Instruction Set Design

When we design a new instruction set, we are creating the encodings for all of the instructions that a machine will be able to execute. The process is as much art as it is engineering. Architects judge instruction set designs qualitatively by looking at how symmetric and regular they are. A symmetric instruction set avoids special conditions and cases that cause operations to operate differently on different operands. For example, in early microprocessors it was common to see registers that could be loaded from memory but not stored, and which had to be moved, using special instructions, to other registers that could be stored. In a symmetric instruction set, a register that can be stored can also be loaded, and if a value can be moved from one register to another, movement in the opposite direction is supported. In a regular instruction set, operations can be applied uniformly to different operands. For example, addition is available for all lengths of integers; loads and stores of all types can refer to all data types; and so on.

An instruction set's encoding is also a matter of art and engineering. Placing related instructions into a class or type together simplifies the decoding process -- the logic that determines how the instruction is executed. For example, dividing instructions into integer execute, floating point execute, load/store, and branch groups enables straightforward decoding (a two-bit type field becomes an index that determines the group to which the instruction belongs) and dispatch (the instructions for a group are all sent to the same execution unit in the processor). It also makes life easier on the compiler writer or assembly language programmer (enabling simple, regular rules for instruction generation).

Some architectures begin with clean, straightforward instruction sets but evolve to include numerous special cases in their instruction sets. Other architectures, start out with irregular and asymmetric instruction sets due to severe constraints that were placed on the designers. It is rare to see an established architecture with a clean instruction set architecture (ISA).

Quantitatively, we judge an instruction set by its effect on performance. Measuring ISA impact on performance is nontrivial. We would like to be able to measure the execution time for a piece of code, with and without an ISA feature. But that requires changing the ISA (usually in a software simulation) and the compiler (to take advantage of the feature). Making a simulation that accurately models all of the performance-affecting aspects of a modern processor is very difficult (even the manufacturers have trouble doing this), and changing a production-quality optimizing compiler is also challenging.

In the past it was common to see hand-coded routines that used a special feature, and beyond calling those library routines, the compiler could not generate code to make use of it. Today, manufacturers are more cautious about adding new instructions or features because each addition requires so much additional software support that it faces an uphill battle for acceptance.

There are fundamental parameters that must be chosen as the basis for an ISA. These include the width of the operands, the data types of the operands, the number of operands that are referenced in an instruction, the address range of the system, the memory addressing modes used by instructions, the number and types of operations that are supported, and the number of registers of each type.

Operand widths are typically a power of two number of bits. Common widths are 8, 16, 32, and 64 bits. There have also been architectures with 12, 18, 24, 30, 36, 48, 60, and 72 bit lengths -- mostly because of early 6-bit character codes.

Data types traditionally include integers for all the different widths, both in twos complement signed form and in unsigned form, characters (8 and 16 bit), and floating point numbers (today based on the IEEE 754 standard, and either 32 or 64 bits in size, though the Intel architecture has 80-bit floating point). Some architectures also support strings, pixels, binary coded decimal numbers, and vectors (arrays of floating-point numbers).

The number of operands in an instruction usually is in the range of 0 to 3. We refer to these as "addresses" in the description of the instruction set because historically the operands were in memory. Today, however, it is more common for these addresses to be indexes into a set of registers. A three-address architecture has two source operands and a destination operand. For example an addition instruction that computes A=B+C. A two-address architecture has one source operand, and a second operand that is both a source and a destination. Its addition instruction would compute A = A+B. A one-address architecture has a single, dedicated arithmetic register that is always a source operand and the destination. The register is called the accumulator (Acc), and the corresponding addition would perform Acc = Acc + A. This instruction would simply be written as "Add A" with the accumulator implicit as the other two operands. A zero-address architecture uses a stack to perform its arithmetic. Assuming that two operands have been pushed onto the stack, the "Add" instruction (no operands), pops the top two elements off the stack, computes their sum, and pushes the result back onto the stack. Three-address designs are the most common today -- most of the others were of interest in times when the technology made it expensive to have more than a few registers.

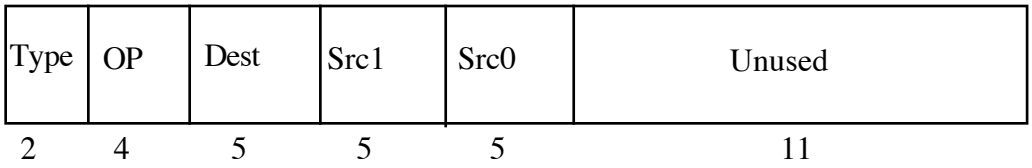
The address range determines the maximum size of the memory that the architecture can use. In desktop and server architectures it is usually far beyond the amount that would be physically employed, but instead provides the operating system with flexibility in mapping virtual memory to different processes. In a simple architecture that lacks virtual memory, such as an embedded processor, the address space may be much smaller. An address range is usually specified by the number of bits. For example, a 32-bit architecture may support a 32-bit (4 GB) address space. An embedded processor with a 16-bit word might only have 12-bits (4 KB) of address space.

Many different addressing modes are possible, although most of the more complex types are a response to limited numbers of registers or a small instruction word. See the example architectures below. Common modes include:

Register direct: An index field in the instruction specifies a register number
Register indirect: A field specifies a register that holds a memory address
Immediate: A field holds a value to be used as an operand.

The number of registers of a given type has a significant impact on the design of an instruction set, assuming they are addressed directly by it. More registers provide a greater amount of fast memory locations to hold operands, and reduce the need for slower accesses to memory. However, registers consume more power and space on the chip, and having more of them takes up more bits within the instruction (making it harder to find an encoding).

For example, with an execute instruction group that supports two sources and a destination among a file of 32 registers, 15 bits of the instruction code must be dedicated to the register references. In a 32-bit instruction, this leaves 17 bits to specify the operation and the instruction type. If there are four types (or classes) of instructions (e.g., integer execute, floating point execute, load/store, branch) then 15 bits remain for the operation. If only 16 operations are supported, then there are still 11 bits available for other uses.



Having extra bits makes it tempting to find a use for them. Let's see what happens if we decide that some of the operations are going to work on an immediate operand in those extra bits. First of all, we can grab the 5 bits from Src0 and append them to the immediate operand field because we need only one source register when our second operand is immediate. This gives us 16 bits for immediate values, which is very nice as it can represent a short, two bytes, a Unicode character, and so on. However, we need some way to specify that this instruction has an immediate value and not a second source register. We need the Type, Dest and Src1 fields, so only the OP field remains for representing this status. With just 16 operations, it would be difficult to dedicate a bit to indicate immediate or register operand – we would be left with just 8 distinct operations. Alternatively, we could just have a few special operations that work on immediate values, say op codes 0 – 3, and the rest refer to registers. However, this makes the instruction set irregular. Another option is to give up the idea of having 16-bit operands, and take a bit from the unused section that specifies immediate or register as Src0. All of these can work, and each one presents a set of tradeoffs. It's up to the architects to choose the direction to follow.

As you can see, instruction set design is largely an exercise in balancing tradeoffs and attempting to estimate the impact. Once an architecture has been built, it is easier to

evaluate the effect of a minor change. But initially it takes a holistic awareness of the system and its intended applications to make decisions that seek the necessary balance.

The Instruction Set Architecture Examples

As you are aware from assembly language programming, machines are programmed with binary codes that represent instructions and data. The fact that the program and data can both reside in the same memory is called the "stored program concept" and is what makes computers so flexible.

Instructions are usually divided into fields that represent different information required to carry out an operation (such as the op-code, data addresses, register operands, etc.). If you have had experience with the Intel processors you have worked with a complex instruction set. Let's briefly review the features of this instruction set architecture:

Intel Instruction Set Architecture

Eight "General Purpose" registers: EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP

The first four can be treated as 32 or 16 bit registers, with the lower 16 bits being further divided into bytes that are designated separately and which provide limited compatibility with the 8-bit Intel architectures.

Six Segment registers: CS, SS, DS, ES, FS, GS

CS being Code Segment, SS being Stack Segment, DS being Data Segment (along with the other three)

Instruction Pointer (16 or 32 bits), Flags Register (32 bits, with some undefined, although each new generation seems to define more of them)

Memory Management Registers:

Global Descriptor Table Register (48 bits)

Interrupt Table Descriptor Register (48 bits)

Task Register (16 selector + 32 base address+32 limit +8 attributes)

Local descriptor table register (same as Task Register)

Control Registers:

CR0 - CR4, each witha different set of control bits.

Floating Point Registers:

R0 - R7, each 80 bits plus a 2-bit tag aggregated into the Tag Word.

Control Register (16 bits), Status Register (16 bits), Tag Word (16 bits), Instruction Pointer (48 bits), Data Pointer (48 bits)

Debug Registers

D0 - D8 , D0 - D4 hold breakpoints while the other four are for control and status.

The Intel ISA has 12 addressing modes:

Register direct
Immediate
Direct memory
Base
Base + Displacement
Index + Displacement
Scaled Index + Displacement
Based Index
Based Scaled Index
Based Index + Displacement
Based Scaled Index + Displacement
Relative

Operands in the Intel processors can be 8, 16, 32, 48, 64, or 80 bits long. The instruction set also supports string operations over the entire address range.

Instructions in the Intel design can be as small as one byte or as long as 12 bytes and any combination in between. The first bytes generally contain opcode, mode specifiers, and register fields, while the remainder are for address displacement and immediate data.

Now for the sake of comparison, let's take a look at the Instruction Set Architecture of the MIPS family of RISC processors.

MIPS R4000

General Purpose registers: R0 - R31, 64 bits. R0 is constant 0, R31 is a link register for link and jump instructions.

Multiply and divide High/Low registers -- 64 bits each.

Program Counter (64 bits)

Floating Point registers: FPR0 - FPR31 (32 or 64 bit)

Control/Status register (32 bit)

Implementation/Revision register (32 bit)

Control Coprocessor registers: 32 named registers, each 32 bits. Cache and MMU control, exception processing, debugging.

The MIPS family has four addressing modes:

Base + immediate offset (loads and stores)
Register direct (arithmetic)
Immediate (jumps)
PC relative (branches)

Memory accesses in the MIPS architecture are to any multiple between 1 and 8 bytes.

There are three instruction formats, all of which are 32 bits in length.

Comparison

Even at this cursory level of review, it's clear how much simpler the Instruction Set Architecture of the MIPS processor is. The same is true of most RISC processors.

What are the advantages of making the ISA simple?

Easier to program.

Easier to build.

Requires less hardware for basic functionality, leaving room for circuitry to increase performance.

Let's take a closer look at each of these reasons in turn.

A RISC ISA is easier to program in the sense that we have fewer alternatives to choose between -- there are only a few ways to accomplish a given operation that make sense. In a CISC ISA, we have many alternatives to choose from. While we, as assembly language programmers, may be comfortable with making those choices, most compilers don't have the necessary sophistication to make the subtle distinctions necessary. In fact, one of the studies that resulted in the RISC approach was an analysis of object code on CISC architectures that showed only a tiny fraction of the available instructions were being used. The compilers were generating simple, straightforward code and did not have the sophistication to make use of the many special operations in the CISC ISA.

However, a RISC ISA is also more difficult to program in that it often requires a greater number of instructions than the CISC ISA to accomplish the same task. It is not unusual for a RISC object file to be 50% larger than a CISC object for the same program.

The RISC architecture is easier to build in that it is more regular, has a simpler set of interconnections, fewer special cases, and generally takes less circuitry to implement. However, it is also more difficult to build in that it must run faster than a CISC architecture in order to execute the greater number of instructions in the same time.

Faster circuits are harder to design and consume more power. But at least the RISC design compensates for this difficulty by eliminating other sources of complexity. The RISC processor also requires a larger cache memory. Why?

Because its object code is larger, it has to hold more instructions in cache.

RISC does require less hardware for the basic functionality, which (if we ignore the need for a larger cache) leaves room for more acceleration circuits to be added. For example, the complex pipeline of the Intel P6 was only just catching up to where RISC processors had been for a full generation. RISC processors also have a greater degree of instruction level parallelism that CISC processors.

The result of the streamlined RISC design is that a RISC processor is typically 1.5 to 5 times faster than a CISC processor of the same period. It is actually possible to emulate a CISC processor in software on some RISC processors with nearly the same performance as the actual CISC processor.

Why CISC?

The natural question at this point is, "Why did CISC develop, if it is lower in performance?"

Architects at the time were using less-developed memory technology. Magnetic core was expensive, and the amount of it in a machine was typically quite small (64K words was considered large). It was also slower than the circuitry of the CPU.

Given that you have very little memory and it is slow, what would you do to the instruction set to compensate?

Make each instruction do more work.

This solution saves memory (fewer instructions are needed to accomplish the same task) and it saves time (instructions don't have to be fetched as often to keep the CPU busy).

Thus, architects of the time set about looking for ways to combine simpler instructions into more complex ones. In addition, they had pressure from assembly language programmers (who were proportionally as numerous then as C programmers are today) to make their jobs easier by providing more powerful instructions. The result was higher performance.

Then the technology shifted: memory became cheap (allowing it to be large) and fast, and compilers started generating efficient code (which put a lot of assembly language programmers out of work). Since the compilers only used a subset of the instructions, the complexity became excess baggage.

What Distinguishes RISC from CISC?

Many people have wrestled with this question, and their answers are usually only partly satisfying. Some say it is the orientation toward a large bank of general purpose registers. But others point to the Motorola 68000, which has a bank of general purpose registers and note that it is clearly CISC. Others point to the smaller number of addressing modes, but there are counterexamples -- some RISC processors actually have nearly as many modes as CISC processors. The number of instructions is often cited, but again there are counterexamples. The same is true for the fixed size of a RISC instruction (one word) vs. the variable size of a CISC instruction, and the relative clock rates.

In the end, we may conclude from these commentators that RISC is a philosophy that attempts to restrict a design to simple, regular, general purpose features as much as possible, but is hard to pin down to specifics. The truly cynical might conclude that it is just a convenient label for marketing purposes. However, one aspect that truly seems to set RISC apart is that it keeps independent operations separate. In particular, it separates computation from memory access.

Why is this important? Keep in mind that a cache doesn't always hold the required portion of main memory.

Given that memory accesses may be slow, if they are independent of computations, a clever compiler can rearrange the operations so that memory accesses are done while the CPU is busy with other work. If the memory operation is tightly bound to the computation (as in a CISC ISA), then there is less flexibility for this reordering and the CPU spends more time waiting.

In a complex implementation of the ISA, with multiple arithmetic units, separating operations from each other frees the compiler to rearrange them to better utilize the multiple units. For example, if a program is operating on an integer array, the address arithmetic and the arithmetic on the array values may compete for the same units. If the address arithmetic can be split into separate operations, however, the compiler may be able to rearrange them with respect to the array value operations so that the units are fully utilized. If that address arithmetic is instead done by a single instruction that locks up the arithmetic units, it may be necessary for the computations on the array values to wait until the units are unlocked, even though some of them may be idle for part of the address instruction.

The Importance of Compiler Technology to RISC

RISC ISAs are simple to program in a naive, unoptimized manner. The trouble is that the sophisticated acceleration mechanisms we've touched on are not part of the ISA. Even though they are hidden, they affect the performance of the system in ways that are dependent on the sequence of instructions executed.

For example, if a 2-dimensional array is laid out with its row values appearing sequentially in memory, and a programmer can either access the data by reading along the rows or along the columns, what order should she or he choose? From the ISA, we would say it doesn't matter. But knowing that there is a cache that always loads

sequences of words from memory, we can expect much higher performance if we write the code to access the array along its rows. That way, we get to spread the penalty for going to main memory across a series of accesses that find the subsequent data already in the cache.

With more complicated acceleration mechanisms, such as having multiple arithmetic units, it becomes difficult for an assembly language programmer to keep all of the details straight. Essentially, there is a large space of potential arrangements of instructions, and one must search this for the arrangement that makes the best use of all of the available resources. People can do this well for a little while, but for millions of instructions it is just too much effort. However, a compiler can dogmatically apply these optimizations across an entire program and in the end produce more efficient code.

Because RISC ISAs are specifically designed to allow their implementers the flexibility to incorporate a greater number of these sorts of enhancements, they depend even more than CISC ISA's on good compiler technology.

Unfortunately, because of the lag between development of an architecture and a mature optimizing compiler for it, architects often base design decisions for the next generation on inaccurate assumptions. The result has been that newer architectures are incorporating more compiler-like optimizations into the hardware. However, this added complexity makes it even more difficult to build a good optimizing compiler!

Where has RISC gone?

Today we see far fewer architectures dominating the market. The most popular for the desktop is the Intel architecture and its AMD clones, which are CISC in design. How did this happen if RISC offers superior performance? Part of the answer is the establishment of a vast code base that uses the architecture. It would be worth sacrificing some performance to avoid having to recompile that code (and in some cases, source code no longer exists and it is would be necessary to rewrite it). But would we actually sacrifice performance? Not really, because internally, these architectures actually run a RISC instruction set. The CISC instructions are translated, on the fly, into those simpler instructions, and cached in that form. We do still pay a penalty, but it is for the extra hardware that does the translation (which takes up a little more space on the chip, and consumes some extra power).

Embedded processors mostly run on the Arm platform, which is mainly a RISC architecture, but has some CISC-like features in the way that it combines multiple operations into a single instruction. In the high performance computing arena, we find the Intel CISC architecture in large cluster systems, but many of these machines are built with the IBM Power architecture that is RISC with some CISC complications, as well as the IBM Blue Gene architecture, derived from a subset of Power, and being more of a vector-oriented RISC design. The IBM Cell architecture, which includes a Power processor and a set of vector units, is also RISC.

Thus, for all of the research that was done to show that RISC or CISC was superior to the other in some way, we have actually settled on a set of designs that blend the two approaches. That is a common thread in computer architecture – whenever it seems that we have to choose between two alternatives, it is more likely that we really should be looking for a happy middle ground that balances between them.