The Instruction Set Architecture Level

270

The ISA Level

- The ISA level is positioned between the Microarchitecture Level and the Operating System Level.
- Historically, this level was developed before any of the other levels, and in fact, was originally the only level.
- This level is sometimes referred to simply as ‘‘the architecture’’ of a machine or sometimes (incorrectly) as ‘‘assembly language.’’

271

The ISA Level

- The ISA level is the interface between the software and the hardware.
- While it might be possible to have the hardware directly execute programs written in C, C++, Java, or some other high-level language, this would not be a good idea.
- The performance advantage of compiling over interpreting would then be lost. Furthermore, to be of much practical use, most computers have to be able to execute programs written in multiple languages, not just one.

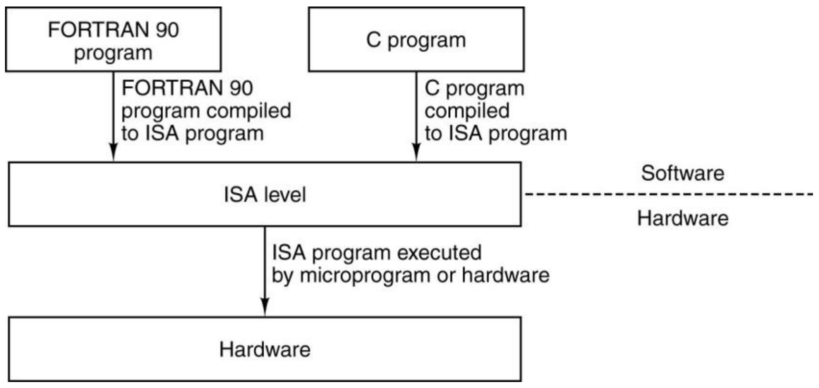
272

The ISA Level

- The approach is to have programs in various high-level languages be translated to a common intermediate form—the ISA Level—and build hardware that can execute ISA-Level programs directly.
- The ISA Level defines the interface between the compilers and the hardware.
- It is the language that both have to understand.

273

ISA Level


FORTRAN 90 program
C program
FORTRAN 90 program compiled to ISA program
C program compiled to ISA program
ISA level
Software
Hardware
ISA program executed by microprogram or hardware
Hardware

The ISA Level is the interface between compilers and the hardware.

274

Backwards Compatibility

- A new generation of machine must be able to run old programs without change.
- A new generation of machine will typically have new instructions and other features that can be exploited only by new software.
- New features can always be added provided backward compatibility is maintained, ensuring old programs run on the new generation of machine.
- The Industry challenge then becomes building better machines subject to the backward compatibility constraint.
- A good ISA design has significant advantages over a poor one, particularly in raw computing power vs. cost. Different ISAs might account for a difference of as much as 25% in performance.

275

What Makes a Good ISA

- A good ISA should define a set of instructions that can be implemented efficiently in current and future technologies, resulting in cost-effective designs over several generations.
- A poor design is more difficult to implement and may require many more hardware and/or more memory to execute a typical program.
- A good ISA should provide a clean target for compiled code.
- It should make the hardware designers happy (be easy to implement efficiently) and make the software designers happy (easing good code generation).

276

Properties of the ISA Level

- The ISA Level interfaces with the machine-language and the output of the compiler.
- To produce ISA-Level code, the compiler writer has to know
  - what the memory model is,
  - what registers there are,
  - what data types and instructions are available,
  - and so on.
- The collection of all this information defines the ISA Level.

277

Properties of the ISA Level

- The ISA Level typically implements two modes of operation:
- Kernel mode is a privileged mode used to run the operating system. In this mode, all instructions can be executed.
- User mode is intended to run application programs and does not permit certain sensitive instructions (such as those that manipulate the cache directly) to be executed.

278

Memory Models

Issues already covered
- Technologies
- Words
- Endianness

279

Registers

Issues already covered
- Special-Purpose Registers
  PC, SP, Status, …
- General-Purpose Registers
- Kernel Mode, User Mode

280

Instructions

The main feature of the ISA level is the set of machine instructions. These control what the machine can do.

There are always
- LOAD and STORE instructions for moving data between memory and registers. (In Samphire these actions are implemented using mov instructions)
- MOVE instructions for copying data among the registers. (In Samphire these are implemented using push and pop instructions)
- Arithmetic instructions (Samphire: add, sub, mul, div, mod, etc)
- Boolean instructions (Samphire: and, or, not, xor, etc)
- Instructions for comparing data items and (Samphire: cmp)
- Branching on the results (Samphire: jz, jnz, js, jns, jo, jno, etc)

281

Data Types

Data comes in different types and different formats

Integer
Floating point numbers
Characters
…

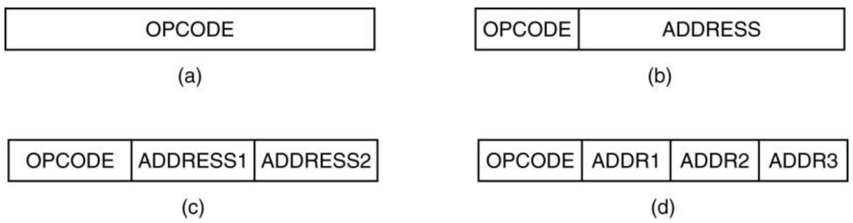
A key issue is whether there is hardware support for a particular data type.

Hardware support means that one or more instructions expect data in a particular format, and the user is not free to pick a different format.

Which bit in a number represents the sign of that number?
How many bits are used to represent a number?
Explicit support for double-precision?
Are calculations performed in the hardware on 64-bit numbers or
in the software on two 32-bit numbers?

282

Instruction Formats

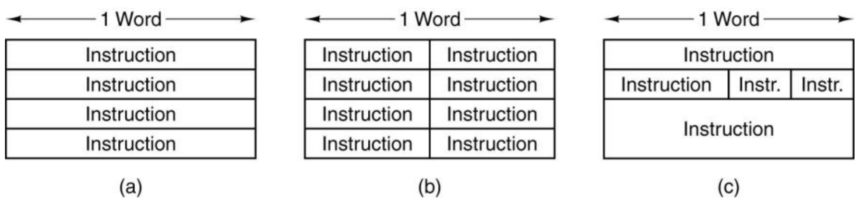


An instruction consists of an opcode, usually along with some additional information such as where operands should come from and where results should go. The general subject of specifying where the operands are (i.e., their addresses) is called **addressing**

Four common instruction formats:
(a) Zero-address instruction. (b) One-address instruction
(c) Two-address instruction. (d) Three-address instruction.

283

Instruction Formats



Some possible relationships between instruction and word length.

284

Instruction Formats

On some machines, all instructions have the same length; on others there may be many different lengths.

Instructions may be shorter than, the same length as, or longer than the word length.

Having all the instructions be the same length is simpler and makes decoding easier but often wastes space, since all instructions then have to be as long as the longest one.

Instruction Format Design Criteria
The difficulty of designing Instruction Formats should not be underestimated.

The decision must be made early in the design of a new computer. If the computer is commercially successful, the instruction set may survive for 40 years or more!

All things being equal, short instructions are better than long ones.

A program consisting of *n* 16-bit instructions takes up only half as much memory space as *n* 32-bit instructions.

285

Instruction Formats

Smaller instruction lengths require less memory bandwidth*. An increasingly common constraint on processors comes from the inability of the memory system to supply instructions and operands as rapidly as the processor can consume them. The bandwidth bottleneck applies not only to the main memory but also to all the caches.

If the bandwidth of an instruction cache is t bits per second (tps) and the average instruction length is r-bits, the cache can deliver at most t/r instructions per second.

This is an *upper limit* on the rate at which the processor can execute instructions.

The rate at which instructions can be executed (i.e., the processor speed) may also be limited by the instruction length. Shorter instructions can typically be executed faster. Since modern processors can execute multiple instructions every clock cycle, fetching multiple instructions per clock cycle can be crucial.

*Bandwidth refers to the amount of information that can be handled in a given time.

286

Instruction Formats

However, minimizing the size of instructions may make them harder to overlap. Therefore, striving for minimum instruction size must be weighed against the time required to execute them individually and sequences of them.

There must be sufficient room in the instruction format to express all the operations desired.

If we require our machine to be able to perform 2^n operations, then we need at least n bits to hold each one. Otherwise, there simply will not be enough room in the opcode to represent all instruction opcodes – not to mention operands.

The number of bits in an address field is important

This leads to a trade-off between the size of the instruction and how much of the memory that can be directly accessed.

287

Addressing

Most instructions have operands, so some way is needed to specify where they are. This subject is called addressing.

Addressing Modes

Immediate Addressing

The simplest way for an instruction to specify an operand is for part of the instruction - rather than a description of where to find it.

Such an operand is called an immediate operand because it is automatically fetched from memory when the instruction itself is fetched; hence it is immediately available for use.

MOV	R1	4

An immediate instruction for loading 4 into Register 1, R1.

Immediate addressing does not require an extra memory reference to fetch the operand. However, only a limited size constant can be supplied this way.

Addressing

Direct Addressing

Another method for specifying an operand in memory is just to give its full address. This mode is called direct addressing.

Like immediate addressing, direct addressing is restricted in its use: the instruction will always access exactly the same memory location. So while the value can change, the location cannot.

Thus direct addressing is used to access variables whose address is known at compile time - global variables. Nevertheless, many programs have global variables, so this mode is widely used.

Addressing

Register Addressing

Register addressing is conceptually the same as direct addressing but specifies
a register instead of a memory location.

Because registers are so important (due to fast access and short addresses) this addressing mode is most common on most computers.

Many compilers go to great lengths to determine which variables will be accessed most often (for example, a loop index) and put these variables in registers.

This addressing mode is known simply as register mode. In load/store architectures such as the ARM architecture, nearly all instructions use this addressing mode exclusively.

290

Addressing

Register Indirect Addressing

In this mode, the operand being specified comes from memory or goes to memory, but its address is not hardwired into the instruction, as in direct addressing.

Instead, the address is contained in a register. An address used in this manner is called a pointer. A big advantage of register indirect addressing is that it can reference memory without paying the price of having a full memory address in the instruction.

291

Addressing

Indexed addressing.
It is frequently useful to be able to reference memory words at a known offset from a register. Addressing memory by giving a register (explicitly or implicitly) plus a constant offset is called indexed addressing

Based-indexed addressing
Some machines have an addressing mode in which the memory address is computed by adding up two registers values plus an (optional) offset. Sometimes this mode is called based-indexed addressing. One of the registers is the base and the
other is the index.

Addressing Mode	Core i7	ARM	Atmega
Immediate	X	X	X
Direct	X		X
Register	X	X	X
Register Indirect	X	X	X
Indexed	X	X	
Based-indexed		X	

292