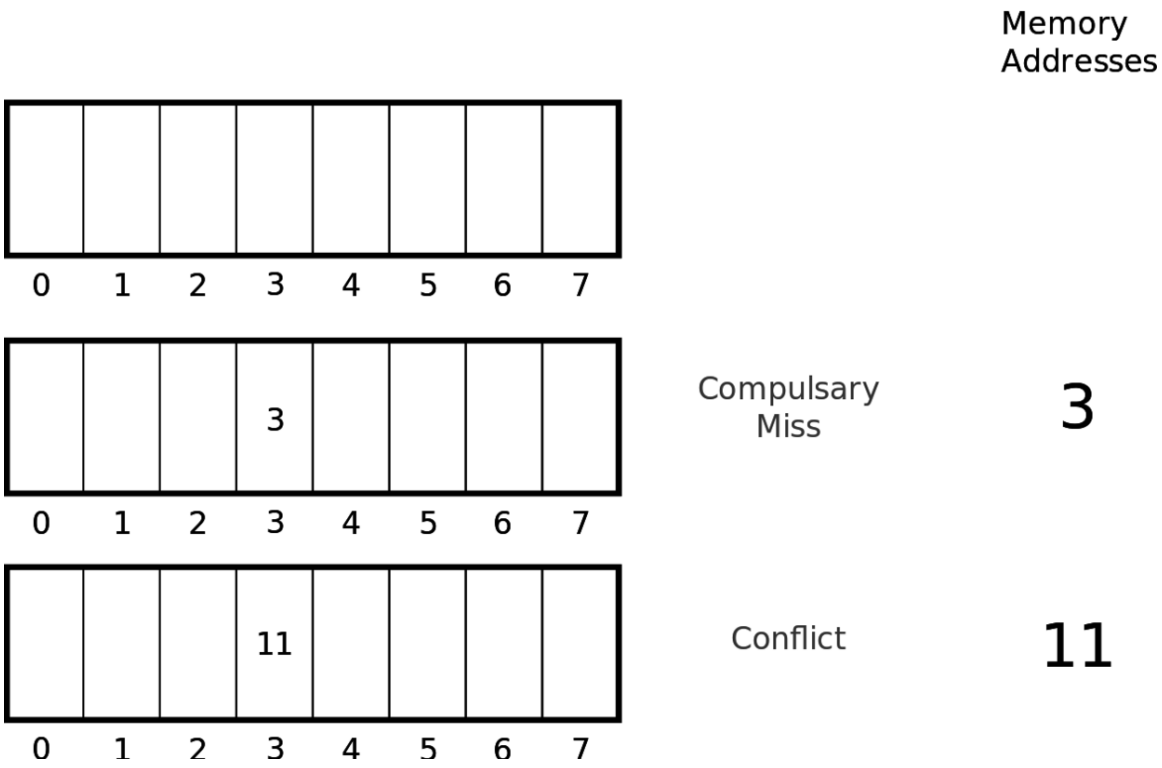

Figure 75

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.

47.11.3 Capacity Misses

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

47.12 Cache 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** .

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.

47.12.1 Write Through

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.

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.

47.12.2 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.

47.12.3 Write bypass

Some processors send writes directly to main memory, bypassing the cache. If that location is *not* already cached, then nothing more needs to be done. If that location *is* already cached, then the old data in the cache(s) needs to be marked "invalid" ("stale") so if the CPU ever reads that location, the CPU will read the latest value from main memory rather than some earlier value(s) in the cache(s).

47.13 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. Some computer systems have several CPUs connected to a common main memory. 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".

The three most popular approaches to dealing with stale data ("cache coherency protocols") are:

- Use simple cache hardware that ignores what the other CPUs are doing.
- Set all caches to write-through all STOREs (write-through policy). Use additional cache hardware to listen in ("snoop") whenever some other device writes to main memory, and invalidate local cache line whenever some other device writes to the corresponding cached location in main memory.

w:MESI protocol[21]

- Design caches to use the MESI protocol.

With simple cache hardware that ignores what the other CPUs are doing, cache coherency is maintained by the OS software. The OS sets up each page in memory as either (a) exclusive to one particular CPU (which is allowed to read, write, and cache it); all other CPUs are not allowed to read or write or cache that page; (b) shared read/write between CPUs, and set to "non-cacheable", in the same way that memory-mapped I/O devices are set to non-cacheable; or (c) shared read-only; all CPUs are allowed to cache but not write that page.

47.14 Split cache

High-performance processors invariably have 2 separate L1 caches, the instruction cache and the data cache (I-cache and D-cache). This "split cache" has several advantages over a unified cache:[22]

- Wiring simplicity: the decoder and scheduler are only hooked to the I-cache; the registers and ALU and FPU are only hooked to the D-cache.
- Speed: the CPU can be reading data from the D-cache, while simultaneously loading the next instruction(s) from the I-cache.

Multi-CPU systems typically have a separate L1 I-cache and L1 D-cache for each CPU, each one direct-mapped for speed.

Open question: To speed up running Java applications in a JVM (and similar interpreters and CPU emulators), would it help to have 3 separate caches -- a machine instruction cache indexed by the program counter PC, a byte code cache indexed by the VM's instruction pointer IP, and a data cache ?

On the other hand, in a high-performance processor, other levels of cache, if any -- L2, L3, etc. -- as well as main memory -- are typically unified, although there are several exceptions (such as the Itanium 2 Montecito). The advantages of a unified cache (and a unified main memory) are:[23]

21 http://en.wikipedia.org/wiki/MESI%20protocol

22 Paul V. Bolotoff. "Functional Principles of Cache Memory" ^{http://alasir.com/articles/cache_principles/cache_hierarchy.html} . 2007.

23 Paul V. Bolotoff. "Functional Principles of Cache Memory" ^{http://alasir.com/articles/cache_principles/cache_hierarchy.html} . 2007.

- Some programs spend most of their time in a small part of the program processing lots of data. Other programs run lots of different subroutines against a small amount of data. A unified cache automatically balances the proportion of the cache used for instructions and the proportion used for data -- to get the same performance on a split cache would require a larger cache.
- when instructions are written to memory -- by an OS loading an executable file from storage, or from a just-in-time compiler translating bytecode to executable code -- a split cache requires the CPU to flush and reload the instruction cache; a unified cache doesn't require that.

47.15 Error detection

Each cache row entry may have error detection bits. Since the cache only holds a copy of information in the main memory (except for the write-back queue), when an error is detected, the desired data can be re-fetched from the main memory -- treated as a kind of miss-on-invalid -- and the system can continue as if no error occurred. A few computer systems use Hamming error correction to correct single-bit errors in the "data" field of the cache without going all the way back to main memory.[24]

47.16 Specialized cache features

Many CPUs use exactly the same hardware for the instruction cache and the data cache. (And, of course, the same hardware is used for instructions as for data in a unified cache. The revolutionary idea of a Von Neumann architecture is to use the same hardware for instructions and for data in the main memory itself). For example, the Fairchild CLIPPER used 2 identical CAMMU chips, one for the instruction cache and one for the data cache.[25]

Because the various caches are used slightly differently, some CPU designers customize each cache in different ways.

- Some CPU designers put the "branch history bits" used for branch prediction[26] in the instruction cache. There's no point to adding such information to a data-only cache.
- Many instruction caches are designed in such a way that the only way to deal with stale instructions is to invalidate the entire cache and reload. Data caches are typically designed with more fine-grained response, with extra hardware that can invalidate and reload only the particular cache lines that have gone stale.
- The virtual-to-physical address translation process often has a lot of specialized hardware associated with it to make it go faster -- the TLB cache, hardware page-walkers, etc. We will discuss this in more detail in the next chapter, ../Virtual Memory/[27].

24 Paul V. Bolotoff. Functional Principles of Cache Memory ^{http://alasir.com/articles/cache_principles/cache_line_tag_index.html} . 2007.

25 Alan Jay Smith. "Design of CPU Cache Memories". Proc. IEEE TENCON, 1987. http://www.eecs.berkeley.edu/Pubs/TechRpts/1987/5288.html http://www.eecs.berkeley.edu/Pubs/TechRpts/1987/CSD-87-357.pdf

26 Chapter 31.4.7 on page 139

27 Chapter 48 on page 197

47.17 References

47.18 Further reading

w: cache[28] w: cache algorithms[29] w: CPU cache[30] w: Cache Coherency[31]

- Parallel Computing and Computer Clusters/Memory[32]
- simulators available for download at University of Maryland: Memory-Systems Research: "Computational Artifacts"[33] 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?")

28 http://en.wikipedia.org/wiki/%20cache
29 http://en.wikipedia.org/wiki/%20cache%20algorithms
30 http://en.wikipedia.org/wiki/%20CPU%20cache
31 http://en.wikipedia.org/wiki/%20Cache%20Coherency
32 http://en.wikibooks.org/wiki/Parallel%20Computing%20and%20Computer%20Clusters%2FMemory
33 http://www.ece.umd.edu/~blj/memory/artifacts.html

48 Virtual Memory

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.

If a simple virtual==physical address path is adequate for your CPU, you don't need virtual memory.

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. This enables virtual memory, memory protection, and other desirable features.

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 RAM at the same time at 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.

48.1 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.

Paging systems are designed to be transparent, that is, the (user-mode) 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, so that data pages cannot be executed as instructions, and instructions cannot be corrupted by data writes (see W^X[1]).

The hardware part of virtual memory is called the memory management unit (MMU). Most MMUs have a granularity of one page.

A few CPU designs use a more fine-grained access control to detect and prevent buffer overflow bugs, a common security vulnerability.[2]

48.2 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 20 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 20 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.

48.3 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, it can frequently be moved to secondary storage instead. The area in the secondary storage is typically known as the **page file** , the ”scratchpad”, or something similar.

48.4 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.

1 `http://en.wikipedia.org/wiki/W%5EX`

2 Albert Kwon, Udit Dhawan, Jonathan M. Smith, Thomas F. Knight, Jr., and André DeHon. ”Low-Fat Pointers: Compact Encoding and Efficient Gate-Level Implementation of Fat Pointers for Spatial Safety and Capability-based Security” ^{`http://www.crash-safe.org/node/27`} . 2013.

48.4.1 Page Faults

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

48.4.2 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 can be considered a kind of cache, caching part of the page table stored in main memory, it is a physically separate structure than the instruction cache or the data cache, and has several features not found in those caches.

TLB entry

The SRAM in the TLB can be seen as entirely composed of TLB entries. The format of the TLB entries in the TLB SRAM is fixed by the TLB hardware. The paging supervisor -- part of the operating system -- typically maintains a page table in main memory which stores the page table entries in exactly the same format as TLB entries. Each TLB entry contains:

- the virtual address of a page (analogous to the "tag" in a cache)
- the physical address of a page (analogous to the "data" in a cache)

While not essential, some TLB hardware has many other optional control and protection fields and flags in the TLB, including:

- the no-execute bit (NX bit), used to implement W^X ("Write XOR Execute")
- a "dirty bit" (also called the "modified bit"), set whenever there is a STORE written into that page, and typically cleared when the modified page is written to the backing store.
- the writable bit, used to implement PaX, sometimes cleared and later set by the OS in order to implement copy-on-write (COW)
- which virtual address space a physical page belongs to (unnecessary on a single address space operating system[3])
- the supervisor bit
- statistics on which TLB entries were most recently or most frequently used, used to decide which TLB entry to discard when loading a new TLB entry from main memory
- statistics on which page was most recently or most frequently used, used to support LRU or more sophisticated page-replacement algorithms that decide which page currently in main memory to "page out" to the backing store when the OS needs to load some other page from the backing store into physical memory

The page table entries may include additional per-page fields that are not copied into the TLB entries, such as

3 http://en.wikipedia.org/wiki/single%20address%20space%20operating%20system

- the "pinned bit" (aka "fixed flag") that indicates that a page must stay in main memory -- the paging supervisor marks as pinned pages that must stay in main memory, including the paging supervisor executable code itself, the device drivers for the secondary storage devices on which pages are swapped out; interrupt handler executable code. Some data buffers are also pinned during I/O transactions during the time that devices outside the CPU read or write those buffers (direct memory access and I/O channel hardware).
- a "present" bit (clear when that particular virtual page does not currently exist in physical main memory)

48.5 Further reading

- Thomas W. Barr, Alan L. Cox, Scott Rixner. "Translation Caching: Skip, Don't Walk (the Page Table)"[4]. describes the Intel x86-64 MMU cache, the AMD Page Walk Cache, 3 other MMU cache arrangements, and compares their performance.
- B. Jacob, and T. Mudge. "Virtual memory in contemporary microprocessors".[5] IEEE Micro 1998 July.

4 http://www.cs.rice.edu/CS/Architecture/docs/barr-isca10.pdf
5 http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.26.352&rep=rep1&type=pdf

49 Power Dissipation

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 than processors with more capability and better performance.

The primary problem in server farms like the ones used by Google is power.[1]

Reducing the amount of energy used, without reducing the performance of the computer system, is one of the Grand Challenges in computer science.[2]

49.1 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.

49.2 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.

---

1 Cade Metz. "How Google Spawned The 384-Chip Server" ^{http://www.wired.com/wiredenterprise/2012/01/seamicro-and-google/all/1} . 2012.

2 "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.

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.

49.3 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.

49.4 further reading

w:low-power electronics[3] w:clock gating[4] w:performance per watt[5] w:data center infrastructure efficiency[6] w:server farm[7] w:Thermal Design Power[8]

- How To Assemble A Desktop PC/Silencing[9] describes some of the problems caused cooling fans, which wouldn't be necessary if CPUs generated less heat.

49.5 Resources

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

3 http://en.wikipedia.org/wiki/low-power%20electronics
4 http://en.wikipedia.org/wiki/clock%20gating
5 http://en.wikipedia.org/wiki/performance%20per%20watt
6 http://en.wikipedia.org/wiki/data%20center%20infrastructure%20efficiency
7 http://en.wikipedia.org/wiki/server%20farm
8 http://en.wikipedia.org/wiki/Thermal%20Design%20Power
9 http://en.wikibooks.org/wiki/How%20To%20Assemble%20A%20Desktop%20PC%2FSilencing

50 Resources

50.1 Further Reading

50.1.1 Related Wikibooks

- Chip Design Made Easy[1]
- MIPS Assembly[2]
- SPARC Assembly[3]
- Programmable Logic[4]
- Semiconductors[5]
- Digital Circuits[6]
- Parallel Computing and Computer Clusters[7]
- Floating Point[8]
- Embedded Control Systems Design/Processors[9]
- Embedded Systems/Microprocessor Architectures[10]
- Floating Point/Floating Point Hardware[11]

50.1.2 Wikipedia Articles

w:CPU design[12]

- Wikipedia:Microprocessor[13]
- Wikipedia:CPU design[14]
- Wikipedia:Instruction set[15]
- Apollo Guidance Computer[16]

1 http://en.wikibooks.org/wiki/Chip%20Design%20Made%20Easy
2 http://en.wikibooks.org/wiki/MIPS%20Assembly
3 http://en.wikibooks.org/wiki/SPARC%20Assembly
4 http://en.wikibooks.org/wiki/Programmable%20Logic
5 http://en.wikibooks.org/wiki/Semiconductors
6 http://en.wikibooks.org/wiki/Digital%20Circuits
7 http://en.wikibooks.org/wiki/Parallel%20Computing%20and%20Computer%20Clusters
8 http://en.wikibooks.org/wiki/Floating%20Point
9 http://en.wikibooks.org/wiki/Embedded%20Control%20Systems%20Design%2FProcessors
10 http://en.wikibooks.org/wiki/Embedded%20Systems%2FMicroprocessor%20Architectures
11 http://en.wikibooks.org/wiki/Floating%20Point%2FFloating%20Point%20Hardware
12 http://en.wikipedia.org/wiki/CPU%20design
13 http://en.wikipedia.org/wiki/Microprocessor
14 http://en.wikipedia.org/wiki/CPU%20design
15 http://en.wikipedia.org/wiki/Instruction%20set
16 http://en.wikipedia.org/wiki/Apollo%20Guidance%20Computer

- Wikipedia:soft microprocessor[17] discusses FPGA CPUs

50.1.3 Wikiversity Courses

- Wikiversity:Computer_Architecture_Lab[18]

50.1.4 Commons Image Categories

- Commons:Category:Microprocessors[19]
- Commons:Category:Microcontrollers[20]

50.1.5 External Links

- "homebrew CPU"[21].
- Knowledge and Concepts of VLSI Chip Design and Development[22]
- "Great moments in microprocessor history"[23] by W. Warner 2004
- c2:AlternativeMicroprocessorDesign[24]
- c2:DoMicroprocessorsLoveCee[25]
- `http://www.engineersgarage.com/articles/rtos-real-time-operating-system`]www.engineersgar
- [26]www.howstuffworks.com
- `http://www.webopedia.com/TERM/O/operating_system.html`www.webopedia.com
- `http://www.slideshare.net/murugan_m1/embedded-system-basics`www.slideshare.net
- Daniel J. Sorin[27] has some good notes online for classes he's taught: "ECE 152: Introduction to Computer Architecture" and "ECE 252 / CPS 220: Advanced Computer Architecture I".
- Jacob Nelson[28] has some information online about his microprocessor designs: "An FPGA-based Custom Computer" and "The uToad Proof of Concept", both reminiscent of the PDP-10.
- OpenCores[29] has many open-hardware FPGA and CPLD designs under development, including dozens of microprocessors[30]. These include entirely new processors such as "JOP: a Java Optimized Processor", "ZPU - the worlds smallest 32 bit CPU with GCC toolchain", the "OpenRISC 1000", "MCPU ... fits into a 32 Macrocell CPLD". These also include processors designed to be software compatible with ("clean-room re-implementations of")

17 `http://en.wikipedia.org/wiki/soft%20microprocessor`
18 `http://en.wikiversity.org/wiki/Computer_Architecture_Lab`
19 `http://en.commons.org/wiki/Category%3AMicroprocessors`
20 `http://en.commons.org/wiki/Category%3AMicrocontrollers`
21 `http://homebrewcpu.com/links.htm`
22 `http://www.vlsichipdesign.com`
23 `http://www.ibm.com/developerworks/library/pa-microhist.html`
24 `http://en.wikibooks.org/wiki/c2%3AAlternativeMicroprocessorDesign`
25 `http://en.wikibooks.org/wiki/c2%3ADoMicroprocessorsLoveCee`
26 `http://www.howstuffworks.com/operating-system1.htm`
27 `http://www.ee.duke.edu/~sorin/`
28 `http://jfet.net/grad/`
29 `http://www.opencores.org/`
30 `http://opencores.org/projects?cat=Processor`

several older proprietary processors -- MIPS, ARM, x86, AVR, PIC, 68HC11, 68000, Alpha, etc.

- The Am1601[31] is a stack based CPU implemented in a FPGA. It is designed to be radiation tolerant.
- VHDL Source Code for Simple 8-bit CPU[32]
- "Microprocessor Architectures"[33] has a "Java based simulator of a pipelined processor. ... The Java code is written in a style to simplify the process of converting the processor into a ... a FPGA implementation."
- "CPU Design HOW-TO"[34] by Alavoor Vasudevan 2002
- The Advanced Processor Technologies Group at Manchester[35] has microprocessor design and synthesis tools you can download and use.
- "The "high-level CPU" challenge"[36] and "'High-level CPU": follow-up"[37] by Yossi Kreinin (and attached comments by a variety of other people) has some interesting ideas on CPU design.
- YASEP means "Yet Another Small Embedded Processor"[38] by Yann Guidon: RTL source code is in VHDL, currently targeting the Actel ProASIC3 FPGA; simulator, an assembler, a disassembler, a manual, a development tool, all available for download (open source).
- StackOverflow: How does an assembly instruction turn into voltage changes on the CPU?[39] -- good book recommendations, and an attempt to briefly summarize what this book is all about.
- Stackexchange: "A fun book to learn computer architecture"[40] lists a few books on computer architecture and CPU design.
- Stackexchange: "Readable and educational implementations of a CPU in a HDL"[41]
- Non-Von1 in a Spartan3E-1200 FPGA board[42]
- Homebrew Cray-1A on a Xilinx Spartan-3E 1600 FPGA development board[43]: built into a 1/10 scale model.
- fpga-cpu : FPGA CPU and SoC discussion list[44]: list is for discussion of the design and implementation of field-programmable gate array based processors and integrated systems. It is also for discussion and community support of the XSOC Project[45].
- "Elementary Microprocessor ... The EM is intended as a simple microprocessor for educational purposes for those of us who are interested in learning not just what a CPU does, but *exactly how* a CPU works"`http://code.google.com/p/elementary-microprocessor/`

31 http://www.amsat.org/amsat/projects/ips/Am1601.html
32 http://web.archive.org/20040618011640/www.geocities.com/leon_heller/cpu.html
33 http://www.labbookpages.co.uk/teaching/upArch.html
34 http://www.faqs.org/docs/Linux-HOWTO/CPU-Design-HOWTO.html
35 http://intranet.cs.manchester.ac.uk/intranet/apt/
36 http://www.yosefk.com/blog/the-high-level-cpu-challenge.html
37 http://www.yosefk.com/blog/high-level-cpu-follow-up.html
38 http://yasep.org/
39 http://stackoverflow.com/questions/3706022/how-does-an-assembly-instruction-turn-into-voltage-changes-on-
40 http://electronics.stackexchange.com/questions/5516/a-fun-book-to-learn-computer-architecture-for-not-exa
41 http://electronics.stackexchange.com/questions/1754/readable-and-educational-implementations-of-a-cpu-in-
42 http://chrisfenton.com/non-von-1/
43 http://chrisfenton.com/homebrew-cray-1a/
44 http://tech.dir.groups.yahoo.com/group/fpga-cpu/
45 http://www.fpgacpu.org/xsoc

  - The original EM was designed on the open-source Logisim digital logic simulator`http://ozark.hendrix.edu/~burch/logisim/`.
- HORNET is a highly configurable, cycle-level multicore simulator with support for power and thermal modeling. HORNET software uses several cores when run on multicore host hardware, and it supports simulating chips with over 100 cores. Mieszko Lis, Pengju Ren, Myong Hyon Cho, Keun Sup Shim, Christopher W. Fletcher, Omer Khan and Srinivas Devadas, "Scalable, accurate multicore simulation in the 1000-core era".`http://csg.csail.mit.edu/hornet/`

50.1.6 Books

- Patterson and Hennessy, *Computer Organization and Design* , 3rd Edition, Morgan Kaufman, 2005. ISBN 1558606041
- *... should we list the other design books recommended by John Doran*[46] *? ...*
- Nisan and Schocken. "The Elements of Computing Systems: Building a Modern Computer from First Principles". 2005. ISBN 978-0262640688. `http://www1.idc.ac.il/tecs/`
  - Shimon Schocken. "From NAND to Tetris in 12 steps: building a modern computer from first principles" `http://video.google.com/videoplay?docid=7654043762021156507&q=type%3Agoogle+engEDU&total=540&start=0&num=10&so=1&type=search&plindex=6` is an overview of the Nisan and Schocken book.
- Hamacher, Vranesic, Zaky, Manjikian. "Computer Organization and Embedded Systems". ISBN 978-0073380650

46 `http://www.timefracture.org/D16docs/D16_Design_Notes.html`

51 Contributors

Edits	User
1	Addihockey10 (automated)[1]
4	Adrignola[2]
1	Avicennasis[3]
1	Billinghurst[4]
1	Cburnett[5]
12	Chazz[6]
1	CommonsDelinker[7]
1	Darklama[8]
124	DavidCary[9]
17	Ervinn[10]
1	Hagindaz[11]
1	HethrirBot[12]
1	Hoo man[13]
1	Iste Praetor[14]
2	JackPotte[15]
23	Jfmantis[16]
17	Jomegat[17]
1	LlamaAl[18]
1	MichaelFrey[19]
1	Mike.lifeguard[20]
3	Panic2k4[21]

1 http://en.wikibooks.org/wiki/User:Addihockey10_(automated)
2 http://en.wikibooks.org/wiki/User:Adrignola
3 http://en.wikibooks.org/wiki/User:Avicennasis
4 http://en.wikibooks.org/wiki/User:Billinghurst
5 http://en.wikibooks.org/wiki/User:Cburnett
6 http://en.wikibooks.org/wiki/User:Chazz
7 http://en.wikibooks.org/wiki/User:CommonsDelinker
8 http://en.wikibooks.org/wiki/User:Darklama
9 http://en.wikibooks.org/wiki/User:DavidCary
10 http://en.wikibooks.org/wiki/User:Ervinn
11 http://en.wikibooks.org/wiki/User:Hagindaz
12 http://en.wikibooks.org/wiki/User:HethrirBot
13 http://en.wikibooks.org/wiki/User:Hoo_man
14 http://en.wikibooks.org/wiki/User:Iste_Praetor
15 http://en.wikibooks.org/wiki/User:JackPotte
16 http://en.wikibooks.org/wiki/User:Jfmantis
17 http://en.wikibooks.org/wiki/User:Jomegat
18 http://en.wikibooks.org/wiki/User:LlamaAl
19 http://en.wikibooks.org/wiki/User:MichaelFrey
20 http://en.wikibooks.org/wiki/User:Mike.lifeguard
21 http://en.wikibooks.org/wiki/User:Panic2k4

6	QuiteUnusual[22]
5	Recent Runes[23]
1	Red4tribe[24]
1	Remi[25]
1	Ruy Pugliesi[26]
2	Spyk[27]
1	Syum90[28]
1	Thenub314[29]
335	Whiteknight[30]
4	Xania[31]
1	YMS[32]

22 http://en.wikibooks.org/wiki/User:QuiteUnusual
23 http://en.wikibooks.org/wiki/User:Recent_Runes
24 http://en.wikibooks.org/wiki/User:Red4tribe
25 http://en.wikibooks.org/wiki/User:Remi
26 http://en.wikibooks.org/wiki/User:Ruy_Pugliesi
27 http://en.wikibooks.org/wiki/User:Spyk
28 http://en.wikibooks.org/wiki/User:Syum90
29 http://en.wikibooks.org/wiki/User:Thenub314
30 http://en.wikibooks.org/wiki/User:Whiteknight
31 http://en.wikibooks.org/wiki/User:Xania
32 http://en.wikibooks.org/wiki/User:YMS

List of Figures

- GFDL: Gnu Free Documentation License. http://www.gnu.org/licenses/fdl.html
- cc-by-sa-3.0: Creative Commons Attribution ShareAlike 3.0 License. http://creativecommons.org/licenses/by-sa/3.0/
- cc-by-sa-2.5: Creative Commons Attribution ShareAlike 2.5 License. http://creativecommons.org/licenses/by-sa/2.5/
- cc-by-sa-2.0: Creative Commons Attribution ShareAlike 2.0 License. http://creativecommons.org/licenses/by-sa/2.0/
- cc-by-sa-1.0: Creative Commons Attribution ShareAlike 1.0 License. http://creativecommons.org/licenses/by-sa/1.0/
- cc-by-2.0: Creative Commons Attribution 2.0 License. http://creativecommons.org/licenses/by/2.0/
- cc-by-2.0: Creative Commons Attribution 2.0 License. http://creativecommons.org/licenses/by/2.0/deed.en
- cc-by-2.5: Creative Commons Attribution 2.5 License. http://creativecommons.org/licenses/by/2.5/deed.en
- cc-by-3.0: Creative Commons Attribution 3.0 License. http://creativecommons.org/licenses/by/3.0/deed.en
- GPL: GNU General Public License. http://www.gnu.org/licenses/gpl-2.0.txt
- LGPL: GNU Lesser General Public License. http://www.gnu.org/licenses/lgpl.html
- PD: This image is in the public domain.
- ATTR: The copyright holder of this file allows anyone to use it for any purpose, provided that the copyright holder is properly attributed. Redistribution, derivative work, commercial use, and all other use is permitted.
- EURO: This is the common (reverse) face of a euro coin. The copyright on the design of the common face of the euro coins belongs to the European Commission. Authorised is reproduction in a format without relief (drawings, paintings, films) provided they are not detrimental to the image of the euro.
- LFK: Lizenz Freie Kunst. http://artlibre.org/licence/lal/de
- CFR: Copyright free use.

- EPL: Eclipse Public License. `http://www.eclipse.org/org/documents/epl-v10.php`

Copies of the GPL, the LGPL as well as a GFDL are included in chapter Licenses[33]. Please note that images in the public domain do not require attribution. You may click on the image numbers in the following table to open the webpage of the images in your webbrower.

33 Chapter 52 on page 217

1	Aleator, Americophile, BMK, Berrucomons, Boivie, Bot-Multichill, Edward, Emijrpbot, Gustavb, Hazard-Bot, Hr.hanafi, Huhsunqu, J.delanoy, JarektBot, Jianhui67, Jon Harald Søby, Kozuch, Mdd, Mhare, Monsterxxl, Origamiemensch, Rocket000, Slovik, Ss181292, Torsch, UED77, ☒☒☒☒ robot	
2	w:en:user:Wgsimon[34]	
3	en:User:Booyabazooka[35]	GFDL
4	Abelsson[36]	GFDL
5	BotMultichill, JackPotte, Jarekt, JarektBot, Jochen Burghardt, LoopZilla, Ma-Lik, Mdd, Moonshadow, StuartBrady	
6	en:User:Cburnett[37]	GFDL
7	en:User:Cburnett[38]	GFDL
8	en:User:Cburnett[39]	GFDL
9	en:User:Cburnett[40]	GFDL
10	Whiteknight	
11	Whiteknight	
12	Whiteknight	
13	Whiteknight	
14	QuiteUnusual, Whiteknight	
15	Whiteknight	
16	Whiteknight	
17	Whiteknight	
18	Whiteknight	
19	Whiteknight	
20	Whiteknight	
21	Whiteknight	
22	en:User:Cburnett[41]	GFDL
23	Cburnett	CC-BY-SA-3.0
24	EugeneZelenko, Glenn, Ilmari Karonen, JarektBot, Lambtron, MGA73bot2, Omegatron, Poil, Wst	
25	JarektBot, Lambtron, MGA73bot2, Poil, StuartBrady	
26	JarektBot, Lambtron, MGA73bot2, Poil, StuartBrady	
27	JarektBot, Lambtron, MGA73bot2, Poil, StuartBrady	
28	JarektBot, Lambtron, MGA73bot2, Poil, StuartBrady	
29	User:Stannered[42]	GFDL
30	en:User:Cburnett[43]	GFDL
31	en:User:Cburnett[44]	GFDL

34 http://en.wikipedia.org/wiki/en:user:Wgsimon
35 http://en.wikipedia.org/wiki/User:Booyabazooka
36 http://commons.wikimedia.org/wiki/User:Abelsson
37 http://en.wikipedia.org/wiki/User:Cburnett
38 http://en.wikipedia.org/wiki/User:Cburnett
39 http://en.wikipedia.org/wiki/User:Cburnett
40 http://en.wikipedia.org/wiki/User:Cburnett
41 http://en.wikipedia.org/wiki/User:Cburnett
42 http://commons.wikimedia.org/wiki/User:Stannered
43 http://en.wikipedia.org/wiki/User:Cburnett
44 http://en.wikipedia.org/wiki/User:Cburnett

32	en:User:Cburnett[45]	GFDL
33	en:User:Cburnett[46]	GFDL
34	en:User:Cburnett[47]	GFDL
35	JarektBot, MGA73bot2, Mdd, Poil, StuartBrady	
36	en:User:Cburnett[48]	GFDL
37	en:User:Cburnett[49]	GFDL
38	en:User:Cburnett[50]	GFDL
39	en:User:Cburnett[51]	GFDL
40	JarektBot, MGA73bot2, Mdd, Poil, StuartBrady, Teslaton, Wikibob	
41	JarektBot, MGA73bot2, Mdd, Poil, StuartBrady, Teslaton	
42	en:User:Cburnett[52]	GFDL
43	en:User:Cburnett[53]	GFDL
44	en:User:Cburnett[54]	GFDL
45	en:User:Cburnett[55]	GFDL
46	en:User:Cburnett[56]	GFDL
47	BotMultichill, BotMultichillT, Emijrpbot, JarektBot, MGA73bot2, Poil, SchlurcherBot, Wknight94	
48	BotMultichill, BotMultichillT, Emijrpbot, JarektBot, MGA73bot2, Poil, SchlurcherBot, Wknight94	
49	Emijrpbot, Jafeluv, JarektBot, MGA73bot2, Mahahahaneapneap, Plugwash, Poil, WikipediaMaster	
50	Hellisp	
51	Hellisp	
52	Hellisp	
53	Unknown	PD
54	Amit6, Emijrpbot, JarektBot, MGA73bot2, Mahahahaneapneap, Poil, StuartBrady, WikipediaMaster	
55	Amit6, BotMultichill, BotMultichillT, JarektBot, MGA73bot2, Poil	
56	BotMultichill, BotMultichillT, JarektBot, MGA73bot2, Poil	
57	Jürgen Melzer	GFDL
58	Hellisp	
59	Hellisp	
60	Whiteknight	
61	Whiteknight	
62	Hellisp	

45 http://en.wikipedia.org/wiki/User:Cburnett
46 http://en.wikipedia.org/wiki/User:Cburnett
47 http://en.wikipedia.org/wiki/User:Cburnett
48 http://en.wikipedia.org/wiki/User:Cburnett
49 http://en.wikipedia.org/wiki/User:Cburnett
50 http://en.wikipedia.org/wiki/User:Cburnett
51 http://en.wikipedia.org/wiki/User:Cburnett
52 http://en.wikipedia.org/wiki/User:Cburnett
53 http://en.wikipedia.org/wiki/User:Cburnett
54 http://en.wikipedia.org/wiki/User:Cburnett
55 http://en.wikipedia.org/wiki/User:Cburnett
56 http://en.wikipedia.org/wiki/User:Cburnett

63	Original uploader was Whiteknight[57] at en.wikibooks[58]	
64	Jürgen Melzer	GFDL
65	Hellisp	
66	Hellisp	
67	Whiteknight	
68	User Baz1521[59] on ja.wikipedia[60]	GFDL
69	Krdan[61]	GFDL
70	D-Kuru, Denniss, Emijrpbot, Hazard-Bot, JarektBot, Rosco	
71	Adamantios	GFDL
72	Traced by User:Stannered[62]	GFDL
73	Whiteknight	
74	Whiteknight	
75	Whiteknight	

57 http://en.wikibooks.org/wiki/en:User:Whiteknight
58 http://en.wikibooks.org
59 http://ja.wikipedia.org/wiki/User:Baz1521
60 http://ja.wikipedia.org
61 http://commons.wikimedia.org/wiki/User:Krdan
62 http://commons.wikimedia.org/wiki/User:Stannered

52 Licenses

52.1 GNU GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed. Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program–to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow. TERMS AND CONDITIONS 0. Definitions.

"This License" refers to version 3 of the GNU General Public License.

"Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

"The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based on the Program.

To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion. 1. Source Code.

The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work. 2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary. 3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures. 4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee. 5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

* a) The work must carry prominent notices stating that you modified it, and giving a relevant date. * b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices". * c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it. * d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate. 6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

* a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange. * b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge. * c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b. * d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements. * e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying. 7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

* a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or * b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or * c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or * d) Limiting the use for publicity purposes of names of licensors or authors of the material; or * e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or * f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way. 8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10. 9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so. 10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it. 11. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law. 12. No Surrender of Others' Freedom.

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy

both those terms and this License would be to refrain entirely from conveying the Program. 13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such. 14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version. 15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. 16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. 17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the program's name and a brief idea of what it does.> Copyright (C) <year> <name of author>

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

<program> Copyright (C) <year> <name of author> This program comes with ABSOLUTELY NO WARRANTY; for details type 'show w'. This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see <http://www.gnu.org/licenses/>.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read <http://www.gnu.org/philosophy/why-not-lgpl.html>.

52.2 GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc. <http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed. 0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference. 1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License. 2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies. 3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document. 4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

* A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission. * B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement. * C. State on the Title page the name of the publisher of the Modified Version, as the publisher. * D. Preserve all the copyright notices of the Document. * E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices. * F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below. * G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice. * H. Include an unaltered copy of this License. * I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence. * J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission. * K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein. * L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles. * M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version. * N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section. * O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version. 5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements". 6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document. 7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate. 8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title. 9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it. 10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document. 11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing. ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright (C) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with ... Texts." line with this:

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

52.3 GNU Lesser General Public License

GNU LESSER GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

This version of the GNU Lesser General Public License incorporates the terms and conditions of version 3 of the GNU General Public License, supplemented by the additional permissions listed below. 0. Additional Definitions.

As used herein, “this License” refers to version 3 of the GNU Lesser General Public License, and the “GNU GPL” refers to version 3 of the GNU General Public License.

“The Library” refers to a covered work governed by this License, other than an Application or a Combined Work as defined below.

An “Application” is any work that makes use of an interface provided by the Library, but which is not otherwise based on the Library. Defining a subclass of a class defined by the Library is deemed a mode of using an interface provided by the Library.

A “Combined Work” is a work produced by combining or linking an Application with the Library. The particular version of the Library with which the Combined Work was made is also called the “Linked Version”.

The “Minimal Corresponding Source” for a Combined Work means the Corresponding Source for the Combined Work, excluding any source code for portions of the Combined Work that, considered in isolation, are based on the Application, and not on the Linked Version.

The “Corresponding Application Code” for a Combined Work means the object code and/or source code for the Application, including any data and utility programs needed for reproducing the Combined Work from the Application, but excluding the System Libraries of the Combined Work. 1. Exception to Section 3 of the GNU GPL.

You may convey a covered work under sections 3 and 4 of this License without being bound by section 3 of the GNU GPL. 2. Conveying Modified Versions.

If you modify a copy of the Library, and, in your modifications, a facility refers to a function or data to be supplied by an Application that uses the facility (other than as an argument passed when the facility is invoked), then you may convey a copy of the modified version:

* a) under this License, provided that you make a good faith effort to ensure that, in the event an Application does not supply the function or data, the facility still operates, and performs whatever part of its purpose remains meaningful, or * b) under the GNU GPL, with none of the additional permissions of this License applicable to that copy.

3. Object Code Incorporating Material from Library Header Files.

The object code form of an Application may incorporate material from a header file that is part of the Library. You may convey such object code under terms of your choice, provided that, if the incorporated material is not limited to numerical parameters, data structure layouts and accessors, or small macros, inline functions and templates (ten or fewer lines in length), you do both of the following:

* a) Give prominent notice with each copy of the object code that the Library is used in it and that the Library and its use are covered by this License. * b) Accompany the object code with a copy of the GNU GPL and this license document.

4. Combined Works.

You may convey a Combined Work under terms of your choice that, taken together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and reverse engineering for debugging such modifications, if you also do each of the following:

* a) Give prominent notice with each copy of the Combined Work that the Library is used in it and that the Library and its use are covered by this License. * b) Accompany the Combined Work with a copy of the GNU GPL and this license document. * c) For a Combined Work that displays copyright notices during execution, include the copyright notice for the Library among these notices, as well as a reference directing the user to the copies of the GNU GPL and this license document. * d) Do one of the following: o 0) Convey the Minimal Corresponding Source under the terms of this License, and the Corresponding Application Code in a form suitable for, and under terms that permit, the user to recombine or relink the Application with a modified version of the Linked Version to produce a modified Combined Work, in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source. o 1) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (a) uses at run time a copy of the Library already present on the user’s computer system, and (b) will operate properly with a modified version of the Library that is interface-compatible with the Linked Version. * e) Provide Installation Information, but only if you would otherwise be required to provide such information under section 6 of the GNU GPL, and only to the extent that such information is necessary to install and execute a modified version of the Combined Work produced by recombining or relinking the Application with a modified version of the Linked Version. (If you use option 4d0, the Installation Information must accompany the Minimal Corresponding Source and Corresponding Application Code. If you use option 4d1, you must provide the Installation Information in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.)

5. Combined Libraries.

You may place library facilities that are a work based on the Library side by side in a single library together with other library facilities that are not Applications and are not covered by this License, and convey such a combined library under terms of your choice, if you do both of the following:

* a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities, conveyed under the terms of this License. * b) Give prominent notice with the combined library that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

6. Revised Versions of the GNU Lesser General Public License.

The Free Software Foundation may publish revised and/or new versions of the GNU Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library as you received it specifies that a certain numbered version of the GNU Lesser General Public License “or any later version” applies to it, you have the option of following the terms and conditions either of that published version or of any later version published by the Free Software Foundation. If the Library as you received it does not specify a version number of the GNU Lesser General Public License, you may choose any version of the GNU Lesser General Public License ever published by the Free Software Foundation.

If the Library as you received it specifies that a proxy can decide whether future versions of the GNU Lesser General Public License shall apply, that proxy’s public statement of acceptance of any version is permanent authorization for you to choose that version for the Library.