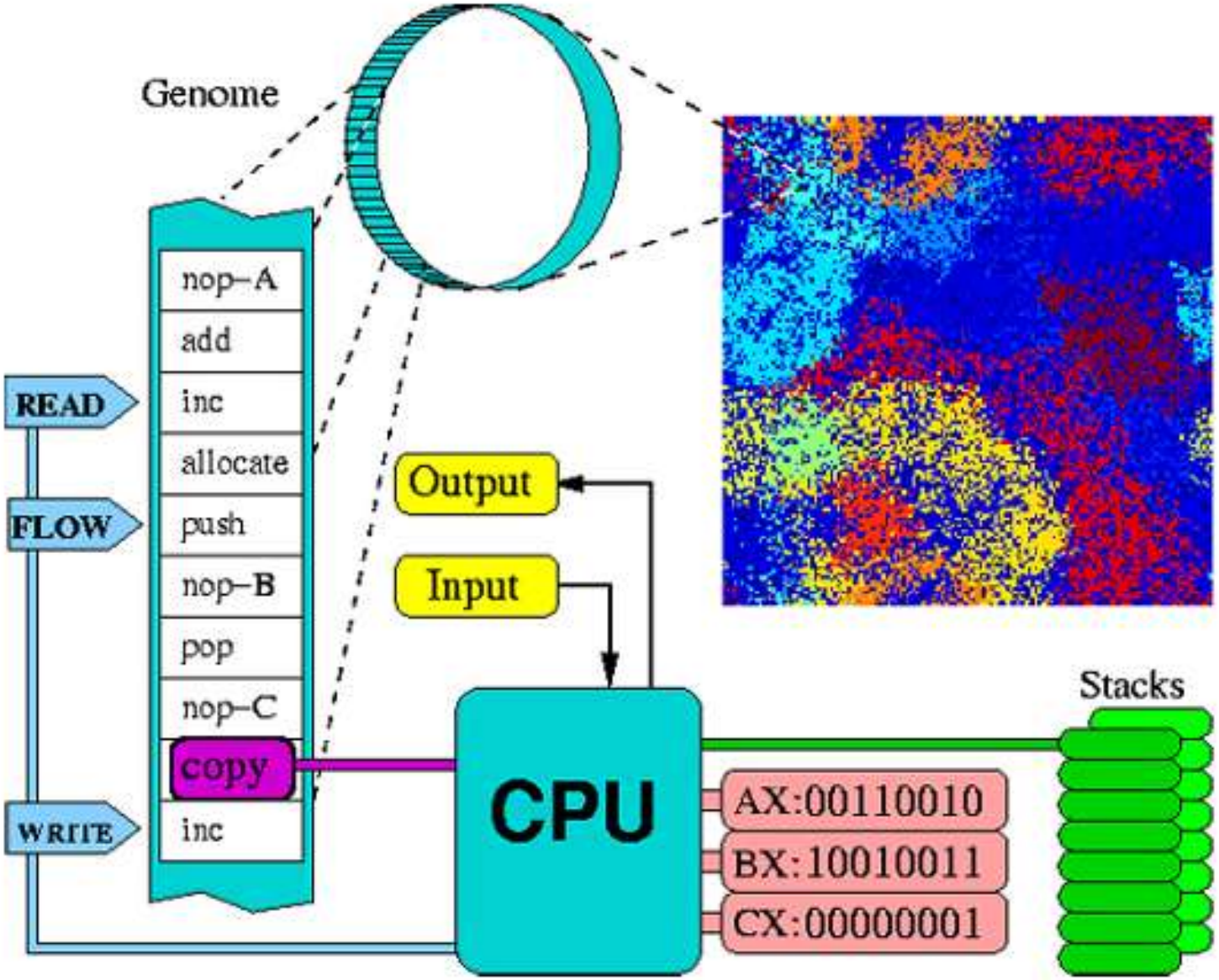
Instruction Processing in Computer Architecture



Mika Waterman

First Edition, 2012

ISBN 978-81-323-3476-7

Published by:
Research World
4735/22 Prakashdeep Bldg,
Ansari Road, Darya Ganj,
Delhi - 110002
Email: info@wtbooks.com

Table of Contents

Chapter-1

Instruction Set

An **instruction set**, or **instruction set architecture** (ISA), is the part of the computer architecture related to programming, including the native data types, instructions, registers, addressing modes, memory architecture, interrupt and exception handling, and external I/O. An ISA includes a specification of the set of opcodes (machine language), and the native commands implemented by a particular processor.

Instruction set architecture is distinguished from the microarchitecture, which is the set of processor design techniques used to implement the instruction set. Computers with different microarchitectures can share a common instruction set. For example, the Intel Pentium and the AMD Athlon implement nearly identical versions of the x86 instruction set, but have radically different internal designs.

This concept can be extended to unique ISAs like TIMI (Technology-Independent Machine Interface) present in the IBM System/38 and IBM AS/400. TIMI is an ISA that is implemented by low-level software translating TIMI code into "native" machine code, and functionally resembles what is now referred to as a virtual machine. It was designed to increase the longevity of the platform and applications written for it, allowing the entire platform to be moved to very different hardware without having to modify any software except that which translates TIMI into native machine code, and the code that implements services used by the resulting native code. This allowed IBM to move the AS/400 platform from an older CISC architecture to the newer POWER architecture without having to rewrite or recompile any parts of the OS or software associated with it other than the aforementioned low-level code. Some virtual machines that support bytecode for Smalltalk, the Java virtual machine, and Microsoft's Common Language Runtime virtual machine as their ISA implement it by translating the bytecode for commonly-used code paths into native machine code, and executing less-frequently-used code paths by interpretation; Transmeta implemented the x86 instruction set atop VLIW processors in the same fashion.

Machine language

Machine language is built up from discrete *statements* or *instructions*. On the processing architecture, a given instruction may specify:

- Particular registers for arithmetic, addressing, or control functions
- Particular memory locations or offsets
- Particular addressing modes used to interpret the operands

More complex operations are built up by combining these simple instructions, which (in a von Neumann architecture) are executed sequentially, or as otherwise directed by control flow instructions.

Instruction types

Some operations available in most instruction sets include:

- Data handling and Memory operations
  - **set** a register (a temporary "scratchpad" location in the CPU itself) to a fixed constant value
  - **move** data from a memory location to a register, or vice versa. This is done to obtain the data to perform a computation on it later, or to store the result of a computation.
  - **read** and **write** data from hardware devices
- Arithmetic and Logic
  - **add**, **subtract**, **multiply**, or **divide** the values of two registers, placing the result in a register
  - perform bitwise operations, taking the **conjunction** and **disjunction** of corresponding bits in a pair of registers, or the **negation** of each bit in a register
  - **compare** two values in registers (for example, to see if one is less, or if they are equal)
- Control flow
  - **branch** to another location in the program and execute instructions there
  - **conditionally branch** to another location if a certain condition holds
  - **indirectly branch** to another location, but save the location of the next instruction as a point to return to (a *call*)

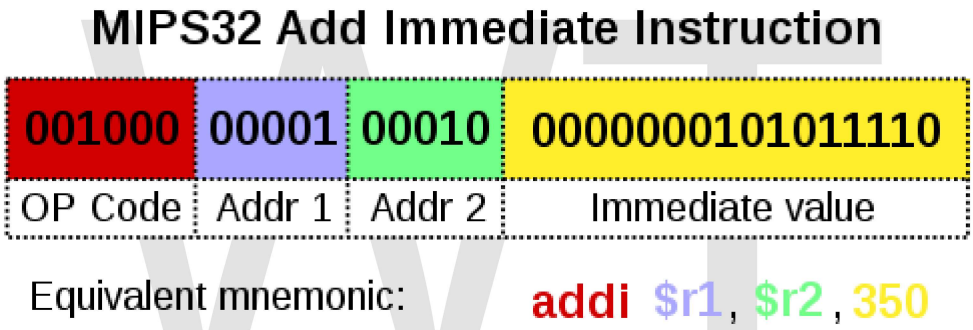
Complex instructions

Some computers include "complex" instructions in their instruction set. A single "complex" instruction does something that may take many instructions on other computers. Such instructions are typified by instructions that take multiple steps, control multiple functional units, or otherwise appear on a larger scale than the bulk of simple instructions implemented by the given processor. Some examples of "complex" instructions include:

- saving many registers on the stack at once
- moving large blocks of memory
- complex and/or floating-point arithmetic (sine, cosine, square root, etc.)
- performing an atomic test-and-set instruction
- instructions that combine ALU with an operand from memory rather than a register

A complex instruction type that has become particularly popular recently is the SIMD or Single-Instruction Stream Multiple-Data Stream operation or vector instruction, an operation that performs the same arithmetic operation on multiple pieces of data at the same time. SIMD have the ability of manipulating large vectors and matrices in minimal time. SIMD instructions allow easy parallelization of algorithms commonly involved in sound, image, and video processing. Various SIMD implementations have been brought to market under trade names such as MMX, 3DNow! and AltiVec.

Parts of an instruction



One instruction may have several fields, which identify the logical operation to be done, and may also include source and destination addresses and constant values. This is the MIPS "Add" instruction which allows selection of source and destination registers and inclusion of a small constant.

On traditional architectures, an instruction includes an opcode specifying the operation to be performed, such as "add contents of memory to register", and zero or more operand specifiers, which may specify registers, memory locations, or literal data. The operand specifiers may have addressing modes determining their meaning or may be in fixed fields.

In very long instruction word (VLIW) architectures, which include many microcode architectures, multiple simultaneous opcodes and operands are specified in a single instruction.

Some exotic instruction sets do not have an opcode field (such as Transport Triggered Architectures (TTA) or the Forth virtual machine), only operand(s). Other unusual "0-operand" instruction sets lack any operand specifier fields, such as some stack machines including NOSC .

Instruction length

The size or length of an instruction varies widely, from as little as four bits in some microcontrollers to many hundreds of bits in some VLIW systems. Processors used in personal computers, mainframes, and supercomputers have instruction sizes between 8 and 64 bits. Within an instruction set, dfferent instructions may have different lengths. In some architectures, notably most Reduced Instruction Set Computers (RISC), instructions are a fixed length, typically corresponding with that architecture's word size. In other architectures, instructions have variable length, typically integral multiples of a byte or a halfword.

Representation

The instructions constituting a program are rarely specified using their internal, numeric form; they may be specified by programmers using an assembly language or, more commonly, may be generated by compilers.

Design

The design of instruction sets is a complex issue. There were two stages in history for the microprocessor. The first was the CISC (Complex Instruction Set Computer) which had many different instructions. In the 1970s, however, places like IBM did research and found that many instructions in the set could be eliminated. The result was the RISC (Reduced Instruction Set Computer), an architecture which uses a smaller set of instructions. A simpler instruction set may offer the potential for higher speeds, reduced processor size, and reduced power consumption. However, a more complex set may optimize common operations, improve memory/cache efficiency, or simplify programming.

Some instruction set designers reserve one or more opcodes for some kind of software interrupt. For example, MOS Technology 6502 uses 00_H, Zilog Z80 uses the eight codes C7,CF,D7,DF,E7,EF,F7,FF_H while Motorola 68000 use codes in the range A000..$AFFF_H$.

Fast virtual machines are much easier to implement if an instruction set meets the Popek and Goldberg virtualization requirements.

The NOP slide used in Immunity Aware Programming is much easier to implement if the "unprogrammed" state of the memory is interpreted as a NOP.

On systems with multiple processors, non-blocking synchronization algorithms are much easier to implement if the instruction set includes support for something like "fetch-and-increment" or "load linked/store conditional (LL/SC)" or "atomic compare and swap".

Instruction set implementation

Any given instruction set can be implemented in a variety of ways. All ways of implementing an instruction set give the same programming model, and they all are able to run the same binary executables. The various ways of implementing an instruction set give different tradeoffs between cost, performance, power consumption, size, etc.

When designing the microarchitecture of a processor, engineers use blocks of "hard-wired" electronic circuitry (often designed separately) such as adders, multiplexers, counters, registers, ALUs etc. Some kind of register transfer language is then often used to describe the decoding and sequencing of each instruction of an ISA using this physical microarchitecture. There are two basic ways to build a control unit to implement this description (although many designs use middle ways or compromises):

1. Early computer designs and some of the simpler RISC computers "hard-wired" the complete instruction set decoding and sequencing (just like the rest of the microarchitecture).
2. Other designs employ microcode routines and/or tables to do this—typically as on chip ROMs and/or PLAs (although separate RAMs have been used historically).

There are also some new CPU designs which compile the instruction set to a writable RAM or FLASH inside the CPU (such as the Rekursiv processor and the Imsys Cjip), or an FPGA (reconfigurable computing). The Western Digital MCP-1600 is an older example, using a dedicated, separate ROM for microcode.

An ISA can also be emulated in software by an interpreter. Naturally, due to the interpretation overhead, this is slower than directly running programs on the emulated hardware, unless the hardware running the emulator is an order of magnitude faster. Today, it is common practice for vendors of new ISAs or microarchitectures to make software emulators available to software developers before the hardware implementation is ready.

Often the details of the implementation have a strong influence on the particular instructions selected for the instruction set. For example, many implementations of the instruction pipeline only allow a single memory load or memory store per instruction, leading to a load-store architecture (RISC). For another example, some early ways of implementing the instruction pipeline led to a delay slot.

The demands of high-speed digital signal processing have pushed in the opposite direction—forcing instructions to be implemented in a particular way. For example, in order to perform digital filters fast enough, the MAC instruction in a typical digital signal processor (DSP) must be implemented using a kind of Harvard architecture that can fetch an instruction and two data words simultaneously, and it requires a single-cycle multiply-accumulate multiplier.

Code density

In early computers, program memory was expensive, so minimizing the size of a program to make sure it would fit in the limited memory was often central. Thus the combined size of all the instructions needed to perform a particular task, the *code density*, was an important characteristic of any instruction set. Computers with high code density also often had (and have still) complex instructions for procedure entry, parameterized returns, loops etc. (therefore retroactively named *Complex Instruction Set Computers*, CISC). However, more typical, or frequent, "CISC" instructions merely combine a basic ALU operation, such as "add", with the access of one or more operands in memory (using addressing modes such as direct, indirect, indexed etc.). Certain architectures may allow two or three operands (including the result) directly in memory or may be able to perform functions such as automatic pointer increment etc. Software-implemented instruction sets may have even more complex and powerful instructions.

Reduced instruction-set computers, RISC, were first widely implemented during a period of rapidly-growing memory subsystems and sacrifice code density in order to simplify implementation circuitry and thereby try to increase performance via higher clock frequencies and more registers. RISC instructions typically perform only a single operation, such as an "add" of registers or a "load" from a memory location into a register; they also normally use a fixed instruction width, whereas a typical CISC instruction set has many instructions shorter than this fixed length. Fixed-width instructions are less complicated to handle than variable-width instructions for several reasons (not having to check whether an instruction straddles a cache line or virtual memory page boundary for instance), and are therefore somewhat easier to optimize for speed. However, as RISC computers normally require more and often longer instructions to implement a given task, they inherently make less optimal use of bus bandwidth and cache memories.

Minimal instruction set computers (MISC) are a form of stack machine, where there are few separate instructions (16-64), so that multiple instructions can be fit into a single machine word. These type of cores often take little silicon to implement, so they can be easily realized in an FPGA or in a multi-core form. Code density is similar to RISC; the increased instruction density is offset by requiring more of the primitive instructions to do a task.

There has been research into executable compression as a mechanism for improving code density. The mathematics of Kolmogorov complexity describes the challenges and limits of this.

Number of operands

Instruction sets may be categorized by the maximum number of operands *explicitly* specified in instructions.

(In the examples that follow, *a*, *b*, and *c* are (direct or calculated) addresses referring to memory cells, while *reg1* and so on refer to machine registers.)

- 0-operand (*zero address machines*), so called stack machines: All arithmetic operations take place using the top one or two positions on the stack; 1-operand push and pop instructions are used to access memory: **push** a, **push** b, **add**, **pop** c.
- 1-operand (*one address machines*), so called accumulator machines, include most early computers and many small microcontrollers: Most instructions specify a single explicit right operand (a register, a memory location, or a constant) with the implicit accumulator as both destination and left (or only) operand: **load** *a*, **add** *b*, **store** *c*. A related class is practical stack machines which often allow a single explicit operand in arithmetic instructions: **push** a, **add** b, **pop** c.
- 2-operand — many CISC and RISC machines fall under this category:
  - CISC — **load** *a,reg1*; **add** *reg1,b*; **store** *reg1,c*
  - RISC — Requiring explicit memory loads, the instructions would be: **load** *a,reg1*; **load** *b,reg2*; **add** *reg1,reg2*; **store** *reg2,c*
- 3-operand, allowing better reuse of data:.
  - CISC — It becomes either a single instruction: **add** *a,b,c*, or more typically: **move** *a,reg1*; **add** *reg1,b,c* as most machines are limited to two memory operands.
  - RISC — Due to the large number of bits needed to encode three registers, this scheme is typically not available in RISC processors using small 16-bit instructions; arithmetic instructions use registers only, so explicit 2-operand load/store instructions are needed: **load** *a,reg1*; **load** *b,reg2*; **add** *reg1+reg2->reg3*; **store** *reg3,c*;
- more operands—some CISC machines permit a variety of addressing modes that allow more than 3 operands (registers or memory accesses), such as the VAX "POLY" polynomial evaluation instruction.

Each instruction specifies some number of operands (registers, memory locations, or immediate values) *explicitly*. Some instructions give one or both operands implicitly, such as by being stored on top of the stack or in an implicit register. When some of the operands are given implicitly, the number of specified operands in an instruction is smaller than the arity of the operation. When a "destination operand" explicitly specifies the destination, the number of operand specifiers in an instruction is larger than the arity of the operation. Some instruction sets have different numbers of operands for different instructions.

List of ISAs

This list is far from comprehensive as old architectures are developed and new ones invented. There are many commercially available microprocessors and microcontrollers implementing ISAs. Customized ISAs are also quite common in some applications, e.g. ASIC, FPGA, and reconfigurable computing.

- List of instruction sets

ISAs implemented in hardware

- 4004, 4040
- 6800, 6502, 6809, 68HC11, 68HC08, etc.
- 8008, 8080, 8085, Z80, Z180, eZ80, etc.
- 8048, 8051, etc.
- Z8, eZ8, etc.
- Alpha
- ARM
- Burroughs B5000 series
- Burroughs B6000/B7000 series
- eSi-RISC
- IA-64 (Itanium)
- Mico32
- MIPS
- Motorola 68k
- PA-RISC
- IBM 700/7000 lines
    - IBM 701
    - IBM 702
    - IBM 704
    - IBM 7010
    - IBM 7030
    - IBM 7040/7044
    - IBM 7070/7072/7074
    - IBM 705/7080
    - IBM 709/7090/7094
- System/360 and upwards compatible successors
    - System/370
    - System/390
    - z/Architecture
- Power Architecture
    - POWER
    - PowerPC
- PDP-11
- SPARC
- SuperH
- TriCore™
- Transputer
- UNIVAC 1100/2200 series
- VAX
- x86
    - IA-32 (32-bit x86, first implemented in the Intel 80386)
    - x86-64 (64-bit superset of IA-32, first implemented in the AMD Opteron)
- EISC (AE32K)

ISAs commonly implemented in software with hardware incarnations

- p-Code (UCSD p-System Version III on Western Digital Pascal MicroEngine)
- Java virtual machine (ARM Jazelle, picoJava, JOP)
- FORTH
- MMIX, a teaching machine used in Donald Knuth's *The Art of Computer Programming*

ISAs only implemented in software

- Common Intermediate Language (CIL) - The assembly language and instruction set developed for the Common Language Infrastructure (CLI), the standard that is the foundation of the Common Language Runtime (CLR) in the .NET Framework and the open-source implementation Mono.

ISAs never implemented in hardware

- ALGOL object code
- SECD machine, a virtual machine used for some functional programming languages.
- Z-machine, a virtual machine originated by Infocom and used for text adventure games, and its successor Glulx

Chapter-2

Instruction Pipeline

Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

Basic five-stage pipeline in a RISC machine (IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Register write back). In the fourth clock cycle (the green column), the earliest instruction is in MEM stage, and the latest instruction has not yet entered the pipeline.

An **instruction pipeline** is a technique used in the design of computers and other digital electronic devices to increase their instruction throughput (the number of instructions that can be executed in a unit of time).

The fundamental idea is to split the processing of a computer instruction into a series of independent steps, with storage at the end of each step. This allows the computer's control circuitry to issue instructions at the processing rate of the slowest step, which is much faster than the time needed to perform all steps at once. The term pipeline refers to the fact that each step is carrying data at once (like water), and each step is connected to the next (like the links of a pipe.)

The origin of pipelining is thought to be either the ILLIAC II project or the IBM Stretch project though a simple version was used earlier in the Z1 in 1939 and the Z3 in 1941.

The IBM Stretch Project proposed the terms, "Fetch, Decode, and Execute" that became common usage.

Most modern CPUs are driven by a clock. The CPU consists internally of logic and register (flip flops). When the clock signal arrives, the flip flops take their new value and the logic then requires a period of time to decode the new values. Then the next clock pulse arrives and the flip flops again take their new values, and so on. By breaking the logic into smaller pieces and inserting flip flops between the pieces of logic, the delay before the logic gives valid outputs is reduced. In this way the clock period can be reduced. For example, the classic RISC pipeline is broken into five stages with a set of flip flops between each stage.

1. Instruction fetch
2. Instruction decode and register fetch
3. Execute
4. Memory access
5. Register write back

When a programmer (or compiler) writes assembly code, they make the assumption that each instruction is executed before execution of the subsequent instruction is begun. This assumption is invalidated by pipelining. When this causes a program to behave incorrectly, the situation is known as a hazard. Various techniques for resolving hazards such as forwarding and stalling exist.

A non-pipeline architecture is inefficient because some CPU components (modules) are idle while another module is active during the instruction cycle. Pipelining does not completely cancel out idle time in a CPU but making those modules work in parallel improves program execution significantly.

Processors with pipelining are organized inside into stages which can semi-independently work on separate jobs. Each stage is organized and linked into a 'chain' so each stage's output is fed to another stage until the job is done. This organization of the processor allows overall processing time to be significantly reduced.

A deeper pipeline means that there are more stages in the pipeline, and therefore, fewer logic gates in each stage. This generally means that the processor's frequency can be increased as the cycle time is lowered. This happens because there are fewer components in each stage of the pipeline, so the propagation delay is decreased for the overall stage.

Unfortunately, not all instructions are independent. In a simple pipeline, completing an instruction may require 5 stages. To operate at full performance, this pipeline will need to run 4 subsequent independent instructions while the first is completing. If 4 instructions that do not depend on the output of the first instruction are not available, the pipeline control logic must insert a stall or wasted clock cycle into the pipeline until the dependency is resolved. Fortunately, techniques such as forwarding can significantly reduce the cases where stalling is required. While pipelining can in theory increase

performance over an unpipelined core by a factor of the number of stages (assuming the clock frequency also scales with the number of stages), in reality, most code does not allow for ideal execution.

Advantages and disadvantages

Pipelining does not help in all cases. There are several possible disadvantages. An instruction pipeline is said to be *fully pipelined* if it can accept a new instruction every clock cycle. A pipeline that is not fully pipelined has wait cycles that delay the progress of the pipeline.

***Advantages of Pipelining**:*

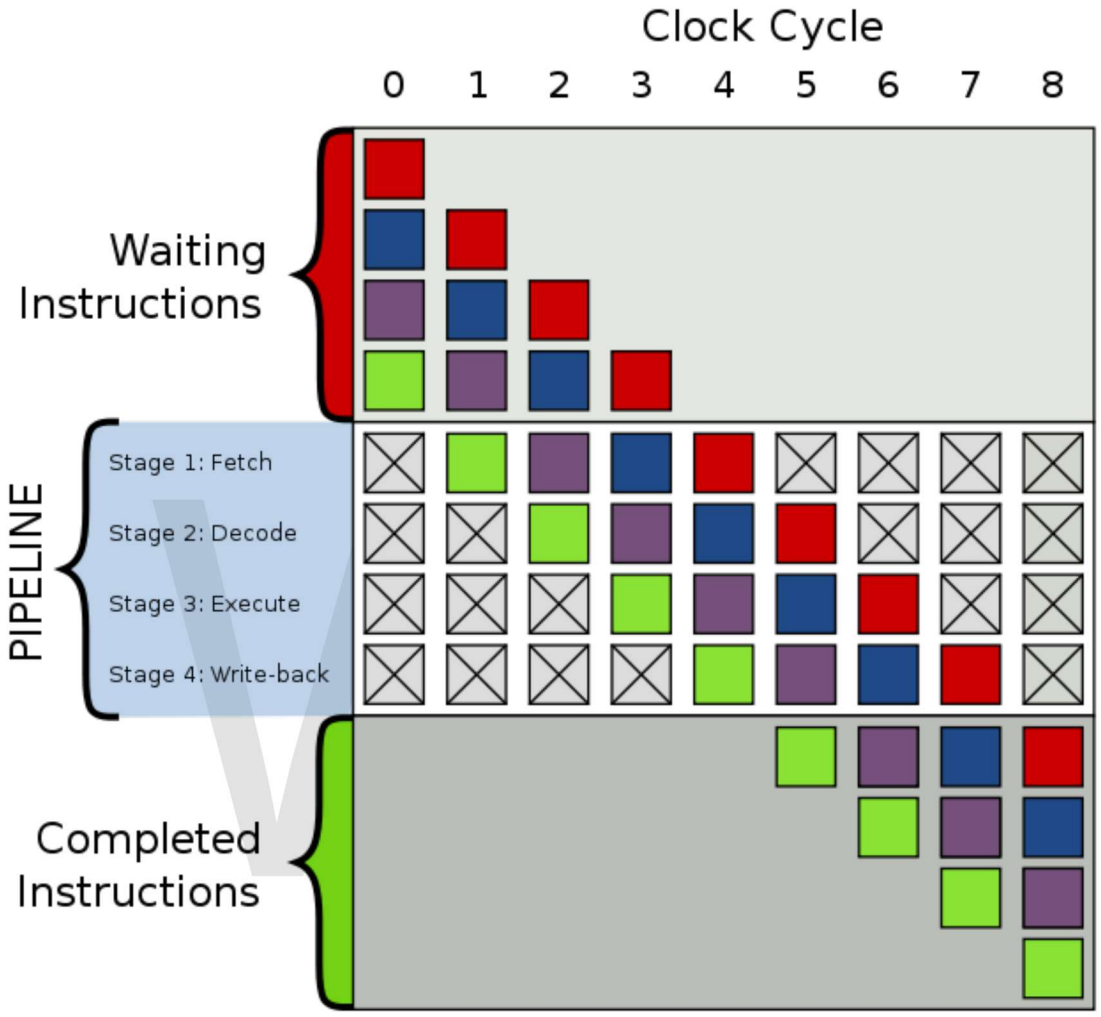
1. The cycle time of the processor is reduced, thus increasing instruction issue-rate in most cases.
2. Some combinational circuits such as adders or multipliers can be made faster by adding more circuitry. If pipelining is used instead, it can save circuitry vs. a more complex combinational circuit.

***Disadvantages of Pipelining**:*

1. A non-pipelined processor executes only a single instruction at a time. This prevents branch delays (in effect, every branch is delayed) and problems with serial instructions being executed concurrently. Consequently the design is simpler and cheaper to manufacture.
2. The instruction latency in a non-pipelined processor is slightly lower than in a pipelined equivalent. This is because extra flip flops must be added to the data path of a pipelined processor.
3. A non-pipelined processor will have a stable instruction bandwidth. The performance of a pipelined processor is much harder to predict and may vary more widely between different programs.

Examples

Generic pipeline



Generic 4-stage pipeline; the colored boxes represent instructions independent of each other

To the right is a generic pipeline with four stages:

1. Fetch
2. Decode
3. Execute
4. Write-back

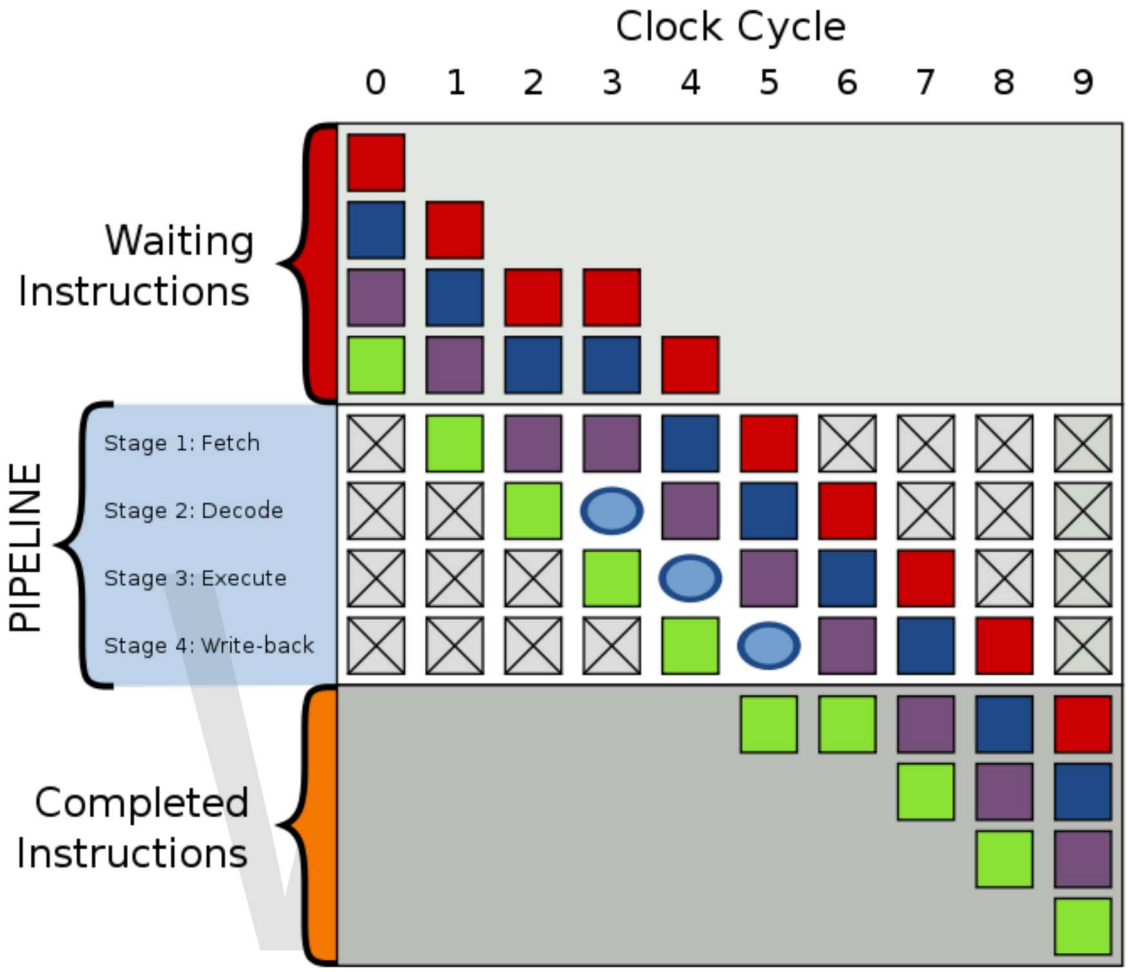
(for lw and sw memory is accessed after execute stage)

The top gray box is the list of instructions waiting to be executed; the bottom gray box is the list of instructions that have been completed; and the middle white box is the pipeline.

Execution is as follows:

Time	Execution
0	Four instructions are awaiting to be executed
1	• the green instruction is fetched from memory
2	• the green instruction is decoded • the purple instruction is fetched from memory
3	• the green instruction is executed (actual operation is performed) • the purple instruction is decoded • the blue instruction is fetched
4	• the green instruction's results are written back to the register file or memory • the purple instruction is executed • the blue instruction is decoded • the red instruction is fetched
5	• the green instruction is completed • the purple instruction is written back • the blue instruction is executed • the red instruction is decoded
6	• The purple instruction is completed • the blue instruction is written back • the red instruction is executed
7	• the blue instruction is completed • the red instruction is written back
8	• the red instruction is completed
9	All instructions are executed

Bubble



A bubble in cycle 3 delays execution

When a "hiccup" in execution occurs, a "bubble" is created in the pipeline in which nothing useful happens. In cycle 2, the fetching of the purple instruction is delayed and the decoding stage in cycle 3 now contains a bubble. Everything "behind" the purple instruction is delayed as well but everything "ahead" of the purple instruction continues with execution.

Clearly, when compared to the execution above, the bubble yields a total execution time of 8 clock ticks instead of 7.

Bubbles are like stalls, in which nothing useful will happen for the fetch, decode, execute and writeback. It can be completed with a nop code.

Example 1

A typical instruction to add two numbers might be `ADD A, B, C`, which adds the values found in memory locations A and B, and then puts the result in memory location C. In a pipelined processor the pipeline controller would break this into a series of tasks similar to:

```
LOAD R1, A
LOAD R2, B
ADD R3, R1, R2
STORE C, R3
LOAD next instruction
```

The locations 'R1', 'R2' and 'R3' are registers in the CPU. The values stored in memory locations labeled 'A' and 'B' are loaded (copied) into the R1 and R2 registers, then added, and the result (which is in register R3) is stored in a memory location labeled 'C'.

In this example the pipeline is three stages long- load, execute, and store. Each of the steps are called **pipeline stages**.

On a non-pipelined processor, only one stage can be working at a time so the entire instruction has to complete before the next instruction can begin. On a pipelined processor, all of the stages can be working at once on different instructions. So when this instruction is at the execute stage, a second instruction will be at the decode stage and a 3rd instruction will be at the fetch stage.

Pipelining doesn't reduce the time it takes to complete an instruction; it increases the number of instructions that can be processed at once and reduces the delay between completed instructions. The more pipeline stages a processor has, the more instructions it can be working on at once and the less of a delay there is between completed instructions. Every microprocessor manufactured today uses at least 2 stages of pipeline. (The Atmel AVR and the PIC microcontroller each have a 2 stage pipeline.) Intel Pentium 4 processors have 20 stage pipelines.

Example 2

To better visualize the concept, we can look at a theoretical 3-stage pipeline:

Stage	Description
Load	Read instruction from memory
Execute	Execute instruction
Store	Store result in memory and/or registers

and a pseudo-code assembly listing to be executed:

```
LOAD  A, #40      ; load 40 in A
```

```
MOVE  B, A        ; copy A in B
ADD   B, #20      ; add 20 to B
STORE 0x300, B    ; store B into memory cell 0x300
```

This is how it would be executed:

Clock 1

Load	Execute	Store
LOAD		

The LOAD instruction is fetched from memory.

Clock 2

Load	Execute	Store
MOVE	LOAD	

The LOAD instruction is executed, while the MOVE instruction is fetched from memory.

Clock 3

Load	Execute	Store
ADD	MOVE	LOAD

The LOAD instruction is in the Store stage, where its result (the number 40) will be stored in the register A. In the meantime, the MOVE instruction is being executed. Since it must move the contents of A into B, it must wait for the ending of the LOAD instruction.

Clock 4

Load	Execute	Store
STORE	ADD	MOVE

The STORE instruction is loaded, while the MOVE instruction is finishing off and the ADD is calculating.

And so on. Note that, sometimes, an instruction will depend on the result of another one (like our MOVE example). When more than one instruction references a particular location for an operand, either reading it (as an input) or writing it (as an output), executing those instructions in an order different from the original program order can lead to hazards (mentioned above). There are several established techniques for either preventing hazards from occurring, or working around them if they do.

Complications

Many designs include pipelines as long as 7, 10 and even 20 stages (like in the Intel Pentium 4). The later "Prescott" and "Cedar Mill" Pentium 4 cores (and their Pentium D derivatives) had a 31-stage pipeline, the longest in mainstream consumer computing. The Xelerator X10q has a pipeline more than a thousand stages long. The downside of a long pipeline is that when a program branches, the processor cannot know where to fetch the next instruction from and must wait until the branch instruction finishes, leaving the pipeline behind it empty. In the extreme case, the performance of a pipelined processor could theoretically approach that of an un-pipelined processor, or even slightly worse if all but one pipeline stages are idle and a small overhead is present between stages. Branch prediction attempts to alleviate this problem by guessing whether the branch will be taken or not and speculatively executing the code path that it predicts will be taken. When its predictions are correct, branch prediction avoids the penalty associated with branching. However, branch prediction itself can end up exacerbating the problem if branches are predicted poorly, as the incorrect code path which has begun execution must be flushed from the pipeline before resuming execution at the correct location.

In certain applications, such as supercomputing, programs are specially written to branch rarely and so very long pipelines can speed up computation by reducing cycle time. If branching happens constantly, re-ordering branches such that the more likely to be needed instructions are placed into the pipeline can significantly reduce the speed losses associated with having to flush failed branches. Programs such as gcov can be used to examine how often particular branches are actually executed using a technique known as coverage analysis; however such analysis is often a last resort for optimization.

Self-Modifying Programs: Because of the instruction pipeline, code that the processor loads will not immediately execute. Due to this, updates in the code very near the current location of execution may not take effect because they are already loaded into the Prefetch Input Queue. Instruction caches make this phenomenon even worse. This is only relevant to self-modifying programs.

Mathematical pipelines: Mathematical or arithmetic pipelines are different from instructional pipelines, in that when mathematically processing large arrays or vectors, a particular mathematical process, such as a multiply is repeated many thousands of times. In this environment, an instruction need only kick off an event whereby the arithmetic logic unit (which is pipelined) takes over, and begins its series of calculations. Most of these circuits can be found today in math processors and math processing sections of CPUs like the Intel Pentium line.

History

Math processing (super-computing) began in earnest in the late 1970s as Vector Processors and Array Processors. Usually very large bulky super-computing machines that needed special environments and super-cooling of the cores. One of the early super computers was the Cyber series built by Control Data Corporation. Its main architect was

Seymour Cray, who later resigned from CDC to head up Cray Research. Cray developed the XMP line of super computers, using pipelining for both multiply and add/subtract functions. Later, Star Technologies took pipelining to another level by adding parallelism (several pipelined functions working in parallel), developed by their engineer, Roger Chen. In 1984, Star Technologies made another breakthrough with the pipelined divide circuit, developed by James Bradley. By the mid 1980s, super-computing had taken off with offerings from many different companies around the world.

Chapter-3

Hazard (Computer Architecture)

Hazards are problems with the instruction pipeline in central processing unit (CPU) microarchitectures that potentially result in incorrect computation. There are typically three types of hazards:

- data hazards
- structural hazards
- control hazards (branching hazards)

There are several methods used to deal with hazards, including pipeline stalls (pipeline bubbling), register forwarding, and in the case of out-of-order execution, the scoreboarding method and the Tomasulo algorithm.

Background

Instructions in a pipelined processor are performed in several stages, so that at any given time several instructions are being processed in the various stages of the pipeline, such as fetch and execute. There are many different instruction pipeline microarchitectures, and instructions may be executed out-of-order. A hazard occurs when two or more of these simultaneous (possibly out of order) instructions conflict.

Types

Data hazards

Data hazards occur when instructions that exhibit data dependence modify data in different stages of a pipeline. Ignoring potential data hazards can result in race conditions (sometimes known as race hazards). There are three situations in which a data hazard can occur:

1. read after write (RAW), a *true dependency*
2. write after read (WAR)
3. write after write (WAW)

consider two instructions i and j, with i occurring before j in program order.

Read After Write (RAW)

(j tries to read a source before i writes to it) A read after write (RAW) data hazard refers to a situation where an instruction refers to a result that has not yet been calculated or retrieved. This can occur because even though an instruction is executed after a previous instruction, the previous instruction has not been completely processed through the pipeline.

Example

For example:

```
i1. R2 <- R1 + R3
i2. R4 <- R2 + R3
```

The first instruction is calculating a value to be saved in register 2, and the second is going to use this value to compute a result for register 4. However, in a pipeline, when we fetch the operands for the 2nd operation, the results from the first will not yet have been saved, and hence we have a data dependency.

We say that there is a data dependency with instruction 2, as it is dependent on the completion of instruction 1.

Write After Read (WAR)

(j tries to write a destination before it is read by i) A write after read (WAR) data hazard represents a problem with concurrent execution.

Example

For example:

```
i1. R4 <- R1 + R3
i2. R3 <- R1 + R2
```

If we are in a situation that there is a chance that i2 may be completed before i1 (i.e. with concurrent execution) we must ensure that we do not store the result of register 3 before i1 has had a chance to fetch the operands.

Write After Write (WAW)

(j tries to write an operand before it is written by i) A write after write (WAW) data hazard may occur in a concurrent execution environment.

Example

For example:

```
i1. R2 <- R4 + R7
i2. R2 <- R1 + R2
```

We must delay the WB (Write Back) of i2 until the execution of i1.

Structural hazards

A structural hazard occurs when a part of the processor's hardware is needed by two or more instructions at the same time. A canonical example is a single memory unit that is accessed both in the fetch stage where an instruction is retrieved from memory, and the memory stage where data is written and/or read from memory. They can often be resolved by separating the component into orthogonal units (such as separate caches) or bubbling the pipeline.

Control hazards (branch hazards)

Branching hazards (also known as control hazards) occur with branches. On many instruction pipeline microarchitectures, the processor will not know the outcome of the branch when it needs to insert a new instruction into the pipeline (normally the *fetch* stage).

Eliminating hazards

Generic

Pipeline bubbling

Bubbling the pipeline, also known as a *pipeline break* or a *pipeline stall*, is a method for preventing data, structural, and branch hazards from occurring. As instructions are fetched, control logic determines whether a hazard could/will occur. If this is true, then the control logic inserts NOPs into the pipeline. Thus, before the next instruction (which would cause the hazard) is executed, the previous one will have had sufficient time to complete and prevent the hazard. If the number of NOPs is equal to the number of stages in the pipeline, the processor has been cleared of all instructions and can proceed free from hazards. This is called *flushing the pipeline*. All forms of stalling introduce a delay before the processor can resume execution.

Data hazards

There are several main solutions and algorithms used to resolve data hazards:

- insert a *pipeline bubble* whenever a read after write (RAW) dependency is encountered, guaranteed to increase latency, or
- utilize out-of-order execution to potentially prevent the need for pipeline bubbles
- utilize *register forwarding* to use data from later stages in the pipeline

In the case of out-of-order execution, the algorithm used can be:

- scoreboarding, in which case a *pipeline bubble* will only be needed when there is no functional unit available
- the Tomasulo algorithm, which utilizes register renaming allowing the continual issuing of instructions

We can delegate the task of removing data dependencies to the compiler, which can fill in an appropriate number of `NOP` instructions between dependent instructions to ensure correct operation, or re-order instructions where possible.

Register forwarding

Forwarding involves feeding output data into a previous stage of the pipeline. Forwarding is implemented by feeding back the output of an instruction into the previous stage(s) of the pipeline as soon as the output of that instruction is available.

Example

For instance, let's say we want to write the value 3 to register 1, (which already contains a 6), and then add 7 to register 1 and store the result in register 2, i.e.:

Instruction 0: Register 1 = ***6***
Instruction 1: Register 1 = ***3***
Instruction 2: Register 2 = Register 1 + 7 = ***10***

Following execution, register 2 should contain the value **10**. However, if Instruction 1 (write **3** to register 1) does not completely exit the pipeline before Instruction 2 starts execution, it means that Register 1 does not contain the value **3** when Instruction 2 performs its addition. In such an event, Instruction 2 adds **7** to the old value of register 1 (**6**), and so register 2 would contain **13** instead, i.e:

Instruction 0: Register 1 = ***6***
Instruction 2: Register 2 = Register 1 + 7 = ***13***
Instruction 1: Register 1 = ***3***

This error occurs because Instruction 2 reads Register 1 before Instruction 1 has committed/stored the result of its write operation to Register 1. So when Instruction 2 is reading the contents of Register 1, register 1 still contains **6**, *not* **3**.

Forwarding (described below) helps correct such errors by depending on the fact that the output of Instruction 1 (which is **3**) can be used by subsequent instructions *before* the value **3** is committed to/stored in Register 1.

Forwarding applied to our example means that *we do not wait to commit/store the output of Instruction 1 in Register 1 (in this example, the output is **3**) before making that output available to the subsequent instruction (in this case, Instruction 2).* The effect is that Instruction 2 uses the correct (the more recent) value of Register 1: the commit/store was made immediately and not pipelined.

With forwarding enabled, the ID/EX or Instruction Decode/Execution stage of the pipeline now has two inputs: the value read from the register specified (in this example, the value **6** from Register 1), and the new value of Register 1 (in this example, this value is **3**) which is sent from the next stage (EX/MEM) or Instruction Execute/Memory Access. Additional control logic is used to determine which input to use.

Control hazards (branch hazards)

To avoid control hazards microarchitectures can:

- insert a *pipeline bubble* (discussed above), guaranteed to increase latency, or
- use branch prediction and essentially guesstimate which instructions to insert, in which case a *pipeline bubble* will only be needed in the case of an incorrect prediction

In the event that a branch causes a pipeline bubble after incorrect instructions have entered the pipeline, care must be taken to prevent any of the wrongly-loaded instructions from having any effect on the processor state excluding energy wasted processing them before they were discovered to be loaded incorrectly.

Chapter-4

Machine Code and Instruction Cycle

Machine code

Machine code or **machine language** is a system of instructions and data executed directly by a computer's central processing unit. Machine code may be regarded as a primitive (and cumbersome) programming language or as the lowest-level representation of a compiled and/or assembled computer program. Programs in interpreted languages are **not** represented by machine code however, although their *interpreter* (which may be seen as a processor executing the higher level program) often is. Machine code is sometimes called **native code** when referring to platform-dependent parts of language features or libraries. Machine code should not be confused with so called "bytecode", which is executed by an interpreter.

Machine code instructions

Every processor or processor family has its own machine code instruction set. Instructions are patterns of bits that by physical design correspond to different commands to the machine. The instruction set is thus specific to a class of processors using (much) the same architecture. Successor or derivative processor designs often include all the instructions of a predecessor and may add additional instructions. Occasionally a successor design will discontinue or alter the meaning of some instruction code (typically because it is needed for new purposes), affecting code compatibility to some extent; even nearly completely compatible processors may show slightly different behavior for some instructions but this is seldom a problem. Systems may also differ in other details, such as memory arrangement, operating systems, or peripheral devices; because a program normally relies on such factors, different systems will typically not run the same machine code, even when the same type of processor is used.

A machine code instruction set may have all instructions of the same length, or it may have variable-length instructions. How the patterns are organized varies strongly with the particular architecture and often also with the type of instruction. Most instructions have

one or more opcode fields which specifies the basic instruction type (such as arithmetic, logical, jump, etc) and the actual operation (such as add or compare) and other fields that may give the type of the operand(s), the addressing mode(s), the addressing offset(s) or index, or the actual value itself (such constant operands contained in an instruction are called *immediates*).

Programs

A computer program is a sequence of instructions that are executed by a CPU. While simple processors execute instructions one after the other, superscalar processors are capable of executing several instructions at once.

Program flow may be influenced by special 'jump' instructions that transfer execution to an instruction other than the following one. Conditional jumps are taken (execution continues at another address) or not (execution continues at the next instruction) depending on some condition.

Assembly languages

A much more readable rendition of machine language, called assembly language, uses mnemonic codes to refer to machine code instructions, rather than simply using the instructions' numeric values. For example, on the Zilog Z80 processor, the machine code `00000101`, which causes the CPU to decrement the `B` processor register, would be represented in assembly language as `DEC B`.

Example

The MIPS architecture provides a specific example for a machine code whose instructions are always 32 bits long. The general type of instruction is given by the *op* (operation) field, the highest 6 bits. J-type (jump) and I-type (immediate) instructions are fully specified by *op*. R-type (register) instructions include an additional field *funct* to determine the exact operation. The fields used in these types are:

```
   6      5     5     5     5      6 bits
[  op  |  rs |  rt |  rd |shamt| funct]  R-type
[  op  |  rs |  rt | address/immediate]  I-type
[  op  |       target address        ]  J-type
```

rs, *rt*, and *rd* indicate register operands; *shamt* gives a shift amount; and the *address* or *immediate* fields contain an operand directly.

For example adding the registers 1 and 2 and placing the result in register 6 is encoded:

```
[  op  |  rs |  rt |  rd |shamt| funct]
    0     1     2     6     0     32     decimal
 000000 00001 00010 00110 00000 100000   binary
```

Load a value into register 8, taken from the memory cell 68 cells after the location listed in register 3:

```
[  op   |  rs |  rt | address/immediate]
   35     3     8           68           decimal
 100011 00011 01000 00000 00001 000100   binary
```

Jumping to the address 1024:

```
[  op   |        target address        ]
    2                 1024              decimal
 000010 00000 00000 00000 10000 000000   binary
```

Relationship to microcode

In some computer architectures, the machine code is implemented by a more fundamental underlying layer of programs called microprograms, providing a common machine language interface across a line or family of different models of computer with widely different underlying dataflows. This is done to facilitate porting of machine language programs between different models. An example of this use is the IBM System/360 family of computers and their successors. With dataflow path widths of 8 bits to 64 bits and beyond, they nevertheless present a common architecture at the machine language level across the entire line.

Using a microcode layer to implement an emulator enables the computer to present the architecture of an entirely different computer. The System/360 line used this to allow porting programs from earlier IBM machines to the new family of computers, e.g. an IBM 1401/1440/1460 emulator on the IBM S/360 model 40.

Storing in memory

The Harvard architecture is a computer architecture with physically separate storage and signal pathways for the code (instructions) and data. Today, most processors implement such separate signal pathways for performance reasons but actually implement a Modified Harvard architecture, so they can support tasks like loading a program from disk storage as data and then executing it. Harvard architecture is contrasted to the Von Neumann architecture, where data and code are stored in the same memory.

From the point of view of a process, the *code space* is the part of its address space where code in execution is stored. In multi-threading environment, threads share code space along with data space, which reduces the overhead of context switching considerably as compared to process switching.

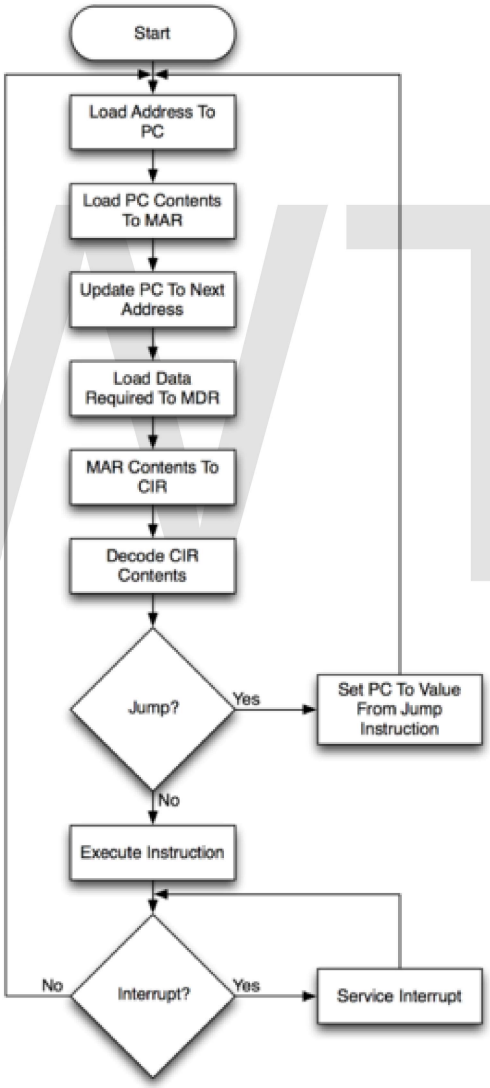
Readability by humans

It has been said that machine code is so unreadable that the Copyright Office cannot even identify whether a particular encoded program is an original work of authorship.

Hofstadter writes, "Looking at a program written in machine language is vaguely comparable to looking at a DNA molecule atom by atom."

Instruction cycle

An **instruction cycle** (sometimes called **fetch-and-execute cycle**, **fetch-decode-execute cycle**, or **FDX**) is the basic operation cycle of a computer. It is the process by which a computer retrieves a program instruction from its memory, determines what actions the instruction requires, and carries out those actions. This cycle is repeated continuously by the central processing unit (CPU), from bootup to when the computer is shut down.



A diagram of the Fetch Execute Cycle.

Circuits used

The circuits used in the CPU during the cycle are:

- **Program Counter (PC) -** an incrementing counter that keeps track of the memory address of which instruction is to be executed next
- **Memory Address Register (MAR) -** holds the address in memory of the next instruction to be executed
- **Memory Data Register (MDR) -** a two-way register that holds data fetched from memory (and ready for the CPU to process) or data waiting to be stored in memory
- **Current Instruction Register (CIR) -** a temporary holding ground for the instruction that has just been fetched from memory
- **Control Unit (CU) -** decodes the program instruction in the CIR, selecting machine resources such as a data source register and a particular arithmetic operation, and coordinates activation of those resources
- **Arithmetic logic unit (ALU) -** performs mathematical and logical operations

The time period during which one instruction is fetched from memory and executed when a computer is given an instruction in machine language. There are typically four stages of an instruction cycle that the CPU carries out: 1) Fetch the instruction from memory. 2) "Decode" the instruction. 3) "Read the effective address" from memory if the instruction has an indirect address. 4) "Execute" the instruction.

Instruction cycle

Each computer's CPU can have different cycles based on different instruction sets, but will be similar to the following cycle:

1. Fetch the instruction

The next instruction is fetched from the memory address that is currently stored in the Program Counter (PC), and stored in the Instruction register (IR). At the end of the fetch operation, the PC points to the next instruction that will be read at the next cycle.

Clock Pulse: T_0-T_1

2. Decode the instruction

The decoder interprets the instruction. During this cycle the instruction inside the IR (instruction register) gets decoded.

Clock Pulse: T_2

3. Read the effective address

In case of a memory instruction (direct or indirect) the execution phase will be in the next clock pulse. If the instruction has an indirect address, the effective address is read from main memory, and any required data is fetched from main memory to be processed and then placed into data registers(Clock Pulse: T_3). If the instruction is direct, nothing is done at this clock pulse. If this is an I/O instruction or a Register instruction, the operation is preformed (executed) at clock Pulse: T_3.

Clock Pulse: T_3

4. Execute the instruction

The CU passes the decoded information as a sequence of control signals to the relevant function units of the CPU to perform the actions required by the instruction such as reading values from registers, passing them to the ALU to perform mathematical or logic functions on them, and writing the result back to a register. If the ALU is involved, it sends a condition signal back to the CU.

Clock Pulse: T_3-T_6 (Up to T_6)

The result generated by the operation is stored in the main memory, or sent to an output device. Based on the condition of any feedback from the ALU, Program Counter may be updated to a different address from which the next instruction will be fetched.

The cycle is then repeated.

Initiating the cycle

The cycle starts immediately when power is applied to the system using an initial PC value that is predefined for the system architecture (in Intel IA-32 CPUs, for instance, the predefined PC value is `0xfffffff0`). Typically this address points to instructions in a read-only memory (ROM) which begin the process of loading the operating system. (That loading process is called *booting*.)

Fetch cycle

Step 1 of the Instruction Cycle is called the Fetch Cycle. These step is the same for each instruction. The fetch cycle processes the instruction from the instruction word which contains an opcode.

Decode

Step 2 of the instruction Cycle is called the decode. The opcode fetched from the memory is being decoded for the next steps and moved to the appropriate registers.

Read the effective address

Step 3 is deciding which operation it is. If this is a Memory operation - in this step the computer checks if it's a direct or indirect memory operation:

- **Direct memory instruction -** Nothing is being done.
- **Indirect memory instruction -** The effective address is being read from the memory.

If this is a I/O or Register instruction - the computer checks it's kind and execute the instruction.

Execute cycle

Step 4 of the Instruction Cycle is the Execute Cycle. These steps will change with each instruction.

The first step of the execute cycle is the Process-Memory. Data is transferred between the CPU and the I/O module. Next is the Data-Processing uses mathematical operations as well as logical operations in reference to data. Central alterations is the next step, is a sequence of operations, for example a jump operation. The last step is a combined operation from all the other steps.

The Fetch-Execute cycle in Transfer Notation

Expressed in register transfer notation:

$$MAR \leftarrow [PC]$$

$$MDR \leftarrow [Memory]_{MARaddress}; PC \leftarrow [PC] + 1$$ (Increment the PC for next cycle at the same time)

$$CIR \leftarrow [MDR]$$

The registers used above, besides the ones described earlier, are the Memory Address Register (MAR) and the Memory Data Register (MDR), which are used (at least conceptually) in the accessing of memory. Often, the MDR is expressed as the MBR (Memory Buffer Register).

Fetch and execute example (written in RTL - Register Transfer Language):
PC=0x5AF , AC=0x7EC3 , M[0x5AF]=0x932E , M[0x32E]=0x09AC , M[0x9AC]=0x8B9F.
T0 : AR = 0x5AF (PC)
T1 : IR = 0x932E (M[AR]) , PC=0x5BO
T2 : DECODE = ADD opCode 0x932E , AR=0x32E , I=1. (Indirect instruction)
T3 : AR = 0x9AC (M[AR])
T4 : DR = 0x8B9F
T5 : AC = 0x8B9F + 0x7EC3 = 0x0A62, E = 1 (carry out) , SC = 0

Summary: this example is for an ADD Instruction which made Indirect where:
T0-T1 is the Fetch operation.
T2 is the operation code Decode.
T3 Indirect Memory reference
T4-T5 Execute ADD operation

Chapter-5

Microcode

Microcode is a layer of hardware-level instructions and/or data structures involved in the implementation of higher level machine code instructions in many computers and other processors; it resides in a special high-speed memory and translates machine instructions into sequences of detailed circuit-level operations. It helps separate the machine instructions from the underlying electronics so that instructions can be designed and altered more freely. It also makes it feasible to build complex multi-step instructions while still reducing the complexity of the electronic circuitry compared to other methods. Writing microcode is often called **microprogramming** and the microcode in a particular processor implementation is sometimes called a **microprogram**.

Modern microcode is normally written by an engineer during the processor design phase and stored in a ROM and/or PLA structure, although machines exist which have some writable microcode in SRAM or flash memory. Microcode is generally not visible or changeable by a normal programmer, not even by an assembly programmer. Unlike machine code which often retains some compatibility among different processors in a family, microcode only runs on the exact electronic circuitry for which it is designed, as it constitutes an inherent part of the particular processor design itself.

More extensive microcoding has also been used to allow small and simple microarchitectures to emulate more powerful architectures with wider word length, more execution units and so on; a relatively simple way to achieve software compatibility between different products in a processor family.

Some hardware vendors, especially IBM, use the term as a synonym for *firmware*, so that all code in a device, whether microcode or machine code, is termed microcode (such as in a hard drive for instance, which typically contains both).

Overview

The elements composing a microprogram exist on a lower conceptual level than a normal application program. Each element is differentiated by the "micro" prefix to avoid confusion: microinstruction, microassembler, microprogrammer, microarchitecture, etc.

The microcode usually does not reside in the main memory, but in a special high speed memory, called the control store. It might be either read-only or read-write memory. In the latter case the microcode would be loaded into the control store from some other storage medium as part of the initialization of the CPU, and it could be altered to correct bugs in the instruction set, or to implement new machine instructions.

Microprograms consist of series of microinstructions. These microinstructions control the CPU at a very fundamental level of hardware circuitry. For example, a single typical microinstruction might specify the following operations:

- Connect Register 1 to the "A" side of the ALU
- Connect Register 7 to the "B" side of the ALU
- Set the ALU to perform two's-complement addition
- Set the ALU's carry input to zero
- Store the result value in Register 8
- Update the "condition codes" with the ALU status flags ("Negative", "Zero", "Overflow", and "Carry")
- Microjump to MicroPC nnn for the next microinstruction

To simultaneously control all processor's features in one cycle, the microinstruction is often wider than 50 bits, e.g., 128 bits on a 360/85 with an emulator feature. Microprograms are carefully designed and optimized for the fastest possible execution, since a slow microprogram would yield a slow machine instruction which would in turn cause all programs using that instruction to be slow.

The reason for microprogramming

Microcode was originally developed as a simpler method of developing the control logic for a computer. Initially CPU instruction sets were "hard wired". Each step needed to fetch, decode and execute the machine instructions (including any operand address calculations, reads and writes) was controlled directly by combinatorial logic and rather minimal sequential state machine circuitry. While very efficient, the need for powerful instruction sets with multi-step addressing and complex operations made such "hard-wired" processors difficult to design and debug; highly encoded and varied-length instructions can contribute to this as well, especially when very irregular encodings are used.

Microcode simplified the job by allowing much of the processor's behaviour and programming model to be defined via microprogram routines rather than by dedicated circuitry. Even late in the design process, microcode could easily be changed, whereas hard wired CPU designs were very cumbersome to change, so this greatly facilitated CPU design.

From the 1940s to the late 1970s, much programming was done in assembly language; higher level instructions meant greater programmer productivity, so an important advantage of microcode was the relative ease by which powerful machine instructions

could be defined. During the 1970s, CPU speeds grew more quickly than memory speeds and numerous techniques such as memory block transfer, memory pre-fetch and multi-level caches were used to alleviate this. High level machine instructions, made possible by microcode, helped further, as fewer more complex machine instructions require less memory bandwidth. For example, an operation on a character string could be done as a single machine instruction, thus avoiding multiple instruction fetches.

Architectures with instruction sets implemented by complex microprograms included the IBM System/360 and Digital Equipment Corporation VAX. The approach of increasingly complex microcode-implemented instruction sets was later called CISC. A middle way, used in many microprocessors, is to use PLAs and/or ROMs (instead of combinatorial logic) mainly for instruction decoding, and let a simple state machine (without much, or any, microcode) do most of the sequencing. The various practical uses of microcode and related techniques (such as PLAs) have been numerous over the years, as well as approaches to where, and to which extent, it should be used. It is still used in modern CPU designs.

Other benefits

A processor's microprograms operate on a more primitive, totally different and much more hardware-oriented architecture than the assembly instructions visible to normal programmers. In coordination with the hardware, the microcode implements the programmer-visible architecture. The underlying hardware need not have a fixed relationship to the visible architecture. This makes it possible to implement a given instruction set architecture on a wide variety of underlying hardware micro-architectures.

Doing so is important if binary program compatibility is a priority. That way previously existing programs can run on totally new hardware without requiring revision and recompilation. However there may be a performance penalty for this approach. The tradeoffs between application backward compatibility vs CPU performance are hotly debated by CPU design engineers.

The IBM System/360 has a 32-bit architecture with 16 general-purpose registers, but most of the System/360 implementations actually use hardware that implemented a much simpler underlying microarchitecture; for example, the System/360 Model 30 had 8-bit data paths to the arithmetic logic unit (ALU) and main memory and implemented the general-purpose registers in a special unit of higher-speed core memory, and the System/360 Model 40 had 8-bit data paths to the ALU and 16-bit data paths to main memory and also implemented the general-purpose registers in a special unit of higher-speed core memory. The Model 50 and Model 65 had full 32-bit data paths and implemented the general-purpose registers in faster transistor circuits. In this way, microprogramming enabled IBM to design many System/360 models with substantially different hardware and spanning a wide range of cost and performance, while making them all architecturally compatible. This dramatically reduced the amount of unique system software that had to be written for each model.

A similar approach was used by Digital Equipment Corporation in their VAX family of computers. Initially a 32-bit TTL processor in conjunction with supporting microcode implemented the programmer-visible architecture. Later VAX versions used different microarchitectures, yet the programmer-visible architecture did not change.

Microprogramming also reduced the cost of field changes to correct defects (bugs) in the processor; a bug could often be fixed by replacing a portion of the microprogram rather than by changes being made to hardware logic and wiring.

History

In 1947, the design of the MIT Whirlwind introduced the concept of a control store as a way to simplify computer design and move beyond *ad hoc* methods. The control store was a two-dimensional lattice: one dimension accepted "control time pulses" from the CPU's internal clock, and the other connected to control signals on gates and other circuits. A "pulse distributor" would take the pulses generated by the CPU clock and break them up into eight separate time pulses, each of which would activate a different row of the lattice. When the row was activated, it would activate the control signals connected to it.

Described another way, the signals transmitted by the control store are being played much like a player piano roll. That is, they are controlled by a sequence of very wide words constructed of bits, and they are "played" sequentially. In a control store, however, the "song" is short and repeated continuously.

In 1951 Maurice Wilkes enhanced this concept by adding *conditional execution*, a concept akin to a conditional in computer software. His initial implementation consisted of a pair of matrices, the first one generated signals in the manner of the Whirlwind control store, while the second matrix selected which row of signals (the microprogram instruction word, as it were) to invoke on the next cycle. Conditionals were implemented by providing a way that a single line in the control store could choose from alternatives in the second matrix. This made the control signals conditional on the detected internal signal. Wilkes coined the term **microprogramming** to describe this feature and distinguish it from a simple control store.

Examples of microprogrammed systems

- In common with many other complex mechanical devices, Charles Babbage's analytical engine used banks of cams to control each operation, i.e. it had a read-only control store. As such it deserves to be recognised as the first microprogrammed computer to be designed, even if it has not yet been realised in hardware.
- The EMIDEC 1100 reputedly used a hard-wired control store consisting of wires threaded through ferrite cores, known as 'the laces'.
- Most models of the IBM System/360 series were microprogrammed:

    - The Model 25 was unique among System/360 models in using the top 16k bytes of core storage to hold the control storage for the microprogram. The 2025 used a 16-bit microarchitecture with seven control words (or microinstructions). At power up, or full system reset, the microcode was loaded from the card reader. The IBM 1410 emulation for this model was loaded this way.
    - The Model 30, the slowest model in the line, used an 8-bit microarchitecture with only a few hardware registers; everything that the programmer saw was emulated by the microprogram. The microcode for this model was also held on special punched cards, which were stored inside the machine in a dedicated reader per card, called "CROS" units (Capacitor Read-Only Storage). A second CROS reader was installed for machines ordered with 1620 emulation.
    - The Model 40 used 56-bit control words. The 2040 box implements both the System/360 main processor and the multiplex channel (the I/O processor). This model used "TROS" dedicated readers similar to "CROS" units, but with an inductive pickup (Transformer Read-only Store).
    - The Model 50 had two internal datapaths which operated in parallel: a 32-bit datapath used for arithmetic operations, and an 8-bit data path used in some logical operations. The control store used 90-bit microinstructions.
    - The Model 85 had separate instruction fetch (I-unit) and execution (E-unit) to provide high performance. The I-unit is hardware controlled. The E-unit is microprogrammed; the control words are 108 bits wide on a basic 360/85 and wider if an emulator feature is installed.

- The NCR 315 was microprogrammed with hand wired ferrite cores (a ROM) pulsed by a sequencer with conditional execution. Wires routed through the cores were enabled for various data and logic elements in the processor.
- The Digital Equipment Corporation PDP-11 processors, with the exception of the PDP-11/20, were microprogrammed.
- Many systems from the Burroughs were microprogrammed:

    - The B700 "microprocessor" executed application-level opcodes using sequences of 16-bit microinstructions stored in main memory, each of these was either a register-load operation or mapped to a single 56-bit "nanocode" instruction stored in read-only memory. This allowed comparatively simple hardware to act either as a mainframe peripheral controller or to be packaged as a standalone computer.
    - The B1700 was implemented with radically different hardware including bit-addressable main memory but had a similar multi-layer organisation. The operating system would preload the interpreter for whatever language was required. These interpreters presented different virtual machines for COBOL, Fortran, etc.

- Microdata produced computers in which the microcode was accessible to the user; this allowed the creation of custom assembler level instructions. Microdata's Reality operating system design made extensive use of this capability.
- The Nintendo 64's Reality Co-Processor, which serves as the console's graphics processing unit and audio processor, utilized microcode; it is possible to implement new effects or tweak the processor to achieve the desired output. Some well-known examples of custom microcode include Factor 5's N64 port of the *Indiana Jones and the Infernal Machine*, *Star Wars: Rogue Squadron* and *Star Wars: Battle for Naboo*.
- The VU0 and VU1 vector units in the Sony PlayStation 2 are microprogrammable; in fact, VU1 was *only* accessible via microcode for the first several generations of the SDK.

Implementation

Each microinstruction in a microprogram provides the bits which control the functional elements that internally compose a CPU. The advantage over a hard-wired CPU is that internal CPU control becomes a specialized form of a computer program. Microcode thus transforms a complex electronic design challenge (the control of a CPU) into a less-complex programming challenge.

To take advantage of this, computers were divided into several parts:

A microsequencer picked the next word of the control store. A sequencer is mostly a counter, but usually also has some way to jump to a different part of the control store depending on some data, usually data from the instruction register and always some part of the control store. The simplest sequencer is just a register loaded from a few bits of the control store.

A register set is a fast memory containing the data of the central processing unit. It may include the program counter, stack pointer, and other numbers that are not easily accessible to the application programmer. Often the register set is a triple-ported register file, that is, two registers can be read, and a third written at the same time.

An arithmetic and logic unit performs calculations, usually addition, logical negation, a right shift, and logical AND. It often performs other functions, as well.

There may also be a memory address register and a memory data register, used to access the main computer storage.

Together, these elements form an "execution unit". Most modern CPUs have several execution units. Even simple computers usually have one unit to read and write memory, and another to execute user code.

These elements could often be bought together as a single chip. This chip came in a fixed width which would form a 'slice' through the execution unit. These were known as 'bit

slice' chips. The AMD Am2900 family is one of the best known examples of bit slice elements.

The parts of the execution units, and the execution units themselves are interconnected by a bundle of wires called a bus.

Programmers develop microprograms. The basic tools are software: A microassembler allows a programmer to define the table of bits symbolically. A simulator program executes the bits in the same way as the electronics (hopefully), and allows much more freedom to debug the microprogram.

After the microprogram is finalized, and extensively tested, it is sometimes used as the input to a computer program that constructs logic to produce the same data. This program is similar to those used to optimize a programmable logic array. No known computer program can produce optimal logic, but even pretty good logic can vastly reduce the number of transistors from the number required for a ROM control store. This reduces the cost and power used by a CPU.

Microcode can be characterized as **horizontal** or **vertical**. This refers primarily to whether each microinstruction directly controls CPU elements (horizontal microcode), or requires subsequent decoding by combinatorial logic before doing so (vertical microcode). Consequently each horizontal microinstruction is wider (contains more bits) and occupies more storage space than a vertical microinstruction.

Horizontal microcode

Horizontal microcode is typically contained in a fairly wide control store; it is not uncommon for each word to be 56 bits or more. On each tick of a sequencer clock a microcode word is read, decoded, and used to control the functional elements which make up the CPU.

In a typical implementation a horizontal microprogram word comprises fairly tightly defined groups of bits. For example, one simple arrangement might be:

register source A	register source B	destination register	arithmetic and logic unit operation	type of jump	jump address

For this type of micromachine to implement a JUMP instruction with the address following the opcode, the microcode might require two clock ticks; the engineer designing it would write microassembler source code looking something like this:

```
   # Any line starting with a number-sign is a comment
   # This is just a label, the ordinary way assemblers symbolically
represent a
   # memory address.
InstructionJUMP:
```

```
      # To prepare for the next instruction, the instruction-decode
microcode has already
      # moved the program counter to the memory address register.  This
instruction fetches
      # the target address of the jump instruction from the memory word
following the
      # jump opcode, by copying from the memory data register to the
memory address register.
      # This gives the memory system two clock ticks to fetch the next
      # instruction to the memory data register for use by the
instruction decode.
      # The sequencer instruction "next" means just add 1 to the
control word address.
   MDR, NONE, MAR,  COPY, NEXT, NONE
      # This places the address of the next instruction into the PC.
      # This gives the memory system a clock tick to finish the fetch
started on the
      # previous microinstruction.
      # The sequencer instruction is to jump to the start of the
instruction decode.
   MAR, 1, PC, ADD,  JMP,  InstructionDecode
      # The instruction decode is not shown, because it is usually a
mess, very particular
      # to the exact processor being emulated.  Even this example is
simplified.
      # Many CPUs have several ways to calculate the address, rather
than just fetching
      # it from the word following the op-code. Therefore, rather than
just one
      # jump instruction, those CPUs have a family of related jump
instructions.
```

For each tick it is common to find that only some portions of the CPU are used, with the remaining groups of bits in the microinstruction being no-ops. With careful design of hardware and microcode this property can be exploited to parallelise operations which use different areas of the CPU, for example in the case above the ALU is not required during the first tick so it could potentially be used to complete an earlier arithmetic instruction.

Vertical microcode

In vertical microcode, each microinstruction is encoded—that is, the bit fields may pass through intermediate combinatory logic which in turn generates the actual control signals for internal CPU elements (ALU, registers, etc.). In contrast, with horizontal microcode the bit fields themselves directly produce the control signals. Consequently vertical microcode requires smaller instruction lengths and less storage, but requires more time to decode, resulting in a slower CPU clock.

Some vertical microcodes are just the assembly language of a simple conventional computer that is emulating a more complex computer. This technique was popular in the time of the PDP-8. Another form of vertical microcode has two fields:

field select field value

The "field select" selects which part of the CPU will be controlled by this word of the control store. The "field value" actually controls that part of the CPU. With this type of microcode, a designer explicitly chooses to make a slower CPU to save money by reducing the unused bits in the control store; however, the reduced complexity may increase the CPU's clock frequency, which lessens the effect of an increased number of cycles per instruction.

As transistors became cheaper, horizontal microcode came to dominate the design of CPUs using microcode, with vertical microcode no longer being used.

Writable control stores

A few computers were built using "writable microcode" -- rather than storing the microcode in ROM or hard-wired logic, the microcode was stored in a RAM called a *Writable Control Store* or *WCS*. Such a computer is sometimes called a *Writable Instruction Set Computer* or *WISC*. Many of these machines were experimental laboratory prototypes, such as the WISC CPU/16 and the RTX 32P.

There were also commercial machines that used writable microcode, such as early Xerox workstations, the DEC VAX 8800 ("Nautilus") family, the Symbolics L- and G-machines, and a number of IBM System/370 implementations. Some DEC PDP-10 machines stored their microcode in SRAM chips (about 80 bits wide x 2 Kwords), which was typically loaded on power-on through some other front-end CPU. Many more machines offered user-programmable writable control stores as an option (including the HP 2100,DEC PDP-11/60 and Varian Data Machines V-70 series minicomputers). WCS offered several advantages including the ease of patching the microprogram and, for certain hardware generations, faster access than ROMs could provide. User-programmable WCS allowed the user to optimize the machine for specific purposes.

Some CPU designs compile the instruction set to a writable RAM or FLASH inside the CPU (such as the Rekursiv processor and the Imsys Cjip), or an FPGA (reconfigurable computing). The Western Digital MCP-1600 is an older example, using a dedicated, separate ROM for microcode.

A CPU that uses microcode generally takes several clock cycles to execute a single instruction, one clock cycle for each step in the microprogram for that instruction. Some CISC processors include instructions that can take a very long time to execute. Such variations interfere with both interrupt latency and, what is far more important in modern systems, pipelining.

Several Intel CPUs in the IA32 architecture family have writable microcode. This has allowed bugs in the Intel Core 2 microcode and Intel Xeon microcode to be fixed in software, rather than requiring the entire chip to be replaced. Such fixes can be installed by Linux, FreeBSD Microsoft Windows, or the motherboard BIOS.

Microcode versus VLIW and RISC

The design trend toward heavily microcoded processors with complex instructions began in the early 1960s and continued until roughly the mid-1980s. At that point the RISC design philosophy started becoming more prominent. This included the points:

- Analysis shows complex instructions are rarely used, hence the machine resources devoted to them are largely wasted.
- Programming has largely moved away from assembly level, so it's no longer worthwhile to provide complex instructions for productivity reasons.
- The machine resources devoted to rarely-used complex instructions are better used for expediting performance of simpler, commonly-used instructions.
- Complex microcoded instructions requiring many, varying clock cycles are difficult to pipeline for increased performance.
- Simpler instruction sets allow direct execution by hardware, avoiding the performance penalty of microcoded execution.

It should be mentioned that there are counter-points as well:

- The complex instructions in heavily microcoded implementations may not take much extra machine resources (except microcode space); for instance, the same ALU is often used to calculate an effective address as well as computing the result from the actual operands, e.g. the original Z80, 8086, and others.
- The simpler non-RISC instructions, i.e. involving direct memory operands are frequently used by modern compilers, even immediate to stack (i.e. memory result) arithmetic operations are commonly employed. Although such memory operations, often with varying length encodings are more difficult to pipeline, it is still fully feasible, clearly exemplified by the i486, AMD K5, Cyrix 6x86, etc.
- Non-RISC instructions inherently perform more work per instruction (on average), and are also normally highly encoded, so they enable smaller overall size of the same program, and thus better use of limited cache memories.
- Modern CISC/RISC implementations, e.g. x86 designs, decode instructions into dynamically buffered micro-operations with instruction encodings similar to traditional fixed microcode. Ordinary static microcode is used as hardware assistance for complex multistep operations such as auto-repeating instructions and for transcendental functions in the floating point unit; it is also used for special purpose instructions (such as CPUID) and internal control and configuration purposes.
- The simpler instructions in CISC architectures are also directly executed in hardware in modern implementations.

Many RISC and VLIW processors are designed to execute every instruction (as long as it is in the cache) in a single cycle. This is very similar to the way CPUs with microcode execute one microinstruction per cycle. VLIW processors have instructions that behave similarly to very wide horizontal microcode, although typically without such fine-grained

control over the hardware as provided by microcode. RISC instructions are sometimes similar to the narrow vertical microcode.

Chapter-6

Branch Predictor

In computer architecture, a **branch predictor** is a digital circuit that tries to guess which way a branch (e.g. an if-then-else structure) will go before this is known for sure. The purpose of the branch predictor is to improve the flow in the instruction pipeline. Branch predictors are crucial in today's pipelined microprocessors for achieving high performance.

A two-way branching is usually implemented with a conditional jump instruction. A conditional jump can either be "not taken" and continue execution with the first branch of code which follows immediately after the conditional jump - or it can be "taken" and jump to a different place in program memory where the second branch of code is stored.

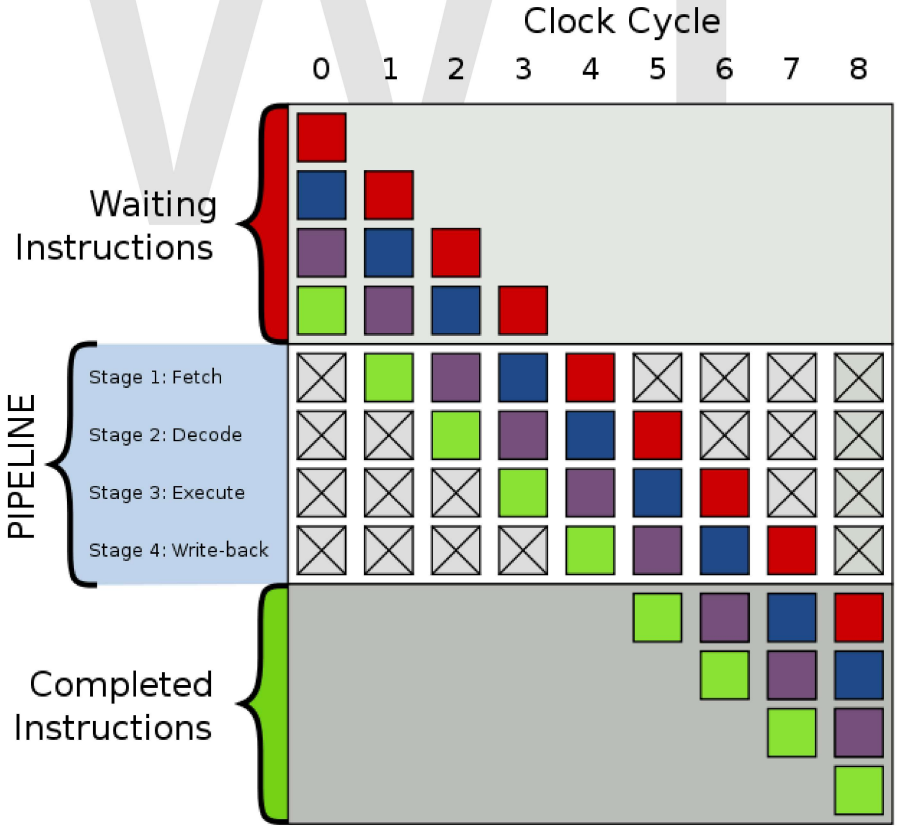


Figure 1: Example of 4-stage pipeline. The colored boxes represent instructions independent of each other

It is not known for certain whether a conditional jump will be taken or not taken until the condition has been calculated and the conditional jump has passed the execution stage in the instruction pipeline.

Without branch prediction, the processor would have to wait until the conditional jump instruction has passed the execute stage before the next instruction can enter the fetch stage in the pipeline. The branch predictor attempts to avoid this waste of time by trying to guess whether the conditional jump is most likely to be taken or not taken. The branch that is guessed to be the most likely is then fetched and speculatively executed. If it is later detected that the guess was wrong then the speculatively executed or partially executed instructions are discarded and the pipeline starts over with the correct branch.

The time that is wasted in case of a branch misprediction is equal to the number of stages in the pipeline from the fetch stage to the execute stage. Modern microprocessors tend to have quite long pipelines so that the misprediction delay is between 10 and 20 clock cycles. The longer the pipeline the higher the need for a good branch predictor.

The first time a conditional jump instruction is encountered, there is not much information to base a prediction on. But the branch predictor keeps records of whether branches are taken or not taken. When it encounters a conditional jump that has been seen several times before then it can base the prediction on the past history. The branch predictor may, for example, recognize that the conditional jump is taken more often than not, or that it is taken every second time.

Branch prediction is not the same as branch target prediction. Branch prediction attempts to guess whether a conditional jump will be taken or not. Branch target prediction attempts to guess the target of a taken conditional or unconditional jump before it is computed by decoding and executing the instruction itself. Branch prediction and branch target prediction are often combined into the same circuitry.

Implementation

Static prediction

Static prediction is the simplest branch prediction technique because it does not rely on information about the dynamic history of code executing. Instead it predicts the outcome of a branch based solely on the branch instruction.

The early implementations of SPARC and MIPS (two of the first commercial RISC architectures) used single direction static branch prediction: they always predicted that a conditional jump would not be taken, so they always fetched the next sequential instruction. Only when the branch or jump was evaluated and found to be taken did the instruction pointer get set to a non-sequential address.

Both CPUs evaluated branches in the decode stage and had a single cycle instruction fetch. As a result, the branch target recurrence was two cycles long, and the machine

would always fetch the instruction immediately after any taken branch. Both architectures defined branch delay slots in order to utilize these fetched instructions.

A more complex form of static prediction assumes that backwards branches will be taken, and forward-pointing branches will not be taken. A backwards branch is one that has a target address that is lower than its own address. This technique can help with prediction accuracy of loops, which are usually backward-pointing branches, and are taken more often than not taken.

Some processors allow branch prediction hints to be inserted into the code to tell whether the static prediction should be taken or not taken. The Intel Pentium 4 accepts branch prediction hints while this feature is abandoned in later processors.

Static prediction is used as a fall-back technique in some processors with dynamic branch prediction when there isn't any information for dynamic predictors to use. Both the Motorola MPC7450 (G4e) and the Intel Pentium 4 use this technique as a fall-back.

Next line prediction

Some superscalar processors (MIPS R8000, Alpha 21264 and Alpha 21464 (EV8)) fetch each line of instructions with a pointer to the next line. This next line predictor handles branch target prediction as well as branch direction prediction.

When a next line predictor points to aligned groups of 2, 4 or 8 instructions, the branch target will usually not be the first instruction fetched, and so the initial instructions fetched are wasted. Assuming for simplicity a uniform distribution of branch targets, 0.5, 1.5, and 3.5 instructions fetched are discarded, respectively.

Since the branch itself will generally not be the last instruction in an aligned group, instructions after the taken branch (or its delay slot) will be discarded. Once again assuming a uniform distribution of branch instruction placements, 0.5, 1.5, and 3.5 instructions fetched are discarded.

The discarded instructions at the branch and destination lines add up to nearly a complete fetch cycle, even for a single-cycle next-line predictor.

Saturating counter

A saturating counter or bimodal predictor is a state machine with four states:

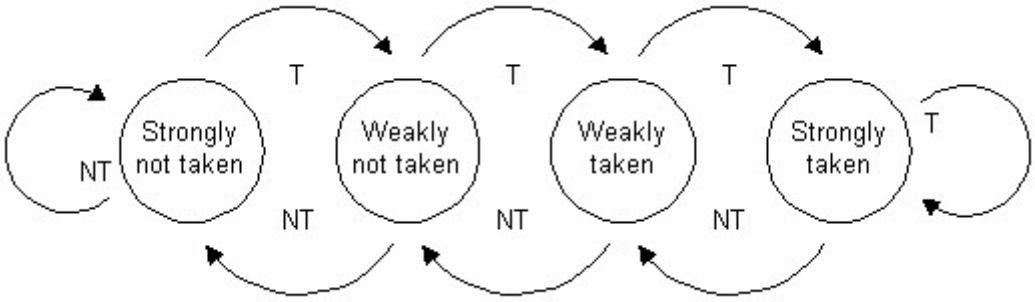


Figure 2: State diagram of 2-bit saturating counter

- Strongly not taken
- Weakly not taken
- Weakly taken
- Strongly taken

When a branch is evaluated, the corresponding state machine is updated. Branches evaluated as not taken decrement the state towards strongly not taken, and branches evaluated as taken increment the state towards strongly taken. The advantage of the two-bit counter over a one-bit scheme is that a conditional jump has to deviate twice from what it has done most in the past before the prediction changes. For example, a loop-closing conditional jump is mispredicted once rather than twice.

The original, non-MMX Intel Pentium processor uses a saturating counter, though with an imperfect implementation.

On the SPEC'89 benchmarks, very large bimodal predictors saturate at 93.5% correct, once every branch maps to a unique counter.

The predictor table is indexed with the instruction address bits, so that the processor can fetch a prediction for every instruction before the instruction is decoded.

Two-level adaptive predictor

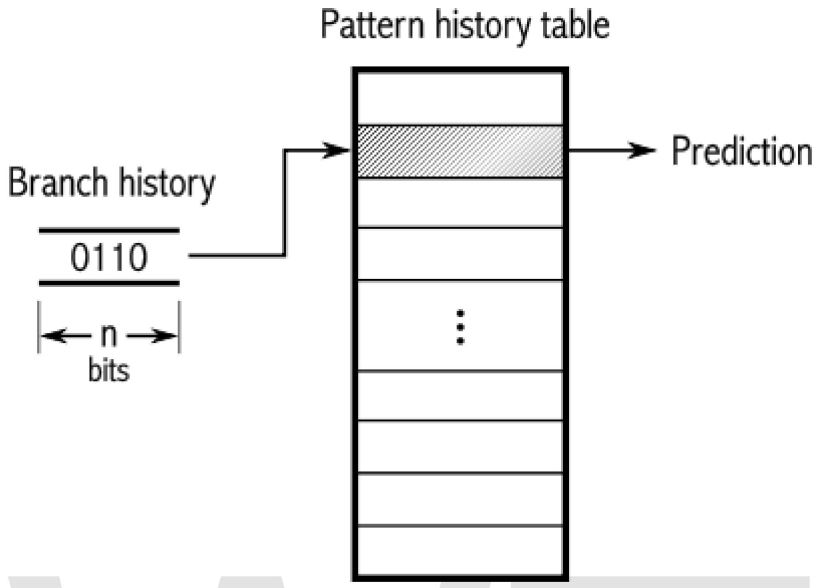


Figure 3: Two-level adaptive branch predictor. Every entry in the pattern history table represents a 2-bit saturating counter of the type shown in figure 2.

Conditional jumps that are taken every second time or have some other regularly recurring pattern are not predicted well by the saturating counter. A two-level adaptive predictor remembers the history of the last n occurrences of the branch and use one saturating counter for each of the possible 2^n history patterns. This method is illustrated in figure 3.

Consider the example of n = 2. This means that the last two occurrences of the branch are stored in a 2-bit shift register. This branch history register can have 4 different binary values: 00, 01, 10, and 11; where 0 means "not taken" and 1 means "taken". Now, we make a pattern history table with four entries, one for each of the $2^n = 4$ possible branch histories. Each entry in the pattern history table contains a 2-bit saturating counter of the same type as in figure 2. The branch history register is used for choosing which of the four saturating counters to use. If the history is 00 then the first counter is used. If the history is 11 then the last of the four counters is used.

Assume, for example, that a conditional jump is taken every third time. The branch sequence is 001001001... In this case, entry number 00 in the pattern history table will go to state "strongly taken", indicating that after two zeroes comes a one. Entry number 01 will go to state "strongly not taken", indicating that after 01 comes a 0. The same is the case with entry number 10, while entry number 11 is never used because there are never two consecutive ones.

The general rule for a two-level adaptive predictor with an n-bit history is that it can predict any repetitive sequence with any period if all n-bit sub-sequences are different.

The advantage of the two-level adaptive predictor is that it can quickly learn to predict an arbitrary repetitive pattern. This method was invented by T.-Y. Yeh and Y. N. Patt of the University of Michigan. Since the initial publication in 1991, this method has become very popular. Variants of this prediction method are used in most modern microprocessors today.

Local branch prediction

A local branch predictor has a separate history buffer for each conditional jump instruction. It may use a two-level adaptive predictor. The history buffer is separate for each conditional jump instruction, while the pattern history table may be separate as well or it may be shared between all conditional jumps.

The Intel Pentium MMX, Pentium II and Pentium III have local branch predictors with a local 4-bit history and a local pattern history table with 16 entries for each conditional jump.

On the SPEC'89 benchmarks, very large local predictors saturate at 97.1% correct.

Global branch prediction

A global branch predictor does not keep a separate history record for each conditional jump. Instead it keeps a shared history of all conditional jumps. The advantage of a shared history is that any correlation between different conditional jumps is utilized in the prediction making. The disadvantage is that the history is diluted by irrelevant information in case the different conditional jumps are uncorrelated, and that the history buffer may not include any bits from the same branch in case there are many other branches in between. It may use a two-level adaptive predictor.

This scheme is only better than the saturating counter scheme for large table sizes, and it is rarely as good as local prediction. The history buffer must be longer in order to make a good prediction. The size of the pattern history table grows exponentially with the size of the history buffer. Hence, the big pattern history table must be shared between all conditional jumps.

A two-level adaptive predictor with globally shared history buffer and pattern history table is called a gshare predictor. This mechanism is used in AMD microprocessors and in Intel Pentium M, Core and Core 2.

On the SPEC'89 benchmarks, very large gshare predictors saturate at 96.6% correct, which is just a little worse than large local predictors.

Alloyed branch prediction

An alloyed branch predictor, also called gselect, combines the local and global prediction principles by concatenating local and global branch histories. Tests indicate that the VIA Nano processor may be using this technique.

Agree predictor

An agree predictor is a two-level adaptive predictor with globally shared history buffer and pattern history table, and an additional local saturating counter. The outputs of the local and the global predictors are XORed with each other to give the final prediction. The purpose is to reduce contentions in the pattern history table where two branches with opposite prediction happen to share the same entry in the pattern history table.

The agree predictor was used in the first version of the Intel Pentium 4, but was later abandoned.

Hybrid predictor

A hybrid predictor, also called combined predictor, implements more than one prediction mechanism. The final prediction is based either on a meta-predictor that remembers which of the predictors has made the best predictions in the past, or a majority vote function based on an odd number of different predictors.

Scott McFarling proposed combined branch prediction in his 1993 paper.

On the SPEC'89 benchmarks, such a predictor is about as good as the local predictor.

Predictors like gshare use multiple table entries to track the behavior of any particular branch. This multiplication of entries makes it much more likely that two branches will map to the same table entry (a situation called aliasing), which in turn makes it much more likely that prediction accuracy will suffer for those branches. Once you have multiple predictors, it is beneficial to arrange that each predictor will have different aliasing patterns, so that it is more likely that at least one predictor will have no aliasing. Combined predictors with different indexing functions for the different predictors are called *gskew* predictors, and are analogous to skewed associative caches used for data and instruction caching.

Loop predictor

A conditional jump that controls a loop is best predicted with a special loop predictor. A conditional jump in the bottom of a loop that repeats N times will be taken N-1 times and then not taken once. If the conditional jump is placed at the top of the loop, it will be not taken N-1 times and then taken once. A conditional jump that goes many times one way and then the other way once is detected as having loop behavior. Such a conditional jump

can be predicted easily with a simple counter. A loop predictor is part of a hybrid predictor where a meta-predictor detects whether the conditional jump has loop behavior.

Many microprocessors today have loop predictors.

Prediction of indirect jumps

An indirect jump instruction can choose between more than two branches. Newer processors from Intel and AMD can predict indirect branches by using a two-level adaptive predictor. An indirect jump that chooses between more than two branches contributes more than one bit to the history buffer.

Processors without this mechanism will simply predict an indirect jump to go to the same target as it did last time.

Prediction of function returns

A function will normally return to where it is called from. The return instruction is an indirect jump that reads its target address from the call stack. Many microprocessors have a separate prediction mechanism for return instructions. This mechanism is based on a so-called *return stack buffer*, which is a local mirror of the call stack. The size of the return stack buffer is typically 4 - 16 entries.

Overriding branch prediction

The trade-off between fast branch prediction and good branch prediction is sometimes dealt with by having two branch predictors. The first branch predictor is fast and simple. The second branch predictor, which is slower, more complicated, and with bigger tables, will override a possibly wrong prediction made by the first predictor.

The Alpha 21264 and Alpha EV8 microprocessors used a fast single-cycle next line predictor to handle the branch target recurrence and provide a simple and fast branch prediction. Because the next line predictor is so inaccurate, and the branch resolution recurrence takes so long, both cores have two-cycle secondary branch predictors which can override the prediction of the next line predictor at the cost of a single lost fetch cycle.

The Intel Core i7 has two branch target buffers and possibly two or more branch predictors.

Neural branch predictors

The first dynamic neural branch predictors (LVQ-predictors and perceptrons) were proposed by Prof. Lucian Vintan (Lucian Blaga University of Sibiu, Romania), in his paper entitled "Towards a High Performance Neural Branch Predictor", Proceedings of The International Joint Conference on Neural Networks - IJCNN '99, Washington DC,

USA, 1999. The neural branch predictor research was strongly developed further by Dr. Daniel Jimenez (Rutgers University, USA). In 2001, (HPCA Conference) it was the first presented perceptron predictor that was feasible to implement in hardware.

The main advantage of the neural predictor is its ability to exploit long histories while requiring only linear resource growth. Classical predictors require exponential resource growth. Jimenez reports a global improvement of 5.7% over a McFarling-style hybrid predictor,

The main disadvantage of the perceptron predictor is its high latency. Even after taking advantage of high-speed arithmetic tricks, the computation latency is relatively high compared to the clock period of many modern microarchitectures. In order to reduce the prediction latency, Jimenez proposed in 2003 the *fast-path neural predictor*, where the perceptron predictor chooses its weights according to the current branch's path, rather than according to the branch's PC. Many other researchers developed this concept (A. Seznec, M. Monchiero, D. Tarjan & K. Skadron, V. Desmet, Akkary et al., K. Aasaraai, Michael Black, etc.)

The neural branch predictor concept is very promising. Most of the state of the art branch predictors are using a perceptron predictor. Intel already implements this idea in one of the IA-64's simulators (2003).

History

The IBM Stretch, designed in the late 1950s, pre-executed all unconditional branches and any conditional branches that depended on the index registers. For other conditional branches, the first two production models implemented predict untaken; subsequent models were changed to implement predictions based on the current values of the indicator bits (corresponding to today's condition codes). The Stretch designers had considered static hint bits in the branch instructions early in the project but decided against them. Misprediction recovery was provided by the lookahead unit on Stretch, and part of Stretch's reputation for less-than-stellar performance was blamed on the time required for misprediction recovery. Subsequent IBM large computer designs did not use branch prediction with speculative execution until the IBM 3090 in 1985.

Two-bit predictors were introduced by Tom McWilliams and Curt Widdoes in 1977 for the Lawrence Livermore National Lab S-1 supercomputer and independently by Jim Smith in 1979 at CDC.

Microprogrammed processors, popular from the 1960s to the 1980s and beyond, took multiple cycles per instruction, and generally did not require branch prediction. However, along with the IBM 3090, there are several examples of microprogrammed designs that incorporated branch prediction.

The Burroughs B4900, a microprogrammed COBOL machine released in ~1982 was pipelined and used branch prediction. The B4900 branch prediction history state was

stored back into the in-memory instructions during program execution. The B4900 implemented 4-state branch prediction by using 4 semantically equivalent branch opcodes to represent each branch operator type. The opcode used indicated the history of that particular branch instruction. If the hardware determined that the branch prediction state of a particular branch needed to be updated, it would rewrite the opcode with the semantically equivalent opcode that hinted the proper history. This scheme obtained a 93% hit rate. US patent 4,435,756 and others were granted on this scheme.

The VAX 9000, announced in 1989, was both microprogrammed and pipelined, and performed branch prediction.

The first commercial RISC processors, the MIPS R2000 and R3000 and the earlier SPARC processors, did only trivial "not-taken" branch prediction. Because they used branch delay slots, fetched just one instruction per cycle, and executed in-order, there was no performance loss. Later, the R4000 used the same trivial "not-taken" branch prediction, and lost two cycles to each taken branch because the branch resolution recurrence was four cycles long.

Branch prediction became more important with the introduction of pipelined superscalar processors like the Intel Pentium, DEC Alpha 21064, the MIPS R8000, and the IBM POWER series. These processors all relied on one-bit or simple bimodal predictors.

The DEC Alpha 21264 (EV6) uses a next-line predictor overridden by a combined local predictor and global predictor, where the combining choice is made by a bimodal predictor.

The AMD K8 has a combined bimodal and global predictor, where the combining choice is another bimodal predictor. This processor caches the base and choice bimodal predictor counters in bits of the L2 cache otherwise used for ECC. As a result, it has effectively very large base and choice predictor tables, and parity rather than ECC on instructions in the L2 cache. Parity is just fine, since any instruction suffering a parity error can be invalidated and refetched from memory.

The Alpha 21464 (EV8, cancelled late in design) had a minimum branch misprediction penalty of 14 cycles. It was to use a complex but fast next line predictor overridden by a combined bimodal and majority-voting predictor. The majority vote was between the bimodal and two gskew predictors.

Chapter-7

Branch Predication and Cycles Per Instruction

Branch predication

Branch predication is a strategy in computer architecture design for mitigating the costs usually associated with conditional branches, particularly branches to short sections of code. It does this by allowing each instruction to conditionally either perform an operation or do nothing.

Overview

Most computer programs contain code which will be executed only under specific conditions depending on factors which can not be determined before-hand, for example user input. As the majority of processors simply execute the next instruction in a sequence, the traditional solution is to insert *branch* instructions that allow a program to conditionally branch to a different section of code, thus changing the next step in the sequence. This was sufficient until designers began improving performance by implementing instruction pipelining, a method which is slowed down by branches.

Luckily, one of the more common patterns of code that normally relies on branching has a more elegant solution. Consider the following pseudocode:

```
if condition
    do this
else
    do that
```

On a system that uses conditional branching, this might translate to machine instructions looking similar to:

```
  branch if condition to label 1
```

```
  do that
  branch to label 2
label 1:
  do this
label 2:
  ...
```

With branch predication, all possible branch paths are executed, the correct path is kept and all others are thrown away. The basic idea is that each instruction is associated with a predicate (the word here used similarly to its usage in predicate logic) and that the instruction will only be executed if the predicate is true. The machine code for the above example using branch predication might look something like this:

```
(condition) do this
(not condition) do that
```

Note that besides eliminating branches, less code is needed in total, provided the architecture provides predicated instructions. While this does not guarantee faster execution in general, it will if the `do this` and `do that` blocks of code are short enough.

Typically, in order to claim a system has branch predication, most or all of the instructions must have this ability to execute conditionally based on a predicate.

Advantages

The main purpose of predication is to avoid jumps over very small sections of program code, increasing the effectiveness of pipelined execution and avoiding problems with the cache. It also has a number of more subtle benefits:

- Functions that are traditionally computed using simple arithmetic and bitwise operations may be quicker to compute using predicated instructions.
- Predicated instructions with different predicates can be mixed with each other and with unconditional code, allowing better instruction scheduling and so even better performance.
- Elimination of unnecessary branch instructions can make the execution of necessary branches, such as those that make up loops, faster by lessening the load on branch prediction mechanisms.

Disadvantages

Unfortunately, predication has one strong drawback: encoding space. In typical implementations, every instruction reserves a bitfield for the predicate specifying whether that instruction should have an effect. When available memory is limited, as on embedded devices, this space cost can be prohibitive. However, some architectures such as Thumb-2 are able to avoid this issue. Other detriments are the following:

- Predication complicates the hardware by adding levels of logic to critical paths and potentially degrades clock speed.
- A predicated block includes cycles for all operations, so shorter paths may take longer and be penalized.

Predication is most effective when paths are balanced or when the longest path is the most frequently executed, but determining such a path is very difficult at compile time, even at the presence of profiling information.

Examples

Predicated instructions were popular in European computer designs of the 1950s, including the Mailüfterl (1955), the Zuse Z22 (1955), the ZEBRA (1958), and the Electrologica X1 (1958). The IBM ACS-1 design of 1967 allocated a "skip" bit in its instruction formats, and the CDC Flexible Processor in 1976 allocated three conditional execution bits in its microinstruction formats.

In Intel's IA-64 architecture, almost every instruction in the IA-64 instruction set is predicated. The predicates themselves are stored in special purpose registers; one of the predicate registers is always true so that *unpredicated* instructions are simply instructions predicated with the value true. The use of predication is essential in the IA-64 implementation of software pipelining because it avoids the need for writing separated code for prologs and epilogs.

On the ARM architecture, almost all instructions can be conditionally executed. Thirteen different predicates are available, each depending on the four flags Carry, Overflow, Zero, and Negative in some way. The ARM's 16-bit Thumb instruction set has no branch predication, in order to save encoding space, but its successor Thumb-2 overcomes this problem using a special instruction which has no effect other than to supply predicates for the next four instructions.

Cycles per instruction

In computer architecture, **cycles per instruction** (aka **clock cycles per instruction**, **clocks per instruction**, or **CPI**) is a term used to describe one aspect of a processor's performance: the number of clock cycles that happen when an instruction is being executed. It is the multiplicative inverse of instructions per cycle.

Explanation

Let us assume a classic RISC pipeline, with the following 5 stages:

1. Instruction fetch cycle (IF)
2. Instruction decode/Register fetch cycle (ID)
3. Execution/Effective address cycle (EX)

4. Memory access (MEM)
5. Write-back cycle (WB)

Each stage requires one clock cycle and an instruction passes through the stages sequentially. Without pipelining, a new instruction is fetched in stage 1 only after the previous instruction finishes at stage 5. Therefore without pipelining the number of cycles it takes to execute an instruction is 5. This is the definition of CPI.

With pipelining we can improve the CPI by exploiting instruction level parallelism. For example, what if an instruction is fetched every cycle? We could theoretically have 5 instructions in the 5 pipeline stages at once (one instruction per stage). In this case, a different instruction would complete stage 5 in every clock cycle, and therefore on average we have one clock cycle per instruction (CPI = 1).

With a single-issue processor, the best CPI attainable is 1. However with multiple-issue processors, we may achieve even better CPI values. For example a processor that issues two instructions per clock cycle can achieve a CPI of 0.5 when two instructions are completing every clock cycle.

Calculations

Multi-cycle example

For the multi-cycle MIPS, there are 5 types of instructions:

- Load (5 cycles)
- Store (4 cycles)
- R-type (4 cycles)
- Branch (3 cycles)
- Jump (3 cycles)

If a program has:

- 50% R-type instructions
- 15% load instructions
- 25% store instructions
- 8% branch instructions
- 2% jump instructions

then, the CPI is:

$$\text{CPI} = (4 \times 50 + 5 \times 15 + 4 \times 25 + 3 \times 8 + 3 \times 2) / 100 = 4.05.$$

Example

A 40-MHz processor was used to execute a benchmark program with the following instruction mix and clock cycle count:

Instruction type	Instruction count	Clock cycle count
Integer arithmetic	45000	1
Data transfer	32000	2
Floating point	15000	2
Control transfer	8000	2

Determine the effective CPI, MIPS rate, and execution time for this program.

Total instruction count = 100000.
CPI = (45000*1 + 32000*2 + 15000*2 + 8000*2)/100000 = 155000/100000 = 1.55.
MIPS = clock frequency/(CPI*1000000) = (40*1000000)/(1.55*1000000) = 25.8.

Therefore:

Execution time (T) = CPI*Instruction count*clock time = CPI*Instruction count/frequency = 1.55*100000/40000000 = 1.55/400 = 3.87 ms.

Chapter-8

Burroughs Large Systems Instruction Sets

The **Burroughs large systems instruction sets** include the set of valid operations for the Burroughs B5000, B5500, and B5700 computers and the set of valid operations for the Burroughs B6500 and later Burroughs large systems including the current (as of 2006) Unisys Clearpath/MCP systems. These unique machines have a distinctive design and instruction set. Each word of data is associated with a type, and the effect of an operation on that word can depend on the type. Further, the machines are stack based to the point that they had no user-addressable registers.

As you would expect from the description of the run-time data structures used in these systems, Burroughs large systems also have an interesting instruction set. There are less than 200 operators, all of which fit into 8-bit "syllables." Many of these operators are polymorphic depending on the kind of data being acted on as given by the tag. If we ignore the powerful string scanning, transfer, and edit operators, the basic set is only about 120 operators. If we remove the operators reserved for the operating system such as MVST and HALT, the set of operators commonly used by user-level programs is less than 100.

Since there are no programmer-addressable registers, most of the register manipulating operations required in other architectures are not needed, nor are variants for performing operations between pairs of registers, since all operations are applied to the top of the stack. This also makes code files very compact, since operators are zero-address and do not need to include the address of registers or memory locations in the code stream.

For example the B5000 has only one ADD operator. Typical architectures require multiple operators for each data type, for example add.i, add.f, add.d, add.l for integer, float, double, and long data types. The architecture only distinguishes single and double precision numbers – integers are just reals with a zero exponent. When one or both of the operands has a tag of 2, a double precision add is performed, otherwise tag 0 indicates single precision. Thus the tag itself is the equivalent of the operator .i, .f, .d, and .l extension. This also means that the code and data can never be mismatched.

Two operators are important in the handling of on-stack data – VALC and NAMC. These are two-bit operators, 00 being VALC, value call, and 01 being NAMC, name call. The following six bits of the syllable provide the address couple. Thus VALC covers syllable values 00 to 3F and NAMC 40 to 7F.

VALC is another polymorphic operator. If it hits a data word, that word is loaded to the top of stack. If it hits an IRW, that is followed, possibly in a chain of IRWs until a data word is found. If a PCW is found, then a function is entered to compute the value and the VALC does not complete until the function returns.

NAMC simply loads the address couple onto the top of the stack as an IRW (with the tag automatically set to 1).

In the following operator explanations remember that A and B are the top two stack registers. Double precision extensions are provided by the X and Y registers; thus the top two double precision operands are given by AX and BY. (Mostly AX and BY is implied by just A and B.)

Arithmetic operators

ADD — Add top two stack operands (B := B + A or BY := BY + AX if double precision)
SUBT — Subtract (B - A)
MULT — Multiply with single or double precision result
MULX — Extended multiply with forced double precision result
DIVD — Divide with real result
IDIV — Divide with integer result
RDIV — Return remainder after division
NTIA — Integerize truncated
NTGR — Integerize rounded
NTGD — Integerize rounded with double precision result
CHSN — Change sign
JOIN — Join two singles to form a double
SPLT — Split a double to form two singles
ICVD — Input convert destructive – convert BCD number to binary (for COBOL)
ICVU — Input convert update – convert BCD number to binary (for COBOL)
SNGL — Set to single precision rounded
SNGT — Set to single precision truncated
XTND — Set to double precision
PACD — Pack destructive
PACU — Pack update
USND — Unpack signed destructive
USNU — Unpack signed update
UABD — Unpack absolute destructive
UABU — Unpack, absolute update
SXSN — Set external sign

ROFF — Read and clear overflow flip flop
RTFF — Read true/false flip flop

Comparison operators

LESS — Is B < A?
GREQ — Is B >= A?
GRTR — Is B > A?
LSEQ — Is B <= A?
EQUL — Is B = A?
NEQL — Is B <> A?
SAME — Does B have the same bit pattern as A, including the tag

Logical operators

LAND — Logical bitwise and of all bits in operands
LOR — Logical bitwise or of all bits in operands
LNOT — Logical bitwise complement of all bits in operand
LEQV — Logical bitwise equivalence of all bits in operands

Branch and call operators

BRUN — Branch unconditional (offset given by following code syllables)
DBUN — Dynamic branch unconditional (offset given in top of stack)
BRFL — Branch if last result false (offset given by following code syllables)
DBFL — Dynamic branch if last result false (offset given in top of stack)
BRTR — Branch if last result true (offset given by following code syllables)
DBTR — Dynamic branch if last result true (offset given in top of stack)
EXIT — Exit current environment (terminate process)
STBR — Step and branch (used in loops; operand must be SIW)
ENTR — Execute a procedure call as given by a tag 7 PCW, resulting in an RCW at D[n] + 1
RETN — Return from current routine to place given by RCW at D[n] + 1 and remove the stack frame

Bit and field operators

BSET — Bit set (bit number given by syllable following instruction)
DBST — Dynamic bit set (bit number given by contents of B)
BRST — Bit reset (bit number given by syllable following instruction)
DBRS — Dynamic bit reset (bit number given by contents of B)
ISOL — Field isolate (field given in syllables following instruction)
DISO — Dynamic field isolate (field given in top of stack words)
FLTR — Field transfer (field given in syllables following instruction)
DFTR — Dynamic field transfer (field given in top of stack words)
INSR — Field insert (field given in syllables following instruction)

DINS — Dynamic field insert (field given in top of stack words)
CBON — Count binary ones in the top of stack word (A or AX)
SCLF — Scale left
DSLF — Dynamic scale left
SCRT — Scale right
DSRT — Dynamic scale right
SCRS — Dynamic scale right save
DSRS — Dynamic scale right save
SCRF — Scale right final
DSRF — Dynamic scale right final
SCRR — Scale right round
DSRR — Dynamic scale right round

Literal operators

LT48 — Load following code word onto top of stack
LT16 — Set top of stack to following 16 bits in code stream
LT8 — Set top of stack to following code syllable
ZERO — Shortcut for LT48 0
ONE — Shortcut for LT48 1

Descriptor operators

INDX — Index create a pointer (copy descriptor) from a base (MOM) descriptor
NXLN — Index and load name (resulting in an indexed descriptor)
NXLV — Index and load value (resulting in a data value)
EVAL — Evaluate descriptor (follow address chain until data word or another descriptor found)

Stack operators

PUSH — Push down stack register
DLET — Pop top of stack
EXCH — Exchange top two words of stack
RSUP — Rotate stack up (top three words)
RSDN — Rotate stack down (top three words)
DUPL — Duplicate top of stack
MKST —Mark stack (build a new stack frame resulting in an MSCW on the top,
— followed by NAMC to load the PCW, then parameter pushes as needed, then ENTR)
IMKS — Insert an MSCW in the B register.
VALC — Fetch a value onto the stack as described above
NAMC — Place an address couple (IRW stack address) onto the stack as described above
STFF — Convert an IRW as placed by NAMC into an SIRW which references data in another stack.
MVST — Move to stack (process switch only done in one place in the MCP)

Store operators

STOD — Store destructive (if the target word has an odd tag throw a memory protect interrupt,
— store the value in the B register at the memory addressed by the A register.
— Delete the value off the stack.
STON — Store non-destructive (Same as STOD but value is not deleted – handy for F := G := H := J expressions).
OVRD — Overwrite destructive, STOD ignoring read-only bit (for use in MCP only)
OVRN — Overwrite non-destructive, STON ignoring read-only bit (for use in MCP only)

Load operators

LOAD — Load the value given by the address (tag 5 or tag 1 word) on the top of stack.
— Follow an address chain if necessary.
LODT — Load transparent – load the word referenced by the address on the top of stack

Transfer operators

These were used for string transfers usually until a certain character was detected in the source string. All these operators are protected from buffer overflows by being limited by the bounds in the descriptors.

TWFD — Transfer while false, destructive (forget pointer)
TWFU — Transfer while false, update (leave pointer at end of transfer for further transfers)
TWTD — Transfer while true, destructive
TWTU — Transfer while true, update
TWSD — Transfer words, destructive
TWSU — Transfer words, update
TWOD — Transfer words, overwrite destructive
TWOU — Transfer words, overwrite update
TRNS — Translate – transfer a source buffer into a destination converting characters as given in a translate table.
TLSD — Transfer while less, destructive
TLSU — Transfer while less, update
TGED — Transfer while greater or equal, destructive
TGEU — Transfer while greater or equal, update
TGTD — Transfer while greater, destructive
TGTU — Transfer while greater, update
TLED — Transfer while less or equal, destructive
TLEU — Transfer while less or equal, update
TEQD — Transfer while equal, destructive
TEQU — Transfer while equal, update
TNED — Transfer while not equal, destructive

TNEU — Transfer while not equal, update
TUND — Transfer unconditional, destructive
TUNU — Transfer unconditional, update

Scan operators

These were used for scanning strings useful in writing compilers. All these operators are protected from buffer overflows by being limited by the bounds in the descriptors.

SWFD — Scan while false, destructive
SISO — String isolate
SWTD — Scan while true, destructive
SWTU — Scan while true, update
SLSD — Scan while less, destructive
SLSU — Scan while less, update
SGED — Scan while greater or equal, destructive
SGEU — Scan while greater or equal, update
SGTD — Scan while greater, destructive
SGTU — Scan while greater, update
SLED — Scan while less or equal, destructive
SLEU — Scan while less or equal, update
SEQD — Scan while equal, destructive
SEQU — Scan while equal, update
SNED — Scan while not equal, destructive
SNEU — Scan while not equal, update

CLSD — Compare characters less, destructive
CLSU — Compare characters less, update
CGED — Compare characters greater or equal, destructive
CGEU — Compare characters greater or equal, update
CGTD — Compare character greater, destructive
CGTU — Compare character greater, update
CLED — Compare characters less or equal, destructive
CLEU — Compare characters less or equal, update
CEQD — Compare character equal, destructive
CEQU — Compare character equal, update
CNED — Compare characters not equal, destructive
CNEU — Compare characters not equal, update

System

SINT — Set interval timer
EEXI — Enable external interrupts
DEXI — Disable external interrupts
SCNI — Scan in – initiate IO read, this changed on different architectures
SCNO — Scan out – initiate IO write, this changed on different architectures

STAG — Set tag (not allowed in user-level processes)
RTAG — Read tag
IRWL — Hardware pseudo operator
SPRR — Set processor register (highly implementation dependent, only used in lower levels of MCP)
RPRR — Read processor register (highly implementation dependent, only used in lower levels of MCP)
MPCW — Make PCW
HALT — Halt the processor (operator requested or some unrecoverable condition has occurred)

Other

VARI — Escape to extended (variable instructions which were less frequent)
OCRX — Occurs index builds an occurs index word used in loops
LLLU — Linked list lookup – Follow a chain of linked words until a certain condition is met
SRCH — Masked search for equal – Similar to LLLU, but testing a mask in the examined words for an equal value
TEED — Table enter edit destructive
TEEU — Table enter edit, update
EXSD — Execute single micro destructive
EXSU — Execute single micro update
EXPU — Execute single micro, single pointer update
NOOP — No operation
NVLD — Invalid operator (hex code FF)
User operators — unassigned operators could cause interrupts into the operating system so that algorithms could be written to provide the required functionality

Edit operators

These were special operators for sophisticated string manipulation, particularly for business applications.

MINS — Move with insert – insert characters in a string
MFLT — Move with float
SFSC — Skip forward source character
SRSC — Skip reverse source characters
RSTF — Reset float
ENDF — End float
MVNU — Move numeric unconditional
MCHR — Move characters
INOP — Insert overpunch
INSG — Insert sign
SFDC — Skip forward destination character
SRDC — Skip reverse destination characters

INSU — Insert unconditional
INSC — Insert conditional
ENDE — End edit

Chapter-9

GPGPU

General-purpose computing on graphics processing units (**GPGPU**, also referred to as **GPGP** and less often **GP²**) is the technique of using a GPU, which typically handles computation only for computer graphics, to perform computation in applications traditionally handled by the CPU. It is made possible by the addition of programmable stages and higher precision arithmetic to the rendering pipelines, which allows software developers to use stream processing on non-graphics data.

GPU improvements

GPU functionality has, traditionally, been very limited. In fact, for many years the GPU was only used to accelerate certain parts of the graphics pipeline. Some improvements were needed before GPGPU became feasible.

Programmability

Programmable vertex and fragment shaders were added to the graphics pipeline to enable game programmers to generate even more realistic effects. Vertex shaders allow the programmer to alter per-vertex attributes, such as position, color, texture coordinates, and normal vector. Fragment shaders are used to calculate the color of a fragment, or per-pixel. Programmable fragment shaders allow the programmer to substitute, for example, a lighting model other than those provided by default by the graphics card, typically simple Gouraud shading. Shaders have enabled graphics programmers to create lens effects, displacement mapping, and depth of field.

The programmability of the pipelines have trended according to Microsoft's DirectX specification , with DirectX 8 introducing Shader Model 1.1, DirectX 8.1 Pixel Shader Models 1.2, 1.3 and 1.4, and DirectX 9 defining Shader Model 2.x and 3.0. Each shader model increased the programming model flexibilities and capabilities, ensuring the conforming hardware follows suit. The DirectX 10 specification introduces Shader

Model 4.0 which unifies the programming specification for vertex, geometry ("Geometry Shaders" are new to DirectX 10) and fragment processing allowing for a better fit for unified shader hardware, thus providing a single computational pool of programmable resource.

Data types

Pre-DirectX 9 graphics cards only supported paletted or integral color types. Various formats are available, each containing a red element, a green element, and a blue element. Sometimes an additional alpha value is added, to be used for transparency. Common formats are:

- 8 bits per pixel – Palette mode, where each value is an index in a table with the real color value specified in one of the other formats. Possibly two bits for red, three bits for green, and three bits for blue.
- 16 bits per pixel – Usually allocated as five bits for red, six bits for green, and five bits for blue.
- 24 bits per pixel – eight bits for each of red, green, and blue
- 32 bits per pixel – eight bits for each of red, green, blue, and alpha

For early fixed-function or limited programmability graphics (i.e. up to and including DirectX 8.1-compliant GPUs) this was sufficient because this is also the representation used in displays. This representation does have certain limitations, however. Given sufficient graphics processing power even graphics programmers would like to use better formats, such as floating point data formats, in order to obtain effects such as high dynamic range imaging. Many GPGPU applications require floating point accuracy, which came with graphics cards conforming to the DirectX 9 specification.

DirectX 9 Shader Model 2.x suggested the support of two precision types: full and partial precision. Full precision support could either be FP32 and FP24 (floating point 24-bit per component) or greater, while partial precision was FP16. ATI's R300 series of GPUs supported FP24 precision only in the programmable fragment pipeline (although FP32 was supported in the vertex processors) while Nvidia's NV30 series supported both FP16 and FP32; other vendors such as S3 Graphics and XGI supported a mixture of formats up to FP24.

Shader Model 3.0 altered the specification, increasing full precision requirements to a minimum of FP32 support in the fragment pipeline. ATI's Shader Model 3.0 compliant R5xx generation (Radeon X1000 series) supports just FP32 throughout the pipeline while Nvidia's NV4x and G7x series continued to support both FP32 full precision and FP16 partial precisions. Although not stipulated by Shader Model 3.0, both ATI and Nvidia's Shader Model 3.0 GPUs introduced support for blendable FP16 render targets, more easily facilitating the support for High Dynamic Range Rendering.

The implementations of floating point on Nvidia GPUs are mostly IEEE compliant; however, this is not true across all vendors. This has implications for correctness which

are considered important to some scientific applications. While 64-bit floating point values (double precision float) are commonly available on CPUs, these are not universally supported on GPUs; some GPU architectures sacrifice IEEE compliance while others lack double-precision altogether. There have been efforts to emulate double-precision floating point values on GPUs; however, the speed tradeoff negates any benefit to offloading the computation onto the GPU in the first place.

Most operations on the GPU operate in a vectorized fashion: a single operation can be performed on up to four values at once. For instance, if one color <R1, G1, B1> is to be modulated by another color <R2, G2, B2>, the GPU can produce the resulting color <R1*R2, G1*G2, B1*B2> in a single operation. This functionality is useful in graphics because almost every basic data type is a vector (either 2-, 3-, or 4-dimensional). Examples include vertices, colors, normal vectors, and texture coordinates. Many other applications can put this to good use, and because of their higher performance, vector instructions (SIMD) have long been available on CPUs.

In November 2006 Nvidia launched CUDA, an SDK and API that allows a programmer to use the C programming language to code algorithms for execution on Geforce 8 series GPUs. AMD offers a similar SDK+API for their ATI-based GPUs, that SDK and technology is called FireStream SDK (formerly a thin hardware interface**.** Close to Metal), designed to compete directly with Nvidia's CUDA. OpenCL from Khronos Group is used paired with OpenGL to unify the C languages extension between different architectures; it supports both Nvidia and AMD/ATI GPUs, and general-purpose CPUs too. GPGPU compared, for example, to traditional floating point accelerators such as the 64-bit CSX700 boards from ClearSpeed that are used in today's supercomputers, current top-end GPUs from Nvidia and AMD emphasize single-precision (32-bit) computation; double-precision (64-bit) computation executes much more slowly.

GPGPU programming concepts

GPUs are designed specifically for graphics and thus are very restrictive in terms of operations and programming. Because of their nature, GPUs are only effective at tackling problems that can be solved using stream processing and the hardware can only be used in certain ways.

Stream processing

GPUs can only process independent vertices and fragments, but can process many of them in parallel. This is especially effective when the programmer wants to process many vertices or fragments in the same way. In this sense, GPUs are stream processors – processors that can operate in parallel by running a single kernel on many records in a stream at once.

A **stream** is simply a set of records that require similar computation. Streams provide data parallelism. **Kernels** are the functions that are applied to each element in the stream. In the GPUs, **vertices** and **fragments** are the elements in streams and vertex and

fragment shaders are the kernels to be run on them. Since GPUs process elements independently there is no way to have shared or static data. For each element we can only read from the input, perform operations on it, and write to the output. It is permissible to have multiple inputs and multiple outputs, but never a piece of memory that is both readable and writable.

Arithmetic intensity is defined as the number of operations performed per word of memory transferred. It is important for GPGPU applications to have high arithmetic intensity else the memory access latency will limit computational speedup.

Ideal GPGPU applications have large data sets, high parallelism, and minimal dependency between data elements.

GPU programming concepts

Computational resources

There are a variety of computational resources available on the GPU:

- Programmable processors – Vertex, primitive, and fragment pipelines allow programmer to perform kernel on streams of data
- Rasterizer – creates fragments and interpolates per-vertex constants such as texture coordinates and color
- Texture Unit – read only memory interface
- Framebuffer – write only memory interface

In fact, the programmer can substitute a write only texture for output instead of the framebuffer. This is accomplished either through Render to Texture (RTT), Render-To-Backbuffer-Copy-To-Texture (RTBCTT), or the more recent stream-out.

Textures as stream

The most common form for a stream to take in GPGPU is a 2D grid because this fits naturally with the rendering model built into GPUs. Many computations naturally map into grids: matrix algebra, image processing, physically based simulation, and so on.

Since textures are used as memory, texture lookups are then used as memory reads. Certain operations can be done automatically by the GPU because of this.

Kernels

Kernels can be thought of as the body of loops. For example, if the programmer were operating on a grid on the CPU they might have code that looked like this:

```
// Input and output grids have 10000 x 10000 or 100 million elements.
```

```
void transform_10k_by_10k_grid(float in[10000][10000], float
out[10000][10000])
{
  for(int x = 0; x < 10000; x++)
  {
    for(int y = 0; y < 10000; y++)
    {
      // The next line is executed 100 million times
      out[x][y] = do_some_hard_work(in[x][y]);
    }
  }
}
```

On the GPU, the programmer only specifies the body of the loop as the kernel and what data to loop over by invoking geometry processing.

Flow control

In sequential code it is possible to control the flow of the program using if-then-else statements and various forms of loops. Such flow control structures have only recently been added to GPUs. Conditional writes could be accomplished using a properly crafted series of arithmetic/bit operations, but looping and conditional branching were not possible.

Recent GPUs allow branching, but usually with a performance penalty. Branching should generally be avoided in inner loops, whether in CPU or GPU code, and various techniques, such as static branch resolution, pre-computation, and Z-cull can be used to achieve branching when hardware support does not exist.

GPU techniques

Map

The map operation simply applies the given function (the kernel) to every element in the stream. A simple example is multiplying each value in the stream by a constant (increasing the brightness of an image). The map operation is simple to implement on the GPU. The programmer generates a fragment for each pixel on screen and applies a fragment program to each one. The result stream of the same size is stored in the output buffer.

Reduce

Some computations require calculating a smaller stream (possibly a stream of only 1 element) from a larger stream. This is called a reduction of the stream. Generally a reduction can be accomplished in multiple steps. The results from the previous step are used as the input for the current step and the range over which the operation is applied is reduced until only one stream element remains.

Stream filtering

Stream filtering is essentially a non-uniform reduction. Filtering involves removing items from the stream based on some criteria.

Scatter

The scatter operation is most naturally defined on the vertex processor. The vertex processor is able to adjust the position of the vertex, which allows the programmer to control where information is deposited on the grid. Other extensions are also possible, such as controlling how large an area the vertex affects.

The fragment processor cannot perform a direct scatter operation because the location of each fragment on the grid is fixed at the time of the fragment's creation and cannot be altered by the programmer. However, a logical scatter operation may sometimes be recast or implemented with an additional gather step. A scatter implementation would first emit both an output value and an output address. An immediately following gather operation uses address comparisons to see whether the output value maps to the current output slot.

Gather

The fragment processor is able to read textures in a random access fashion, so it can gather information from any grid cell, or multiple grid cells, as desired.

Sort

The sort operation transforms an unordered set of elements into an ordered set of elements. The most common implementation on GPUs is using sorting networks.

Search

The search operation allows the programmer to find a particular element within the stream, or possibly find neighbors of a specified element. The GPU is not used to speed up the search for an individual element, but instead is used to run multiple searches in parallel.

Data structures

A variety of data structures can be represented on the GPU:

- Dense arrays
- Sparse arrays – static or dynamic
- Adaptive structures

Applications

The following are some of the areas where GPUs have been used for general purpose computing:

- MATLAB acceleration using the Parallel Computing Toolbox and MATLAB Distributed Computing Server, as well as 3rd party packages like Jacket.
- k-nearest neighbor algorithm
- Computer clusters or a variation of a parallel computing (utilizing GPU cluster technology) for highly calculation-intensive tasks:
    - High-performance computing clusters (HPC clusters) (often referred to as supercomputers)
        - including cluster technologies like Message Passing Interface, and single-system image (SSI), distributed computing, and Beowulf
    - Grid computing (a form of distributed computing) (networking many heterogeneous computers to create a virtual computer architecture)
    - Load-balancing clusters (sometimes referred to as a server farm)
- Physical based simulation and physics engines (usually based on Newtonian physics models)
    - Conway's Game of Life, cloth simulation, incompressible fluid flow by solution of Navier-Stokes equations
- Statistical physics
    - Ising model
- Lattice gauge theory
- Segmentation – 2D and 3D
- Level-set methods
- CT reconstruction
- Fast Fourier transform
- Tone mapping
- Audio signal processing
    - Audio and Sound Effects Processing, to use a GPU for DSP (digital signal processing)
    - Analog signal processing
    - Speech processing
- Digital image processing
- Video Processing
    - Hardware accelerated video decoding and post-processing
        - Motion compensation (mo comp)
        - Inverse discrete cosine transform (iDCT)
        - Variable-length decoding (VLD)
        - Inverse quantization (IQ)
        - In-loop deblocking
        - Bitstream processing (CAVLC/CABAC) using special purpose hardware for this task because this is a serial task not suitable for regular GPGPU computation
        - Deinterlacing

        - Spatial-temporal de-interlacing
      - Noise reduction
      - Edge enhancement
      - Color correction
    - Hardware accelerated video encoding and pre-processing
- Raytracing
- Global illumination – photon mapping, radiosity, subsurface scattering
- Geometric computing – constructive solid geometry, distance fields, collision detection, transparency computation, shadow generation
- Scientific computing
    - Monte Carlo simulation of light propagation
    - Weather forecasting
    - Climate research
    - Molecular modeling on GPU
    - Quantum mechanical physics
    - Astrophysics
- Bioinformatics
- Computational finance
- Medical imaging
- Computer vision
- Digital signal processing / signal processing
- Control engineering
- Neural networks
- Database operations
- Lattice Boltzmann methods
- Cryptography and cryptanalysis
    - Implementation of MD6
    - Implementation of AES
    - Implementation of DES
    - Implementation of RSA
    - Implementation of ECC
    - Password cracking
- Electronic Design Automation
- Antivirus software
- Intrusion Detection

Chapter-10

MikroSim and Memory Barrier

MikroSim

The program **MikroSim** is an educational software for hardware-non-specific explanation of the general functioning and behaviour of a virtual processor, running on the operating system Microsoft Windows. With this e-learning tool, devices like miniaturized calculators, microcontroller, microprocessors, and computer can be explained didactically on own-developed instruction code on register transfer level, which are controlled by program sequences of micro instructions (microcode). Based on this, on higher level of abstraction it is possible to develop an instruction set being able to control a virtual application board.

General

Initially, MikroSim was developed with the intention to be a processor simulation software, which is available manifold in educational areas. Since MikroSim operability starts on the basis of microcode development defined as a sequence of micro instructions (microcoding) for a virtual control unit, the software's intention is on first approach a microcode simulator with various levels of abstractions including the ability of CPU simulators and instruction set emulators. In the current software revision, even a microcode controlled virtual application is feasible to operate on own coded instruction sets. With MikroSim typical and well known concepts in the area of computer engineering like computer architecture and instruction set architecture are unspecifically treated, which have been established since the early days of the information era and being still valid. In this fashion the simulation software gains a timeless, free didactical benefit without being restricted on special developments of the past and in the future. The detailed documentation and the bilingual application's graphical user interface (GUI) in German and English, as well as the software's upward compatibility given to some extend by MicroSoft's operating system Windows, are reasons for being a well established valuable e-learning tool in field of computer engineering since 1992 for educational use.

History of development

The software bases on a revision written under Turbo-Pascal compiled for MS-DOS operating systems, which has been used for educational use in computer engineering and computer science at the Philipps-University Marburg (Germany) until 1992. The didactical concept has been picked up by Martin Perner during his study of physics (1990–1995) in summer 1992, revised and converted into a windows application compiled with Microsoft Visual Basic and running on Windows 3.1x. In doing so, at this time a simulator with huge conceptual improvements arose by exploiting the novel functionality and utilisation of Windows' GUI for supporting the composition of microcode and the traceability of its instructional influence. The enhancements of the e-learning tool under Windows has been supported and promoted by the Fachbereich Mathematik/Informatik of the University of Marburg by Heinz-Peter Gumm until end 1995.

The Simulator has been awarded with the ''European Academic Software Award 1994'' in the category computer science in Heidelberg (Germany) in November 1994. In March 1995 the simulator has been presented at the computer exhibition CeBIT'95 in Hannover at the stand of the ''Hessischen Hochschulen''. Between 1995 until the year 2000 the simulator has been published as ''Mikrocodesimulator MikroSim 1.2'' without any significant improvements. At this time the tool has been awarded by the European Union in the context of the ''European Year of Livelong Learning 1996'' with 1000 ECU. In 1997, the software has been presented at the contest ''Multimedia Transfer'97'' in connection to the exhibition ''LearnTec'97''. In its penultimate revision, the simulator has been published under ''Mikrocodesimulator MikroSim2000'', optimized for Windows95's 32-bit operation.
Between 2008 and 2009, the simulator concept has been revised, reworked, and thoughtful extended. So it has received wide ranging improvements and extensions without touching the successful conceptual aspects of the microcode simulation abilities in the core. For this purpose, advantage is taken of today's computing system's performance determined by operating system and underlying computational power to extend MikroSim's simulation possibilities up to the stage of a virtual application board. Since January 2010, the simulator is distributed as ''Mikrocodesimulator MikroSim 2010'' by 0/1-SimWare.

Compatibility

MikroSim is compiled and optimized for sake of unrestricted compatibility and for widest distribution possible for Windows XP as a 32-bit version. The program runs on all 32 - and 64-bit operating systems of the Windows Vista and Windows 7. Thereby, no special XP compatibility mode is needed.

Licensing and distribution

The MikroSim can be used as freely distributable demo version with only few limitations in some in special functionalities negligible for private use and demonstration purpose.

Most of the features are unlocked for private study and personal interests on crash course level. The demo version's limitations concern educational aspects getting useful when using the tool extensively on long term. However, the software can be activated anonymously online without complicated registration process for a testing period of 10 days. For unlimited functionality, 0/1-SimWare offers several licensing models for single and multi user registration for private users, high schools, vocational schools, colleges, and universities.

Functionality

The windows application allows for the gradual establishment of a virtual application that is predetermined and such unchangeable in its functionality.

In exploration mode, the operating principle and control of newly added components influenced by one microcode instruction within a cycle can be evaluated. The width of MikroSim's micro instructions is 49 bits. A single micro instruction is executed in three phases of a 3-phase clock. The partial phases are referred to as "GET", "CALCULATE" and "PUT" phase, causing to fetch some register value, to execute a 32-bit calculation, and to store the calculation result into a CPU's internal register, finally.

In simulation mode, seamlessly executed micro instructions control the central processing unit of the simulator in subsequent cycles. Therefore, the intrinsic ability of one micro instruction is utilized to address the next micro instruction in the control store. The control store holding the micro instruction set (commonly referred as "microcode") comprises 1024 micro instructions words each 49-bit wide.

Using structuring opportunities of the control store for addressable scheduling of the microcode and the implementation of a cyclically operating machine code interpreter, that is programmed in microcode as well allows the implementation of individual micro-operation sequences, known as machine instructions. The microcode can be regarded as firmware for MikroSim, that can be modified, and stored in and reloaded from a microcode-ROM-file.

Within a micro instruction execution cycle, the CPU as well as an input / output controller is connected to an external 16 kByte huge random access memory device (RAM). Via the input-output controller device, communication with virtual input and output devices is supported by Direct Memory Access mode (DMA), Inter-Integrated Circuit Connection (I2C), and Interrupt request functionality (IRQ]]. A output port, a display, a timer, an event trigger, a digital-analog converter, a keyboard and data input / output channel is provided as virtual IC device for explaining didactically the communication with external devices.

The microcode simulator uses eight freely usable register each 32-bit wide connected with a 32-bit arithmetic logic unit (ALU). The register content can be regarded as signed or unsigned integer values, or as 32-bit floating point numbers. The register content can be easily viewed, interpreted, and modified bitwise an integrated system number editor.

The 32-bit ALU is the key unit of the central processing unit. It supports 128 different basic arithmetic operations for integer operation, interrupt control, and for floating point arithmetic.

The didactical approach to floating point calculations, which has been introduced in a comparable manner already in the early 1940s by Konrad Zuse, is introduced by using elemental sublevel operations for exponent and mantissa involved in the key operations of addition/subtraction and multiplication/division. A set of powerful 32-bit floating point arithmetic commands in mantissa and exponent for the basic operations and elementary analytical functions are provided, as they are realized in today's mathematical coprocessors. Here, in the simulation with MikroSim it is ideally assumed that the execution of each supported ALU arithmetic operation requires only a distinct computing duration independent of circuit complexity realistically needed in practice.

The execution of micro instructions can be operated on various simulation levels with different temporal resolution:

- In the lowest simulation level, the simulator supports the phased wise execution of GET, CALCULATE, and PUT phase. The processing of the partial phases is possible with an adjustable delay for better traceability.
- In next upper level, the current micro instruction is executed in a complete three-phase clock without time delay. A continuous execution of several 3-phase clock cycles is supported within a so-called "Load Increment Execute" (LIE) cycle. The LIE cycle regarded as an interpreter written in microcode has the function to load machine instructions coded as byte value from the external RAM and to let branch the micro instruction sequence to the referenced microcode subroutine for execution given by the opcode and returning to the LIE back to retrieve the next machine instruction.
- One execution level higher, a sequence of several machine instructions are executable until a user-defined break point is reached, which is placed in the machine code sequence. It is possible to measure run times between break points. So it is possible to benchmark execution performance on machine and microcode level.
- In the top most simulation level the microcode simulator continuously executes micro instructions without interrupt. In this level, machine instruction by machine instruction is loaded. So, it is possible to focus on the interaction of the CPU with external devices..

With various additional options, visual CPU activities can be suppressed for the benefit of increasing the processing speed when the control of the application by machine programming is put forward. The performance index monitor provided with the simulator enables the user to benchmark the processing performance of MikroSim and setting it into relation with computing power of the simulator's hardware, measurable in floating-point operations per second (FLOPS) and instructions per second (IPS).

With the so-called ''Basic Assembler Tool for MikroSim'' MikroBAT, simple programs can be developed in assembler programming language. Here, all supported mnemonics of the assembler programming language are determined by the user's self created machine's instruction set on micro instruction level. The add-on tool is able to translate the assembly language program into machine code and data and transferring the binary code into the external RAM for subsequent simulations. Together with MikroBAT the microcode simulator MikroSim supports the didactical introduction of teaching aspects in technical computer science from a switch-controlled calculating machine to a assembler programmable application.

Memory barrier

Memory barrier, also known as **membar** or **memory fence** or **fence instruction**, is a type of barrier and a class of instruction which causes a central processing unit (CPU) or compiler to enforce an ordering constraint on memory operations issued before and after the barrier instruction.

CPUs employ performance optimizations that can result in out-of-order execution. The reordering of memory operations (loads and stores) normally goes unnoticed within a single thread of execution, but causes unpredictable behaviour in concurrent programs and device drivers unless carefully controlled. The exact nature of an ordering constraint is hardware dependent, and defined by the architecture's memory ordering model. Some architectures provide multiple barriers for enforcing different ordering constraints.

Memory barriers are typically used when implementing low-level machine code that operates on memory shared by multiple devices. Such code includes synchronization primitives and lock-free data structures on multiprocessor systems, and device drivers that communicate with computer hardware.

An illustrative example

When a program runs on a single CPU, the hardware performs the necessary bookkeeping to ensure that programs execute as if all memory operations were performed in the order specified by the programmer (*program order*), hence memory barriers are not necessary. However, when the memory is shared with multiple devices, such as other CPUs in a multiprocessor system, or memory mapped peripherals, out-of-order access may affect program behavior. For example, a second CPU may see memory changes made by the first CPU in a sequence which differs from program order.

The following two processor programs give a concrete example of how such out-of-order execution can affect program behavior:

Initially, memory locations `x` and `f` both hold the value `0`. The program running on processor #1 loops while the value of `f` is zero, then it prints the value of `x`. The program running on processor #2 stores the value `42` into `x` and then stores the value `1` into `f`.

Pseudo-code for the two program fragments is shown below. The steps of the program correspond to individual processor instructions.

```
Processor #1:
 loop:
  load the value in location f, if it is 0 goto loop
 print the value in location x

Processor #2:
 store the value 42 into location x
 store the value 1 into location f
```

You might expect the print statement to always print the number "42"; however, if processor #2's store operations are executed out-of-order, it is possible for `f` to be updated *before* `x`, and the print statement might therefore print "0". For most programs this situation is not acceptable. A memory barrier can be inserted before processor #2's assignment to `f` to ensure that the new value of `x` is visible to other processors at or prior to the change in the value of `f`.

Low-level architecture-specific primitives

Memory barriers are low-level primitives which are part of the definition of an architecture's memory model. Like instruction sets, memory models vary considerably between architectures, so it is not appropriate to generalize about memory barrier behavior. The conventional wisdom is that using memory barriers correctly requires careful study of the architecture manuals for the hardware being programmed. That said, the following paragraph offers a glimpse of some memory barriers which exist in contemporary products.

Some architectures, including the ubiquitous x86/x64, provide several memory barrier instructions including an instruction sometimes called "full fence". A full fence ensures that all load and store operations prior to the fence will have been committed prior to any loads and stores issued following the fence. Other architectures, such as the Itanium, provide separate "acquire" and "release" memory barriers which address the visibility of read-after-write operations from the point of view of a reader (sink) or writer (source) respectively. Some architectures provide separate memory barriers to control ordering between different combinations of system memory and I/O memory. When more than one memory barrier instruction is available it is important to consider that the cost of different instructions may vary considerably.

Multithreaded programming and memory visibility

Multithreaded programs usually use synchronization primitives provided by a high-level programming environment, such as Java and .Net Framework, or an API such as POSIX Threads or Win32. Primitives such as mutexes and semaphores are provided to synchronize access to resources from parallel threads of execution. These primitives are usually implemented with the memory barriers required to provide the expected memory

visibility semantics. In such environments explicit use of memory barriers is not generally necessary.

Each API or programming environment in principle has its own high-level memory model that defines its memory visibility semantics. Although programmers do not usually need to use memory barriers in such high level environments, it is important to understand their memory visibility semantics, to the extent possible. Such understanding is not necessarily easy to achieve because memory visibility semantics are not always consistently specified or documented.

Just as programming language semantics are defined at a different level of abstraction than machine language opcodes, a programming environment's memory model is defined at a different level of abstraction than that of a hardware memory model. It is important to understand this distinction and realize that there is not always a simple mapping between low-level hardware memory barrier semantics and the high-level memory visibility semantics of a particular programming environment. As a result, a particular platform's implementation of (say) POSIX Threads may employ stronger barriers than required by the specification. Programs which take advantage of memory visibility as-implemented rather than as-specified may not be portable.

Out-of-order execution versus compiler reordering optimizations

Memory barrier instructions only address reordering effects at the hardware level. Compilers may also reorder instructions as part of the program optimization process. Although the effects on parallel program behavior can be similar in both cases, in general it is necessary to take separate measures to inhibit compiler reordering optimizations for data that may be shared by multiple threads of execution. Note that such measures are usually only necessary for data which is not protected by synchronization primitives such as those discussed in the previous section.

In C and C++, the `volatile` keyword was intended to allow C and C++ programs to directly access memory-mapped I/O. Memory-mapped I/O generally requires that the reads and writes specified in source code happen in the exact order specified with no omissions. Omissions or reorderings of reads and writes by the compiler would break the communication between the program and the device accessed by memory-mapped I/O. A C or C++ compiler may not reorder reads and writes to volatile memory locations, nor may it omit a read or write to a volatile memory location, allowing a pointer to volatile memory to be used for memory-mapped I/O.

The C and C++ standards do not address multiple threads (or multiple processors), and as such, the usefulness of `volatile` depends on the compiler and hardware. Although `volatile` guarantees that the volatile reads and volatile writes will happen in the exact order specified in the source code, the compiler may generate code (or the CPU may re-order execution) such that a volatile read or write is reordered with regard to non-volatile reads or writes, thus limiting its usefulness as an inter-thread flag or mutex. Preventing such is compiler specific, but some compilers, like gcc, will not reorder operations

around in-line assembly code with `volatile` and `"memory"` tags, like in: `asm volatile ("" : : : "memory")`. Moreover, you are not guaranteed that volatile reads and writes will be seen in the same order by other processors due to caching, cache coherence protocol and relaxed memory ordering, meaning volatile variables alone may not even work as inter-thread flags or mutexes.

Some languages and compilers may provide sufficient facilities to implement functions which address both the compiler reordering and machine reordering issues. In Java version 1.5 (also known as version 5), the `volatile` keyword is now guaranteed to prevent certain hardware *and* compiler re-orderings, as part of the new Java Memory Model. The proposed C++ memory model does not use `volatile`, instead C++0x will include special atomic types and operations with semantics similar to those of `volatile` in the Java Memory Model.

Chapter-11

Very Long Instruction Word

Very long instruction word or **VLIW** refers to a CPU architecture designed to take advantage of instruction level parallelism (ILP). A processor that executes every instruction one after the other (i.e. a non-pipelined scalar architecture) may use processor resources inefficiently, potentially leading to poor performance. The performance can be improved by executing different sub-steps of sequential instructions simultaneously (this is *pipelining*), or even executing multiple instructions entirely simultaneously as in superscalar architectures. Further improvement can be achieved by executing instructions in an order different from the order they appear in the program; this is called out-of-order execution.

As often implemented, these three techniques all come at a cost: increased hardware complexity. Before executing any operations in parallel, the processor must verify that the instructions do not have interdependencies. For example a first instruction's result is used as an second instruction's input. Clearly, they cannot execute at the same time, and the second instruction can't be executed before the first. Modern out-of-order processors have increased the hardware resources which do the scheduling of instructions and determining of interdependencies.

The VLIW approach, on the other hand, executes operations in parallel based on a fixed schedule determined when programs are compiled. Since determining the order of execution of operations (including which operations can execute simultaneously) is handled by the compiler, the processor does not need the scheduling hardware that the three techniques described above require. As a result, VLIW CPUs offer significant computational power with less hardware complexity (but greater compiler complexity) than is associated with most superscalar CPUs.

As is the case with any novel architectural approach, the concept is only as useful as code generation makes it. That is, the fact that a number of special-purpose instructions are available to facilitate certain complicated operations—say, fast Fourier transform (FFT) computation or certain calculations that recur in tomographic contexts—is useless if

compilers are unable to spot relevant source code constructs and generate target code that duly utilizes the CPU's advanced offerings. *A fortiori*, the programmer must be able to express his algorithms in a manner that makes the compiler's task easier.

Design

In superscalar designs, the number of execution units is invisible to the instruction set. Each instruction encodes only one operation. For most superscalar designs, the instruction width is 32 bits or fewer. VLIW is a type of MIMD.

In contrast, one VLIW instruction encodes multiple operations; specifically, one instruction encodes at least one operation for each execution unit of the device. For example, if a VLIW device has five execution units, then a VLIW instruction for that device would have five operation fields, each field specifying what operation should be done on that corresponding execution unit. To accommodate these operation fields, VLIW instructions are usually at least 64 bits wide, and on some architectures are much wider.

For example, the following is an instruction for the SHARC. In one cycle, it does a floating-point multiply, a floating-point add, and two autoincrement loads. All of this fits into a single 48-bit instruction.

```
f12=f0*f4, f8=f8+f12, f0=dm(i0,m3), f4=pm(i8,m9);
```

Since the earliest days of computer architecture, some CPUs have added several additional arithmetic logic units (ALUs) to run in parallel. Superscalar CPUs use *hardware* to decide which operations can run in parallel. VLIW CPUs use *software* (the compiler) to decide which operations can run in parallel. Because the complexity of instruction scheduling is pushed off onto the compiler, the hardware's complexity can be substantially reduced.

A similar problem occurs when the result of a parallelisable instruction is used as input for a branch. Most modern CPUs "guess" which branch will be taken even before the calculation is complete, so that they can load up the instructions for the branch, or (in some architectures) even start to compute them speculatively. If the CPU guesses wrong, all of these instructions and their context need to be "flushed" and the correct ones loaded, which is time-consuming.

This has led to increasingly complex instruction-dispatch logic that attempts to guess correctly, and the simplicity of the original RISC designs has been eroded. VLIW lacks this logic, and therefore lacks its power consumption, possible design defects and other negative features.

In a VLIW, the compiler uses heuristics or profile information to guess the direction of a branch. This allows it to move and preschedule operations speculatively before the branch is taken, favoring the most likely path it expects through the branch. If the branch

goes the unexpected way, the compiler has already generated compensatory code to discard speculative results to preserve program semantics.

The acronym **VLIW** may also refer to **Variable Length Instruction Word**, a CPU instruction set designed to load (or copy) a literal value count of inline Machine code to the on-chip RAM for higher speed CPU decoding.

History

The term *VLIW*, and the concept of VLIW architecture itself, were invented by Josh Fisher in his research group at Yale University in the early 1980s. His original development of trace scheduling as a compilation technique for VLIW was developed when he was a graduate student at New York University. Prior to VLIW, the notion of prescheduling functional units and instruction-level parallelism in software was well established in the practice of developing horizontal microcode. Fisher's innovations were around developing a compiler that could target horizontal microcode from programs written in an ordinary programming language. He realized that to get good performance and target a wide-issue machine, it would be necessary to find parallelism beyond that generally within a basic block. He developed region scheduling techniques to identify parallelism beyond basic blocks. Trace scheduling is such a technique, and involves scheduling the most likely path of basic blocks first, inserting compensation code to deal with speculative motions, scheduling the second most likely trace, and so on, until the schedule is complete.

Fisher's second innovation was the notion that the target CPU architecture should be designed to be a reasonable target for a compiler — the compiler and the architecture for VLIW must be co-designed. This was partly inspired by the difficulty Fisher observed at Yale of compiling for architectures like Floating Point Systems' FPS164, which had a complex instruction set architecture (CISC) that separated instruction initiation from the instructions that saved the result, requiring very complicated scheduling algorithms. Fisher developed a set of principles characterizing a proper VLIW design, such as self-draining pipelines, wide multi-port register files, and memory architectures. These principles made it easier for compilers to write fast code.

The first VLIW compiler was described in a Ph.D. thesis by John Ellis, supervised by Fisher. The compiler was christened Bulldog, after Yale's mascot. John Ruttenberg also developed certain important algorithms for scheduling.

Fisher left Yale in 1984 to found a startup company, Multiflow, along with co-founders John O'Donnell and John Ruttenberg. Multiflow produced the TRACE series of VLIW minisupercomputers, shipping their first machines in 1987. Multiflow's VLIW could issue 28 operations in parallel per instruction. The TRACE system was implemented in an MSI/LSI/VLSI mix packaged in cabinets, a technology that fell out of favor when it became more cost-effective to integrate all of the components of a processor (excluding memory) on a single chip. Multiflow was too early to catch the following wave, when chip architectures began to allow multiple issue CPUs. The major semiconductor

companies recognized the value of Multiflow technology in this context, so the compiler and architecture were subsequently licensed to most of these companies.

Implementations

Cydrome was a company producing VLIW numeric processors using ECL technology in the same timeframe (late 1980s). This company, like Multiflow, went out of business after a few years.

One of the licensees of the Multiflow technology is Hewlett-Packard, which Josh Fisher joined after Multiflow's demise. Bob Rau, founder of Cydrome, also joined HP after Cydrome failed. These two would lead computer architecture research within Hewlett-Packard during the 1990s.

In addition to the above systems, at around the same period (i.e. 1989-1990), Intel implemented VLIW in the Intel i860, their first 64bit microprocessor; the i860 was also the first processor to implement VLIW on a single chip. This processor could operate in both simple RISC mode and VLIW mode:

In the early 1990s, Intel introduced the i860 RISC microprocessor. This simple chip had two modes of operation: a scalar mode and a VLIW mode. In the VLIW mode, the processor always fetched two instructions and assumed that one was an integer instruction and the other floating-point.

The i860's VLIW mode was used extensively in embedded DSP applications since the application execution and datasets were simple, well ordered and predictable, allowing the designer to take full advantage of the parallel execution advantages that VLIW lent itself to; in VLIW mode the i860 was able to maintain floating-point performance in the range of 20-40 double-precision MFLOPS (an extremely high figure for its time and for a processor operating at 25-50Mhz).

In the 1990s, Hewlett-Packard researched this problem as a side effect of ongoing work on their PA-RISC processor family. They found that the CPU could be greatly simplified by removing the complex dispatch logic from the CPU and placing it into the compiler. Today's compilers are much more complex than those from the 1980s, so the added complexity in the compiler was considered to be a small cost.

VLIW CPUs are usually constructed of multiple RISC-like functional units that operate independently. Contemporary VLIWs typically have four to eight main functional units. Compilers generate initial instruction sequences for the VLIW CPU in roughly the same manner that they do for traditional CPUs, generating a sequence of RISC-like instructions. The compiler analyzes this code for dependence relationships and resource requirements. It then schedules the instructions according to those constraints. In this process, independent instructions can be scheduled in parallel. Because VLIWs typically represent instructions scheduled in parallel with a longer instruction word that

incorporates the individual instructions, this results in a much longer opcode (thus the term "very long") to specify what executes on a given cycle.

Examples of contemporary VLIW CPUs include the TriMedia media processors by NXP (formerly Philips Semiconductors), the SHARC DSP by Analog Devices, the C6000 DSP family by Texas Instruments, and the STMicroelectronics ST200 family based on the Lx architecture (also designed by Josh Fisher). These contemporary VLIW CPUs are primarily successful as embedded media processors for consumer electronic devices.

VLIW features have also been added to configurable processor cores for SoC designs. For example, Tensilica's Xtensa LX2 processor incorporates a technology dubbed FLIX (Flexible Length Instruction eXtensions) that allows multi-operation instructions. The Xtensa C/C++ compiler can freely intermix 32- or 64-bit FLIX instructions with the Xtensa processor's single-operation RISC instructions, which are 16 or 24 bits wide. By packing multiple operations into a wide 32- or 64-bit instruction word and allowing these multi-operation instructions to be intermixed with shorter RISC instructions, FLIX technology allows SoC designers to realize VLIW's performance advantages while eliminating the code bloat of early VLIW architectures. The Infineon Carmel DSP is another VLIW processor core intended for SoC; it uses a similar code density improvement technique called "configurable long instruction word" (CLIW).

Outside embedded processing markets, Intel's Itanium IA-64 EPIC appears as the only example of a widely used VLIW CPU architecture. However, EPIC architecture is sometimes distinguished from a pure VLIW architecture, since EPIC advocates full instruction predication, rotating register files, and a very long instruction word that can encode non-parallel instruction groups. VLIWs have, however, gained significant consumer penetration in the GPU market. In particular, the ATI/AMD Radeon R600 family of GPU architectures (including the R700 and Cypress derivatives) are VLIWs.

Backward compatibility

When silicon technology allowed for wider implementations (with more execution units) to be built, the compiled programs for the earlier generation would not run on the wider implementations, as the encoding of the binary instructions depended on the number of execution units of the machine.

Transmeta addresses this issue by including a binary-to-binary software compiler layer (termed *Code Morphing*) in their Crusoe implementation of the x86 architecture. Basically, this mechanism is advertised to recompile, optimize, and translate x86 opcodes at runtime into the CPU's internal machine code. Thus, the Transmeta chip is *internally* a VLIW processor, effectively decoupled from the x86 CISC instruction set that it executes.

Intel's Itanium architecture (among others) solved the backward-compatibility problem with a more general mechanism. Within each of the multiple-opcode instructions, a bit field is allocated to denote dependency on the previous VLIW instruction within the

program instruction stream. These bits are set at compile time, thus relieving the hardware from calculating this dependency information. Having this dependency information encoded into the instruction stream allows wider implementations to issue multiple non-dependent VLIW instructions in parallel per cycle, while narrower implementations would issue a smaller number of VLIW instructions per cycle.

Another perceived deficiency of VLIW architectures is the code bloat that occurs when not all of the execution units have useful work to do and thus have to execute NOPs. This occurs when there are dependencies in the code and the functional pipelines must be allowed to drain before subsequent operations can proceed.

Since the number of transistors on a chip has grown, the perceived disadvantages of the VLIW have diminished in importance. The VLIW architecture is growing in popularity, particularly in the embedded market, where it is possible to customize a processor for an application in an embedded system-on-a-chip. Embedded VLIW products are available from several vendors, including the FR-V from Fujitsu, the BSP15/16 from Pixelworks, the ST231 from STMicroelectronics, the TriMedia from NXP, the CEVA-X DSP from CEVA, the Jazz DSP from Improv Systems, and Silicon Hive. The Texas Instruments TMS320 DSP line has evolved, in its C6xxx family, to look more like a VLIW, in contrast to the earlier C5xxx family.

Chapter-12

Orthogonal Instruction Set and Out-of-Order Execution

Orthogonal instruction set

Orthogonal instruction set is a term used in computer engineering. A computer's instruction set is said to be orthogonal if any instruction can use data of any type via any addressing mode. The word *orthogonal*, which means *right angle* in this context, implies that it is possible to move along one axis (the operations) independently of the other axis (the addressing modes) and vice versa. This meaning is similar, but not identical, to the meaning of the word in pure mathematics.

Orthogonality in practice

In many CISC computers, an instruction could access either registers or memory, usually in several different ways. This made the CISC machines easier to program, because rather than being required to remember thousands of individual instruction opcodes, an orthogonal instruction set allowed a programmer to instead remember just thirty to a hundred operation codes ("ADD", "SUBTRACT", "MULTIPLY", "DIVIDE", etc.) and a set of three to ten addressing modes ("FROM REGISTER 0", "FROM REGISTER 1", "FROM MEMORY", etc.). The DEC PDP-11 and Motorola 68000 computer architectures are examples of nearly orthogonal instruction sets, while the ARM11 and VAX are examples of CPUs with fully orthogonal instruction sets.

The PDP-11

With the exception of its floating point instructions, the PDP-11 was very strongly orthogonal. Every integer instruction could operate on either 1-byte or 2-byte integers and could access data stored in registers, stored as part of the instruction, stored in memory, or stored in memory and pointed to by addresses in registers. Even the PC and the stack pointer could be affected by the ordinary instructions using all of the ordinary

data modes. In fact, "immediate" mode (hardcoded numbers within an instruction, such as ADD #4, R1 (R1 = R1 + 4) was implemented as the mode "register indirect, autoincrement" and specifying the program counter (R7) as the register to use reference for indirection and to autoincrement.

Since the PDP-11 was an octal-oriented (3-bit sub-byte) machine (addressing modes 0 - 7, registers R0 - R7), there were (electronically) 8 addressing modes. Through the use of the Stack Pointer (R6) and Program Counter (R7) as referenceable registers, there were 10 conceptual addressing modes available.

The VAX-11

The VAX-11 extended the PDP-11's orthogonality to all data types, including floating point numbers (although instructions such as 'ADD' was divided into data-size dependent variants such as ADDB, ADDW, ADDL, ADDP, ADDF for add byte, word, longword, packed BCD and single-precision floating point, respectively). Like the PDP-11, the Stack Pointer and Program Counter were in the general register file (R14 and R15).

The general form of a VAX 11 instruction would be:

opcode [operand] [operand] ...

Each component being one byte, the opcode a value in the range 0 - 255, and each operand consisting of two nibbles, the upper 4 bits specifying an addressing mode, and the lower 4 bits (usually) specifying a register number (R0 - R15).

Unlike the octal-oriented PDP-11, the VAX-11 was a hexadecimal-oriented machine (4-bit sub-byte). This resulted in 16 logical addressing modes (0-15), however, addressing modes 0-3 were "short immediate" for immediate data of 6 bits or less (the 2 low-order bits of the addressing mode being the 2 high-order bits of the immediate data, when prepended to the remaining 4 bits in that data-addressing byte). Since addressing modes 0-3 were identical, this made 13 (electronic) addressing modes, but as in the PDP-11, the use of the Stack Pointer (R14) and Program Counter (R15) created a total of over 15 conceptual addressing modes (with the assembler program translating the source code into the actual stack-pointer or program-counter based addressing mode needed).

The MC68000

By comparison, Motorola's designers attempted to make the assembly language orthogonal while the underlying machine language was somewhat less so. Compared to the PDP-11, the MC68000 used separate registers to store instructions and the addresses of data in memory; the assembly language "hid" some of this separation from the programmer. Many programmers disliked the "near" orthogonality, while others were grateful for the attempt.

At the bit level, the person writing the assembler (or debugging machine code) would clearly see that these "instructions" could become any of several different op-codes. It was quite a good compromise because it gave almost the same convenience as a truly orthogonal machine, and yet also gave the CPU designers freedom to use the bits in the instructions more efficiently than a purely-orthogonal approach might have.

The 8080 and follow on designs

The 8-bit Intel 8080 (as well as the 8085 and 8051) microprocessor was basically a slightly extended accumulator-based design and therefore not orthogonal. An assembly-language programmer or compiler writer had to be mindful of which operations were possible on each register: Most 8-bit operations could be performed only on the 8-bit accumulator (the A-register), while 16-bit operations could be performed only on the 16-bit pointer/accumulator (the HL-register pair), whereas simple operations, such as increment, were possible on all seven 8-bit registers. This was largely due to a desire to keep all opcodes one byte long and to maintain source code compatibility with the original Intel 8008 (an LSI-implementation of the Datapoint 2200's CPU).

The binary-compatible Z80 later added prefix-codes to escape from this 1-byte limit and allow for a more powerful instruction set. The same basic idea was employed for the Intel 8086, although, to allow for more radical extensions, *binary*-compatibility with the 8080 was not attempted here; instead the 8086 was designed as a more regular and fully 16-bit processor that was *source*-compatible with the 8008, 8080, and 8085. It maintained some degree of non-orthogonality for the sake of high code density (even though this was derided as being "baroque" by some computer scientists at the time). The 32-bit extension of this architecture that was introduced with the 80386, was, by all practical means, fully orthogonal, despite keeping all the 8086 instructions and their extended counterparts. However, the *encoding-strategy* used still shows many traces from the 8008 and 8080 (and Z80); for instance, single-byte encodings remain for certain frequent operations such as **push** and **pop** of registers and constants, and the primary accumulator, **eax**, employ shorter encodings than the other registers on certain types of operations; observations like this are sometimes exploited for code optimization in both compilers and hand written code.

Into the RISC age

A fully orthogonal architecture may not be the most "bit efficient" architecture. In the late 1970s research at IBM (and similar projects elsewhere) demonstrated that the majority of these "orthogonal" addressing modes were ignored by most programs. Perhaps some of the bits that were used to express the fully orthogonal instruction set could instead be used to express more virtual address bits or select from among more registers.

In the RISC age, computer designers strove to achieve a balance that they thought better. In particular, most RISC computers, while still being highly orthogonal with regard to which instructions can process which data types, now have reverted to "load/store" architectures. In these architectures, only a very few *memory reference instructions* can

access main memory and only for the purpose of loading data into registers or storing register data back into main memory; only a few addressing modes may be available, and these modes may vary depending on whether the instruction refers to data or involves a transfer of control (jump). Conversely, data must be in registers before it can be operated upon by the other instructions in the computer's instruction set. This trade off is made explicitly to enable the use of much larger register sets, extended virtual addresses, and longer *immediate data* (data stored directly within the computer instruction).

Out-of-order execution

In computer engineering, **out-of-order execution** (**OoOE** or **OOE**) is a paradigm used in most high-performance microprocessors to make use of instruction cycles that would otherwise be wasted by a certain type of costly delay. In this paradigm, a processor executes instructions in an order governed by the availability of input data, rather than by their original order in a program. In doing so, the processor can avoid being idle while data is retrieved for the next instruction in a program, processing instead the next instructions which is able to run immediately.

History

Out-of-order execution is a restricted form of data flow computation, which was a major research area in computer architecture in the 1970s and early 1980s. Important academic research in this subject was led by Yale Patt and his HPSm simulator. A paper by James E. Smith and A.R. Pleszkun, published in 1985 completed the scheme by describing how the precise behavior of exceptions could be maintained in out-of-order machines.

Arguably the first machine to use out-of-order execution was probably the CDC 6600 (1964), which used a scoreboard to resolve conflicts. In modern usage, such scoreboarding is considered to be in-order execution, not out-of-order execution, since such machines stall on the first RAW (Read After Write) conflict. Strictly speaking, such machines initiate execution in-order, although they may complete execution out-of-order.

About three years later, the IBM 360/91 (1966) introduced Tomasulo's algorithm, supporting full out-of-order execution.

In 1990, IBM introduced the first out-of-order microprocessor, the POWER1, although out-of-order execution was limited to floating point instructions only.

Throughout the 1990s out-of-order execution became more common, and was featured in the IBM/Motorola PowerPC 601 (1993), Fujitsu/HAL SPARC64 (1995), Intel Pentium Pro (1995), MIPS R10000 (1996), HP PA-8000 (1996), AMD K5 (1996) and DEC Alpha 21264 (1998). Notable exceptions to this trend include the Sun UltraSPARC, HP/Intel Itanium, Transmeta Crusoe, Intel Atom, and the IBM POWER6.

The logical complexity of the out-of-order schemes was the reason that this technique did not reach mainstream machines until the mid-1990s. Many low-end processors meant for cost-sensitive markets still do not use this paradigm due to large silicon area that is required to build this class of machine. Low power usage is another design goal that's harder to achieve with an OoOE design.

Basic concept

In-order processors

In earlier processors, the processing of instructions is normally done in these steps:

1. Instruction fetch.
2. If input operands are available (in registers for instance), the instruction is dispatched to the appropriate functional unit. If one or more operand is unavailable during the current clock cycle (generally because they are being fetched from memory), the processor stalls until they are available.
3. The instruction is executed by the appropriate functional unit.
4. The functional unit writes the results back to the register file.

Out-of-order processors

This new paradigm breaks up the processing of instructions into these steps:

1. Instruction fetch.
2. Instruction dispatch to an instruction queue (also called instruction buffer or reservation stations).
3. The instruction waits in the queue until its input operands are available. The instruction is then allowed to leave the queue before earlier, older instructions.
4. The instruction is issued to the appropriate functional unit and executed by that unit.
5. The results are queued.
6. Only after all older instructions have their results written back to the register file, then this result is written back to the register file. This is called the graduation or retire stage.

The key concept of OoO processing is to allow the processor to avoid a class of stalls that occur when the data needed to perform an operation are unavailable. In the outline above, the OoO processor avoids the stall that occurs in step (2) of the in-order processor when the instruction is not completely ready to be processed due to missing data.

OoO processors fill these "slots" in time with other instructions that *are* ready, then re-order the results at the end to make it appear that the instructions were processed as normal. The way the instructions are ordered in the original computer code is known as *program order*, in the processor they are handled in *data order*, the order in which the data, operands, become available in the processor's registers. Fairly complex circuitry is

needed to convert from one ordering to the other and maintain a logical ordering of the output; the processor itself runs the instructions in seemingly random order.

The benefit of OoO processing grows as the instruction pipeline deepens and the speed difference between main memory (or cache memory) and the processor widens. On modern machines, the processor runs many times faster than the memory, so during the time an in-order processor spends waiting for data to arrive, it could have processed a large number of instructions.

Dispatch and issue decoupling allows out-of-order issue

One of the differences created by the new paradigm is the creation of queues which allows the dispatch step to be decoupled from the issue step and the graduation stage to be decoupled from the execute stage. An early name for the paradigm was *decoupled architecture*. In the earlier *in-order* processors, these stages operated in a fairly lock-step, pipelined fashion.

To avoid false operand dependencies, which would decrease the frequency when instructions could be issued out of order, a technique called register renaming is used. In this scheme, there are more physical registers than defined by the architecture. The physical registers are tagged so that multiple versions of the same architectural register can exist at the same time.

Execute and writeback decoupling allows program restart

The queue for results is necessary to resolve issues such as branch mispredictions and exceptions/traps. The results queue allows programs to be restarted after an exception, which requires the instructions to be completed in program order. The queue allows results to be discarded due to mispredictions on older branch instructions and exceptions taken on older instructions.

The ability to issue instructions past branches which have yet to resolve is known as speculative execution.

Micro-architectural choices

- Are the instructions dispatched to a centralized queue or to multiple distributed queues?

  IBM PowerPC processors use queues which are distributed among the different functional units while other Out-of-Order processors use a centralized queue. IBM uses the term *reservation stations* for their distributed queues.

- Is there an actual results queue or are the results written directly into a register file? For the latter, the queueing function is handled by register maps which hold the register renaming information for each instruction in flight.

Early Intel out-of-order processors use a results queue called a *re-order buffer*, while most later Out-of-Order processors use register maps.
More precisely: Intel P6 family microprocessors have both a ROB *re-order buffer* and a RAT register map mechanism. The ROB was motivated mainly by branch misprediction recovery.
The Intel P6 family was among the earliest OoO processors, was supplanted by the Intel Pentium 4 Willamette microarchitecture, but which returned after the right hand turn and, at the time of writing (2009) is still Intel's flagship microprocessor family.

Chapter-13

Jazelle and Delay Slot

Jazelle

Jazelle DBX (Direct Bytecode eXecution) allows some ARM processors to execute Java bytecode in hardware as a third execution state alongside the existing ARM and Thumb modes. Jazelle functionality was specified in the ARMv5TEJ architecture and the first processor with Jazelle technology was the **ARM926EJ-S**. Jazelle is denoted by a 'J' appended to the CPU name, except for post-v5 cores where it is required (albeit only in trivial form) for architecture conformance.

Jazelle RCT (Runtime Compilation Target) is a different technology and is based on ThumbEE mode and supports ahead-of-time (AOT) and just-in-time (JIT) compilation with Java and other execution environments.

The most prominent use of Jazelle DBX is by manufacturers of mobile phones to increase the execution speed of Java ME games and applications. A Jazelle-aware Java Virtual Machine (JVM) will attempt to run Java bytecodes in hardware, while returning to the software for more complicated, or lesser-used bytecode operations. ARM claims that approximately 95% of bytecode in typical program usage ends up being directly processed in the hardware.

The published specifications are very incomplete, being only sufficient for writing operating system code that can support a JVM that uses Jazelle. The declared intent is that only the JVM software needs to (or is allowed to) depend on the hardware interface details. This tight binding facilitates that the hardware and JVM can evolve together without affecting other software. In effect, this gives ARM Holdings considerable control over which JVMs are able to exploit Jazelle. It also prevents open source JVMs from using Jazelle. These issues do not apply to the ARMv7 ThumbEE environment, the nominal successor to Jazelle DBX.

Implementation

The Jazelle extension uses low-level binary translation, implemented as an extra stage between the fetch and decode stages in the processor instruction pipeline. Recognised bytecodes are converted into a string of one or more native ARM instructions.

The Jazelle mode moves JVM interpretation into hardware for the most common simple JVM instructions. This is intended to significantly reduce the cost of interpretation. Among other things, this reduces the need for JIT and other JVM accelerating techniques. JVM instructions that are not implemented in Jazelle hardware cause appropriate routines in the Jazelle-aware JVM implementation to be invoked. Details are not published, since all JVM innards are transparent (except for performance) if correctly interpreted.

Jazelle mode is entered via the BXJ instructions. A hardware implementation of Jazelle will only cover a subset of JVM bytecodes. For unhandled bytecodes—or if overridden by the operating system—the hardware will invoke the software JVM. The system is designed so that the software JVM does not need to know which bytecodes are implemented in hardware and a software fallback is provided by the software JVM for the full set of bytecodes.

Instruction set

The instruction set used in Jazelle mode is documented—it is Java bytecode after all. However, ARM have chosen to remain quiet on the exact execution environment details; the documentation provided with Sun's HotSpot Java Virtual Machine goes as far as to state: *For the avoidance of doubt, distribution of products containing software code to exercise the BXJ instruction and enable the use of the ARM Jazelle architecture extension without [..] agreement from ARM is expressly forbidden.*.

Employees of ARM have in the past published several white papers that do give some good pointers about the processor extension. Versions of the ARM Architecture Reference Manual available from 2008 have included pseudocode for the 'BXJ' (Branch and eXchange to Java) instruction, but with the finer details being shown as "SUB-ARCHITECTURE DEFINED" and documented elsewhere.

Application binary interface (ABI)

The Jazelle state relies on an agreed calling convention between the JVM and the Jazelle hardware state. This application binary interface is not published by ARM, rendering Jazelle an undocumented feature for most users and Free Software JVMs.

The entire VM state is held within normal ARM registers, allowing compatibility with existing operating systems and interrupt handlers unmodified. Restarting a bytecode (such as following a return from interrupt) will re-execute the complete sequence of related ARM instructions.

Specific registers are designated to hold the most important parts the JVM state, registers r0-r3 hold an alias of the top of the Java stack, r4 holds Java local operand zero (pointer to `*this`) and r6 contains the Java stack pointer.

Jazelle reuses the existing Program Counter register r15. A pointer to the *next* bytecode goes in r14, so the use of the PC is not generally user-visible except during debugging.

CPSR: Mode indication

Java bytecode is indicated as the current instruction set by a combination of two-bits in the ARM CPSR (Current Program Status Register). The 'T'-bit must be cleared and the 'J'-bit set.

Bytecodes are decoded by the hardware in two stages (versus a single stage for Thumb and ARM code) and switching between hardware and software decoding (Jazelle mode and ARM mode) takes ~4 clock cycles.

For entry to Jazelle hardware state to succeed, the JE (Jazelle Enable) bit in the CP14:c0(c2)[bit 0] register must be set; clearing of the JE bit by a [privileged] operating-system provides a high-level override to prevent application programs from using the hardware Jazelle acceleration, additionally the CV (Configuration Valid) bit found in CP14:c0(c1)[bit 1] must be set to show that there is a consistent Jazelle state setup for the hardware to use.

BXJ: Branch to Java

The BXJ instruction attempts to switch to Jazelle state, and if allowed and successful, sets the 'J' bit in the CPSR; otherwise "falling through" and acting as a standard BX (Branch) instruction. The only time when an operating system, or debugger must be fully aware of the Jazelle mode is when decoding a faulted or trapped instruction. The Java program counter (PC) pointing to the next instructions must be placed in the Link Register (r14) before executing the BXJ branch request, as regardless of hardware or software processing, the system must know where to begin decoding.

Because the current state is held in the CPSR, the bytecode instruction set is automatically reselected after task-switching and processing of the current Java bytecode is restarted.

Following an entry into the Jazelle state mode, bytecodes can be processed in one of three ways; decoded and executed natively in hardware, handled in software (with optimised ARM/ThumbEE JVM code), or treated as an invalid/illegal opcode. The third case will cause a branch to an ARM exception mode, as will a Java bytecode of 0xff, which is used for setting JVM breakpoints.

Execution will continue in hardware until an unhandled bytecode is encountered, or an exception occurs. Between 134 and 149 bytecodes (out of 203 bytecodes specified in the JVM specification) are translated and executed directly in the hardware.

Low-level registers

Low-level configuration registers, for the hardware virtual machine, are held in the ARM Co-processor "CP14 register c0". The registers allow detecting, enabling or disabling the hardware accelerator—if it is available.

- The Jazelle Identity Register in register CP14:c0(c0) is read-only accessible in all modes.
- The Jazelle OS Control Register at CP14:c0(c1) is only accessible in kernel mode and will cause an exception when accessed in user-mode.
- The Jazelle Main Configuration Register at CP14:c0(c2) is write-only in user-mode and read-write in kernel mode.

A "trivial" hardware implementation of Jazelle (as found in the QEMU emulator) is only required to support the BXJ opcode itself (treating BXJ as a normal BX instruction) and to return RAZ (Read-As-Zero) for all of the CP14:c0 Jazelle-related registers.

Successor: ThumbEE

The ARMv7 architecture has de-emphasized Jazelle and *Direct Bytecode Execution* of JVM bytecodes. In implementation terms, only trivial hardware support for Jazelle is now required: support for entering and exiting Jazelle mode, but not for executing any Java bytecodes.

Instead, the *Thumb Execution Environment* (ThumbEE) is now preferred. Support for this is mandatory in ARMv7-A processors (such as the Cortex-A8 and Cortex-A9), and optional in ARMv7-R processors. ThumbEE targets compiled environments, perhaps using JIT technologies. It is not at all specific to Java, and is fully documented; much broader adoption is anticipated than Jazelle was able to achieve.

ThumbEE is a variant of the Thumb2 16/32-bit instruction set. It integrates null pointer checking; defines some new fault mechanisms; and repurposes the 16-bit LDM and STM opcode space to support a few instructions such as range checking, a new handler invocation scheme, and more. Accordingly, compilers that produce Thumb or Thumb2 code can be modified to work with ThumbEE-based runtime environments.

Delay slot

In computer architecture, a **delay slot** is an instruction slot that gets executed without the effects of a preceding instruction. The most common form is a single arbitrary instruction located immediately after a branch instruction on a RISC or DSP architecture; this instruction will execute even if the preceding branch is taken. Thus, by design, the instructions appear to execute in an illogical or incorrect order. It is typical for assemblers to automatically reorder instructions by default, hiding the awkwardness from assembly developers and compilers.

Branch delay slots

When a branch instruction is involved, the location of the following delay slot instruction in the pipeline may be called a **branch delay slot**. Branch delay slots are found mainly in DSP architectures and older RISC architectures. MIPS, PA-RISC, ETRAX CRIS, SuperH, and SPARC are RISC architectures that each have a single branch delay slot; PowerPC, ARM, and the more recently designed Alpha do not have any. DSP architectures that each have a single branch delay slot include the µPD77230 and TMS320C3x. The SHARC DSP and MIPS-X use a double branch delay slot; such a processor will execute a pair of instructions following a branch instruction before the branch takes effect.

The following example shows delayed branches in assembly language for the SHARC DSP. Registers R0 through R9 are cleared to zero in order by number (the register cleared after R6 is R7, not R9). No instruction executes more than once.

```
     R0 = 0;
     CALL fn (DB);      /* call a function, below at label "fn" */
     R1 = 0;            /* first delay slot */
     R2 = 0;            /* second delay slot */
     /***** discontinuity here (the CALL takes effect) *****/

     R6 = 0;            /* the CALL/RTS comes back here, not at "R1 =
0" */
     JUMP end (DB);
     R7 = 0;            /* first delay slot */
     R8 = 0;            /* second delay slot */
     /***** discontinuity here (the JUMP takes effect) *****/

     /* next 4 instructions are called from above, as function "fn" */
fn:  R3 = 0;
     RTS (DB);          /* return to caller, past the caller's delay
slots */
     R4 = 0;            /* first delay slot */
     R5 = 0;            /* second delay slot */
     /***** discontinuity here (the RTS takes effect) *****/

end: R9 = 0;
```

The goal of a pipelined architecture is to complete an instruction every clock cycle. To maintain this rate, the pipeline must be full of instructions at all times. The branch delay slot is a side effect of pipelined architectures due to the branch hazard, i.e. the fact that the branch would not be resolved until the instruction has worked its way through the pipeline. A simple design would insert stalls into the pipeline after a branch instruction until the new branch target address is computed and loaded into the program counter. Each cycle where a stall is inserted is considered one branch delay slot. A more sophisticated design would execute program instructions which are not dependent on the result of the branch instruction. This optimization can be performed in software at compile time by moving instructions into branch delay slots in the in-memory instruction stream, if the hardware supports this. Another side effect is that special handling should be taken care of managing breakpoint on instructions as well as stepping while debugging within branch delay slot.

The ideal number of branch delay slots in a particular pipeline implementation is dictated by the number of pipeline stages, the presence of register forwarding, what stage of the pipeline the branch conditions are computed, whether or not a branch target buffer (BTB) is used and many other factors. Software compatibility requirements dictate that an architecture may not change the number of delay slots from one generation to the next. This inevitably requires that newer hardware implementations contain extra hardware to ensure that the architectural behavior is followed despite no longer being relevant.

Load delay slot

A load delay slot is an instruction which executes immediately after a load (of a register from memory) but does not see the result of the load. Load delay slots are very uncommon because load delays are highly unpredictable on modern hardware. A load may be satisfied from RAM or from a cache, and may be slowed by resource contention. Load delays were seen on very early RISC processor designs. The MIPS I ISA (implemented in the R2000 and R3000 microprocessors) suffers from this problem.

The following example is MIPS I assembly code, showing both a load delay slot and a branch delay slot.

```
lw   v0,4(v1)    # load word from address v1+4 into v0
nop              # useless load delay slot
jr   v0          # jump to the address specified by v0
nop              # useless branch delay slot
```