Instruction set architecture

An **instruction set architecture** (**ISA**) is an abstract model of a computer. It is also referred to as **architecture** or **computer architecture**. A realization of an ISA is called an *implementation*. An ISA permits multiple implementations that may vary in performance, physical size, and monetary cost (among other things); because the ISA serves as the interface between software and hardware. Software that has been written for an ISA can run on different implementations of the same ISA. This has enabled binary compatibility between different generations of computers to be easily achieved, and the development of computer families. Both of these developments have helped to lower the cost of computers and to increase their applicability. For these reasons, the ISA is one of the most important abstractions in computing today.

An ISA defines everything a machine language programmer needs to know in order to program a computer. What an ISA defines differs between ISAs; in general, ISAs define the supported data types, what state there is (such as the main memory and registers) and their semantics (such as the memory consistency and addressing modes), the *instruction set* (the set of machine instructions that comprises a computer's machine language), and the input/output model.

Contents

Overview

Classification of ISAs

Machine language

- Instruction types
  - Data handling and memory operations
  - Arithmetic and logic operations
  - Control flow operations
  - Coprocessor instructions
- Complex instructions
- Parts of an instruction
- Instruction length
- Representation
- Design

Instruction set implementation

- Code density
- Number of operands
- Register pressure

See also

References

Further reading

External links

Overview

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

The concept of an *architecture*, distinct from the design of a specific machine, was developed by Fred Brooks at IBM during the design phase of System/360.

> Prior to NPL [System/360], the company's computer designers had been free to honor cost objectives not only by selecting technologies but also by fashioning functional and architectural refinements. The SPREAD compatibility objective, in contrast, postulated a single architecture for a series of five processors spanning a wide range of cost and performance. None of the five engineering design teams could count on being able to bring about adjustments in architectural specifications as a way of easing difficulties in achieving cost and performance objectives.[1]:p.137

Some virtual machines that support bytecode as their ISA such as Smalltalk, the Java virtual machine, and Microsoft's Common Language Runtime, implement this by translating the bytecode for commonly used code paths into native machine code. In addition, these virtual machines execute less frequently used code paths by interpretation (see: Just-in-time compilation). Transmeta implemented the x86 instruction set atop VLIW processors in this fashion.

Classification of ISAs

An ISA may be classified in a number of different ways. A common classification is by architectural *complexity*. A complex instruction set computer (CISC) has many specialized instructions, some of which may only be rarely used in practical programs. A reduced instruction set computer (RISC) simplifies the processor by efficiently implementing only the instructions that are frequently used in programs, while the less common operations are implemented as subroutines, having their resulting additional processor execution time offset by infrequent use.[2]

Other types include very long instruction word (VLIW) architectures, and the closely related *long instruction word* (LIW) and *explicitly parallel instruction computing* (EPIC) architectures. These architectures seek to exploit instruction-level parallelism with less hardware than RISC and CISC by making the compiler responsible for instruction issue and scheduling.

Architectures with even less complexity have been studied, such as the minimal instruction set computer (MISC) and one instruction set computer (OISC). These are theoretically important types, but have not been commercialized.

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 are executed sequentially, or as otherwise directed by control flow instructions.

Instruction types

Examples of operations common to many instruction sets include:

Data handling and memory operations

- *Set* a register to a fixed constant value.
- *Copy* data from a memory location to a register, or vice versa (a machine instruction is often called *move*; however, the term is misleading). Used to store the contents of a register, result of a computation, or to retrieve stored data to perform a computation on it later. Often called load and store operations.
- *Read* and *write* data from hardware devices.

Arithmetic and logic operations

- *Add*, *subtract*, *multiply*, or *divide* the values of two registers, placing the result in a register, possibly setting one or more condition codes in a status register.
  - *increment*, *decrement* in some ISAs, saving operand fetch in trivial cases.
- Perform bitwise operations, e.g., taking the *conjunction* and *disjunction* of corresponding bits in a pair of registers, taking 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).
- *Floating-point instructions* for arithmetic on floating-point numbers.

Control flow operations

- *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.
- *Call* another block of code, while saving the location of the next instruction as a point to return to.

Coprocessor instructions

- Load/store data to and from a coprocessor, or exchanging with CPU registers.
- Perform coprocessor operations.

Complex instructions

Processors may 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:

- transferring multiple registers to or from memory (especially the stack) at once
- moving large blocks of memory (e.g. string copy or DMA transfer)
- complicated integer and floating-point arithmetic (e.g. square root, or transcendental functions such as logarithm, sine, cosine, etc.)
- *SIMD instructions*, a single instruction performing an operation on many homogeneous values in parallel, possibly in dedicated SIMD registers
- performing an atomic test-and-set instruction or other read-modify-write atomic instruction
- instructions that perform ALU operations with an operand from memory rather than a register

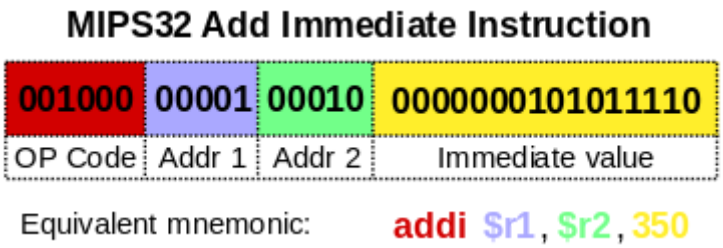
Complex instructions are more common in CISC instruction sets than in RISC instruction sets, but RISC instruction sets may include them as well. RISC instruction sets generally do not include ALU operations with memory operands, or instructions to move large blocks of memory, but most RISC instruction sets include SIMD or vector instructions that perform the same arithmetic operation on multiple pieces of data at the same time. SIMD instructions 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

On traditional architectures, an instruction includes an opcode that specifies the operation to perform, 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), only operand(s).

The Forth virtual machine and other "0-operand" instruction sets lack any operand specifier fields, such as some stack machines including NOSC.[3]



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

Conditional instructions often have a predicate field—a few bits that encode the specific condition to cause the operation to be performed rather than not performed. For example, a conditional branch instruction will be executed, and the branch taken, if the condition is true, so that execution proceeds to a different part of the program, and not executed, and the branch not taken, if the condition is false, so that execution continues sequentially. Some instruction sets also have conditional moves, so that the move will be executed, and the data stored in the target location, if the condition is true, and not executed, and the target location not modified, if the condition is false. Similarly, IBM z/Architecture has a conditional store instruction. A few instruction sets include a predicate field in every instruction; this is called branch predication.

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. The longest possible instruction on x86 is 15 bytes (120 bits).[4] Within an instruction set, different 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. Some, such as the ARM with *Thumb-extension* have *mixed* variable encoding, that is two fixed, usually 32-bit and 16-bit encodings, where instructions can not be mixed freely but must be switched between on a branch (or exception boundary in ARMv8).

A RISC instruction set normally has a fixed instruction length (often 4 bytes = 32 bits), whereas a typical CISC instruction set may have instructions of widely varying length (1 to 15 bytes for x86). Fixed-length instructions are less complicated to handle than variable-length instructions for several reasons (not having to check whether an instruction straddles a cache line or virtual memory page boundary[5] for instance), and are therefore somewhat easier to optimize for speed.

Representation

The instructions constituting a program are rarely specified using their internal, numeric form (machine code); they may be specified by programmers using an assembly language or, more commonly, may be generated from programming languages 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 that 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 and cache efficiency, or simplify programming.

Some instruction set designers reserve one or more opcodes for some kind of system call or 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[6] 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 such as "fetch-and-add", "load-link/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 a particular instruction set provide the same programming model, and all implementations of that instruction set are able to run the same 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. Some computer designs "hardwire" the complete instruction set decoding and sequencing (just like the rest of the microarchitecture).
2. Other designs employ microcode routines or tables (or both) to do this—typically as on-chip ROMs or PLAs or both (although separate RAMs and ROMs have been used historically). The Western Digital MCP-1600 is an older example, using a dedicated, separate ROM for microcode.

Some designs use a combination of hardwired design and microcode for the control unit.

Some CPU designs use a writable control store—they compile the instruction set to a writable RAM or flash inside the CPU (such as the Rekursiv processor and the Imsys Cjip),[7] or an FPGA (reconfigurable computing).

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, to perform digital filters fast enough, the MAC instruction in a typical digital signal processor (DSP) must use 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, 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 often have 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. They sacrifice code density to simplify implementation circuitry, and try to increase performance via higher clock frequencies and more registers. A single RISC instruction typically performs only a single operation, such as an "add" of registers or a "load" from a memory location into a register. A RISC instruction set normally has a fixed instruction length, whereas a typical CISC instruction set has instructions of widely varying length. 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.

Certain embedded RISC ISAs like Thumb and AVR32 typically exhibit very high density owing to a technique called code compression. This technique packs two 16-bit instructions into one 32-bit instruction, which is then unpacked at the decode stage and executed as two instructions.[8]

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. The code density of MISC is similar to the code density of 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.)

```
C = A+B
```

- 0-operand (*zero-address machines*), so called stack machines: All arithmetic operations take place using the top one or two positions on the stack: `push a`, `push b`, *add*, `pop c`.
  - `C = A+B` needs *four instructions*. For stack machines, the terms "0-operand" and "zero-address" apply to arithmetic instructions, but not to all instructions, as 1-operand push and pop instructions are used to access memory.
- 1-operand (*one-address machines*), so called accumulator machines, include early computers and many small microcontrollers: most instructions specify a single right operand (that is, constant, a register, or a memory location), with the implicit accumulator as the left operand (and the destination if there is one): `load a`, `add b`, `store c`.
  - `C = A+B` needs *three instructions*.
- 2-operand — many CISC and RISC machines fall under this category:
  - CISC — `move A` to *C*; then `add B` to *C*.
    - `C = A+B` needs *two instructions*. This effectively 'stores' the result without an explicit *store* instruction.
  - CISC — Often machines are limited to one memory operand per instruction: `load a,reg1`; `add b,reg1`; `store reg1,c`; This requires a load/store pair for any memory movement regardless of whether the *add* result is an augmentation stored to a different place, as in `C = A+B`, or the same memory location: `A = A+B`.
    - `C = A+B` needs *three instructions*.
  - RISC — Requiring explicit memory loads, the instructions would be: `load a,reg1`; `load b,reg2`; `add reg1,reg2`; `store reg2,c`.
    - `C = A+B` needs *four instructions*.

- 3-operand, allowing better reuse of data:[5]
  - CISC — It becomes either a single instruction: `add a,b,c`
    - `C = A+B` needs *one instruction*.
  - CISC — Or, on machines limited to two memory operands per instruction, `move a,reg1; add reg1,b,c;`
    - `C = A+B` needs *two instructions*.
  - RISC — 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;`
    - `C = A+B` needs *four instructions*.
    - Unlike 2-operand or 1-operand, this leaves all three values a, b, and c in registers available for further reuse.[5]
- 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.

Due to the large number of bits needed to encode the three registers of a 3-operand instruction, RISC architectures that have 16-bit instructions are invariably 2-operand designs, such as the Atmel AVR, TI MSP430, and some versions of ARM Thumb. RISC architectures that have 32-bit instructions are usually 3-operand designs, such as the ARM, AVR32, MIPS, Power ISA, and SPARC architectures.

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. If some of the operands are given implicitly, fewer operands need be specified in the instruction. When a "destination operand" explicitly specifies the destination, an additional operand must be supplied. Consequently, the number of operands encoded in an instruction may differ from the mathematically necessary number of arguments for a logical or arithmetic operation (the arity). Operands are either encoded in the "opcode" representation of the instruction, or else are given as values or addresses following the instruction.

Register pressure

Register pressure measures the availability of free registers at any point in time during the program execution. Register pressure is high when a large number of the available registers are in use; thus, the higher the register pressure, the more often the register contents must be spilled into memory. Increasing the number of registers in an architecture decreases register pressure but increases the cost.[9]

While embedded instruction sets such as Thumb suffer from extremely high register pressure because they have small register sets, general-purpose RISC ISAs like MIPS and Alpha enjoy low register pressure. CISC ISAs like x86-64 offer low register pressure despite having smaller register sets. This is due to the many addressing modes and optimizations (such as sub-register addressing, memory operands in ALU instructions, absolute addressing, PC-relative addressing, and register-to-register spills) that CISC ISAs offer.[10]

See also

- Comparison of instruction set architectures
- Computer architecture
- CPU design
- Emulator
- Simulator
- List of instruction sets
- Instruction set simulator
- OVPsim full systems simulator providing ability to create/model/emulate any instruction set using C and standard APIs
- Register transfer language (RTL)
- Micro-operation

References

1. Pugh, Emerson W.; Johnson, Lyle R.; Palmer, John H. (1991). *IBM's 360 and Early 370 Systems* (https://www.amazon.com/IBMs-Early-Systems-History-Computing/dp/0262161230). MIT Press. ISBN 0-262-16123-0.
2. Crystal Chen; Greg Novick; Kirk Shimano (December 16, 2006). "RISC Architecture: RISC vs. CISC" (http://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/). *cs.stanford.edu*. Retrieved February 21, 2015.
3. "Forth Resources: NOSC Mail List Archive" (http://strangegizmo.com/forth/NOSC/). *strangegizmo.com*. Retrieved 2014-07-25.
4. "Intel® 64 and IA-32 Architectures Software Developer's Manual" (http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html). Intel Corporation. Retrieved 12 July 2012.
5. The evolution of RISC technology at IBM by John Cocke (http://domino.watson.ibm.com/tchjr/journalindex.nsf/0/22d06c5aa961e78085256bfa0067fa93?OpenDocument) – IBM Journal of R&D, Volume 44, Numbers 1/2, p.48 (2000)
6. Ganssle, Jack. "Proactive Debugging" (http://embedded.com/showArticle.jhtml?articleID=9900044) Published February 26, 2001.
7. "Great Microprocessors of the Past and Present (V 13.4.0)" (http://cpushack.net/CPU/cpu7.html). *cpushack.net*. Retrieved 2014-07-25.
8. Weaver, Vincent M.; McKee, Sally A. (2009). *Code density concerns for new architectures* (http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5413117&tag=1). IEEE International Conference on Computer Design.
9. Page, Daniel (2009). "11. Compilers". *A Practical Introduction to Computer Architecture*. Springer. p. 464. ISBN 978-1-84882-255-9.
10. Venkat, Ashish; Tullsen, Dean M. (2014). *Harnessing ISA Diversity: Design of a Heterogeneous-ISA Chip Multiprocessor* (http://dl.acm.org/citation.cfm?id=2665692). 41st Annual International Symposium on Computer Architecture.

Further reading

- Bowen, Jonathan P. (July–August 1985). "Standard Microprocessor Programming Cards". **9** (6): 274–290. doi:10.1016/0141-9331(85)90116-4.

External links

- Programming Textfiles: Bowen's Instruction Summary Cards
- Mark Smotherman's Historical Computer Designs Page
