# Chapter 2

# The Instruction Set Architecture (The ISA)

## 2.1 The ISA: the interface of the computer to the outside world

Every computer has an instruction set architecture (ISA). The ISA defines the interface of the computer to the outside world. It is the interface between the software (what the programmer wants the hardware to do) and the hardware (what the hardware agrees to deliver). It is a contract between the software and the hardware, and in fact that contract is best specified by contributions from both hardware and software engineers. There are plenty of bad examples of ISAs that were specified by hardware engineers without any input from software developers that ended up as disasters in the marketplace because software developers found it unwieldy to write programs exploiting the computer's ISA.

### 2.1.1 More than just a set of instructions

Many people incorrectly think of the ISA as the set of instructions the computer can execute. And certainly, the set of instructions in the ISA is an important part of the ISA. BUT it is not the only part. Everything about the computer that the software needs to know to specify what it needs the hardware to do is part of the ISA. As we have said, how the hardware actually does each task – underneath the covers – the software does not need to know. It is enough for the software to know that the task specified by the software was carried out by the hardware in a way that agrees with the dictates of the software.

### 2.1.2 Not the Microarchitecture

It is important to distinguish the ISA from the microarchitecture of a computer. Since the ISA is the interface of the computer to the outside world, everything in the ISA is visible to the software which directs the activity of the computer. On the other hand, the microarchitecture of the computer is the structure that implements the ISA.

### 2.1.3 An analogy

A useful analogy from the world you are familiar with that illustrates that difference is the automobile. The ISA is the interface of the auto to the driver. The steering wheel is part of the auto's "ISA." It enables the driver to turn left or right. The brake and accelerator (and the clutch if there is a clutch) pedals on the floor, and the ignition system are all parts of the ISA. That is, the auto's "ISA" is everything the driver needs to know to operate the automobile.

The microarchitecture on the other hand is what is underneath the hood: how many pistons, the actual braking system that is invoked when the driver steps on the brake pedal, for example. The driver knows that he/she can adjust the speed of the automobile by how much he/ she steps on the accelerator without having any idea of what makes the car go faster or slower in response to the degree to which the accelerator pedal is depressed.

This differentiation is true of the computer as well. The programmer can specify an instruction ADD X,Y,Z that the computer will accomplish without having any idea how the computer will accomplish that task. And, indeed, there are many algorithms for adding two numbers, depending in part on how the numbers are represented, where they are stored, and also in part on how soon you want the result. In Chapters 3 and 4, we will delve deeply into the various structures of the microarchitecture. Right now we will study the ISA.

## 2.2 Characteristics of an ISA

Table 2.1 lists the common characteristics of an ISA, and also provides as an example what those characteristic are for the LC-3b, an ISA we will be studying in this book.

### 2.2.1 Memory

Different ISAs can accommodate different maximum sizes of memory. The size of memory is the number of unique memory locations. The LC-3b specifies a 16-bit address for each memory location, ergo the maximum size is $2^{16}$ locations. (Actually, 512 addresses are used for another purpose which we will get to in chapter 9, so the LC-3b really only specifies $2^{16}$ - 512 memory locations. Another characteristic of the ISA is memory's addressability, the number of bits stored at each location. Memory locations in the ISA are byte addressable; i.e., 8 bits are stored in each memory location.

### 2.2.2 Registers

There are several registers in every ISA. Among them are general purpose registers that are used to perform computations. The LC-3b has 8 general purpose registers. Most computers have far more. For the LC-3b the instruction ADD X,Y,Z, is really ADD Ra,Rb,Rc; i.e., X,Y, and Z are general purpose registers, and a,b, and c are integers from the set 0,..7; for example, ADD R1,R3,R5. The computer carries out the work of that instruction by adding the values stored in general purpose registers R3 and R5, and storing the sum in R1. Some ISAs have no general purpose registers, preferring to use a stack to store an instruction's source operands and the result of execution.

### 2.2.3 Condition codes

Most ISAs have several one-bit registers called condition codes that describe the nature of a specific value. The LC-3b ISA has three such one-bit registers which carry the information that the designated value is negative, zero, or positive. Actually, two one-bit registers would have been sufficient since every value has to be one of the three, so we know (for example) if the value is not negative and not zero, it must be positive. We specified three condition codes in order to

| **Characteristics of an ISA** | | **LC-3b Details** |
|---|---|---|
| Processor State (memory, registers) | Memory Addressability | Byte |
| | Memory Address Space | $2^{16}$ |
| | Registers | 8 GPR, Condition Codes (N, Z, P) |
| | Word Length | 16 bits |
| | Program Counter (Instruction Pointer) | |
| | Process Status Register | Contains Privilege, Priority, CC |
| Privilege | | 2 levels (Supervisor, User) |
| Priority | | 8 levels |
| Instruction Format | | Fixed length, uniform decode (16 bits) |
| Endian-ness | | Little Endian |
| Instructions | Opcode | 14 opcodes (XOR, SHF, LDB, etc.) |
| | Addressing Modes | PC-relative, Register + Offset |
| | Data Types | 2's complement 16-bit integers, bit vector |
| | Addressing Scheme | Three-address machine (Load/Store ISA) |

Table 2.1: Characteristics of an ISA

simplify the resulting programs. Most ISAs specify only N and Z for negative and zero, and include additional condition codes for detecting other information, most commonly overflow situations and carry generation.

### 2.2.4 The Program Counter

Most ISAs identify a program counter (PC), more accurately described as an instruction pointer (IP), which keeps track of the next instruction in a program to be carried out.

### 2.2.5 The Process Status Register

Most ISAs include a status register to specify various characteristics of the process, for example the priority (urgency) of that process gaining control of the computer and privilege, (rights) accorded to the process specifying what instructions it is allowed to execute and what part of memory it is allowed to access. The LC-3b has 8 levels of priority (levels 0 to 7) and two levels of privilege, privileged (supervisor) and unprivileged (user). The LC-3b has only one instruction – Return from Trap or Interrupt (RTI) – that requires the process to be in privileged mode in order to have the right to execute it. Most ISAs have two levels of privilege, although there are some that have as many as 4.

### 2.2.6 System Architecture

Parts of the ISA specify activity that is only available to the operating system, for example, structures available for managing the virtual memory system. These structures include registers for differentiating the memory associated with each process, a translation mechanism for converting virtual addresses to physical addresses, a protection scheme that allows multiple processes to co-exist in the computer's memory without improperly accessing memory it has no right to access. We will discuss virtual memory in Chapter 7. The LC-3b has a trap vector table and an interrupt and exception vector table that the operating systems uses to handle interrupts, exceptions, and TRAP requests.

### 2.2.7 Endianness

Although most computer memories store 8 bits (one byte) of information in each memory location, most values require more than 8 bits to represent its value. This is accomplished by using more than one location to store the value. For example, if a value requires 64 bits to represent it, 8 memory locations are needed. The value would be stored in memory locations X, X+1, X+2, ... X+7, and it would be accessed by referring to location X. The question then arises: Are the bits in location X bits[7:0] of the 64 bit value, or are they bits[63:56] of the 64 bit value. The ISA specifies which byte of the 64 bit value is contained in memory location X. We refer to the choice as big endian if the contents of bits[63:56] are contained in location X and little endian if the contents of bits[7:0] are contained in location x.

## 2.3 The set of instructions

Many ISAs have hundreds of instructions in their instruction sets. The LC-3b has 14. Figure 2.1 shows the format for each instruction.

### 2.3.1 General notions

First some general comments about all instructions in the instruction set.

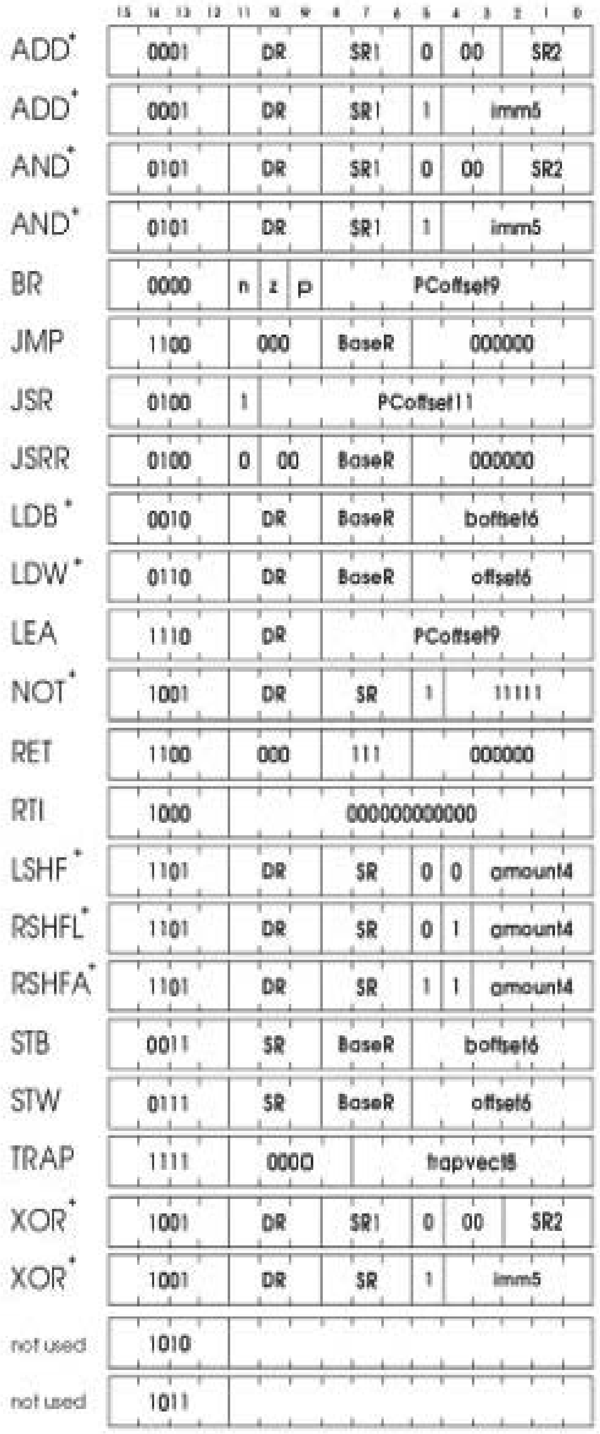
#### The instruction is the atomic unit of processing

We refer to the instruction as the atomic unit of processing to indicate that the computer either executes the entire instruction or it executes none of it. For example, if a problem occurs during the execution of an instruction that does not allow the instruction to complete execution, the microarchitecture undoes the work accomplished thus far and returns the computer to its state before that instruction was fetched. For example, consider an ISA that contains an instruction ADD R1,R2,(R3)+, where one source is from memory, its location is specified in R3, and the contents of R3 is incremented after using it to access the source operand. Suppose the addition results in an exception. Undoing the instruction means not only decrementing the PC to the address of the instruction that caused the exception. It also means decrementing R3, i.e., undoing the post-increment of R3.

#### Processing instructions

The hardware executes programs specified by the software. The hardware guarantees that its execution of a program will produce the same result as if the program was executed one instruction at a time, in the order specified by the program. Executing a program in program order means the following: The computer is in $\text{state}_0$ before the program starts execution. Execution of the first instruction modifies $\text{state}_0$ of the computer to $\text{state}_1$. The second instruction executed modifies $\text{state}_1$, producing $\text{state}_2$, and so forth until the program terminates.

For example, suppose the ISA was the LC-3b and the instruction to be executed next is ADD R1,R2,R3. Execution of this instruction will change the state of the computer in three ways: It will increment the PC by 2, since memory is byte addressable and LC-3b instructions require 16 bits. That is, 2 bytes of memory are needed to construct an instruction. It will load into R1 the result of adding the contents of R2 and R3. And, finally, it will set the condition codes N, Z, and P depending on whether the result loaded into R1, is negative, zero, or positive.



| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ADD$^+$ | 0001 | | | | DR | | | SR1 | | | 0 | 00 | | SR2 | | |
| ADD$^+$ | 0001 | | | | DR | | | SR1 | | | 1 | imm5 | | | | |
| AND$^+$ | 0101 | | | | DR | | | SR1 | | | 0 | 00 | | SR2 | | |
| AND$^+$ | 0101 | | | | DR | | | SR1 | | | 1 | imm5 | | | | |
| BR | 0000 | | | | n | z | p | PCoffset9 | | | | | | | | |
| JMP | 1100 | | | | 000 | | | BaseR | | | 000000 | | | | | |
| JSR | 0100 | | | | 1 | PCoffset11 | | | | | | | | | | |
| JSRR | 0100 | | | | 0 | 00 | | BaseR | | | 000000 | | | | | |
| LDB$^+$ | 0010 | | | | DR | | | BaseR | | | boffset6 | | | | | |
| LDW$^+$ | 0110 | | | | DR | | | BaseR | | | offset6 | | | | | |
| LEA | 1110 | | | | DR | | | PCoffset9 | | | | | | | | |
| NOT$^+$ | 1001 | | | | DR | | | SR | | | 1 | 11111 | | | | |
| RET | 1100 | | | | 000 | | | 111 | | | 000000 | | | | | |
| RTI | 1000 | | | | 000000000000 | | | | | | | | | | | |
| LSHF$^+$ | 1101 | | | | DR | | | SR | | | 0 | 0 | amount4 | | | |
| RSHFL$^+$ | 1101 | | | | DR | | | SR | | | 0 | 1 | amount4 | | | |
| RSHFA$^+$ | 1101 | | | | DR | | | SR | | | 1 | 1 | amount4 | | | |
| STB | 0011 | | | | SR | | | BaseR | | | boffset6 | | | | | |
| STW | 0111 | | | | SR | | | BaseR | | | offset6 | | | | | |
| TRAP | 1111 | | | | 0000 | | | | trapvect8 | | | | | | | |
| XOR$^+$ | 1001 | | | | DR | | | SR1 | | | 0 | 00 | | SR2 | | |
| XOR$^+$ | 1001 | | | | DR | | | SR | | | 1 | imm5 | | | | |
| not used | 1010 | | | | | | | | | | | | | | | |
| not used | 1011 | | | | | | | | | | | | | | | |

Figure 2.1: Format of the LC-3b Instruction Set. NOTE: + indicates instructions that modify condition codes

Clever microarchitects have come up with many ways to speed up the execution of a program. One way is by executing the instructions of the program out-of-order. Such speed-ups are good provided the result produced by doing so is identical to the result that would have occurred if the program had been executed one instruction at a time in program order.

### Instruction Format

The formats of the 16 different instructions are shown in Figure 2.1. Note that all instructions are 16 bits and the opcode is always in bits[15:12]. This is referred to as "fixed length, uniform decode." Having a fixed length, uniform decode format means the hardware can fetch more than one instruction at a time and know immediately where the opcode of each instruction is located. This enables many instructions to be decoded concurrently very easily.

Intel's x86 ISA is neither fixed length nor uniform decode. As to length, each x86 instruction can consist of from 1 to 16 bytes of information, depending on the work to be performed by that instruction. Furthermore, the x86 instruction can append up to four prefixes to the first bytes of an x86 instruction which means the opcode could be the first, second, third, fourth, or fifth byte of the instruction, making decoding multiple instructions concurrently a nightmare.

### Load/Store ISA

The LC-3b is a Load/Store ISA. A load/store ISA is defined as an ISA where accessing memory to obtain a source operand and operating on that operand in the same instruction is not allowed. Also, operating on source operands by means of an operate instruction and storing the result of that operation in memory in the same instruction is also not allowed. Since an instruction can not perform a Load and an operate in the same instruction's execution, or perform an operate and a store in the same instruction's execution, the effect of these constraints is to require operate instructions like ADD, SUB, MUL, and DIV to obtain their source operands from registers and to store the result of the operation in a register.

### Addressing Modes

The mechanism used to obtain the memory address for a load or store instruction is called its "addressing mode." One could specify more than a dozen possible addressing modes for obtaining a memory address. The LC-3b has very few. An address can be formed by adding an offset to the contents of a general purpose register or adding an offset to the Program Counter. The x86, on the other hand has several addressing modes, including one that enables accessing an element (row and column) of a two dimensional array by means of its SIB byte. The VAX ISA, produced by Digital Equipment Corporation in the 1980s had 13 addressing modes, obtained by including more than one register, each performing a different function, to obtain the desired memory address. Their addressing modes included post-incrementing and pre-decrementing registers, useful for instructions in the loop body of a for loop when each execution of the loop body needs the next value in a table of values in memory.

**Data Types**

A Data Type is a representation of information such that there are instructions in the ISA that operate on that representation. The LC-3b has three data types: 2's complement integer, bit vector, and character string. Most ISAs also have multiple floating point data types, generally consisting of 16, 32, and 64 bits. The VAX found it useful to incorporate the doubly-linked list as a data type and included two instructions INSQUE and REMQUE for efficiently inserting and removing nodes from a doubly-linked list.

### 2.3.2 Three types of instructions

There are basically three kinds of instructions: (1) instructions that operate on data, (2) instructions that move data between the processor, the memory, and input/output devices, and (3) instructions that control the flow of instructions in the program (i.e., they determine which instruction is to be executed next.

The LC-3b has 5 operate instructions (ADD, AND, XOR, SHF, and LEA), 4 data movement instructions (LDB, LDW, STB, and STW), and 5 control instructions (BR, JMP, JSR(R), RTI, and TRAP). Figure 2.1 shows the format of the 14 instructions in the LC-3b ISA. If you count them, you will see 24 instructions, not 14. The discrepancy is due to three things:

(1) The operate instructions provide a variation that allows one of the source operands to be part of the instruction, rather than requiring all operands to be obtained from general purpose registers. These operands are referred to as "immediates" because they are immediately available in the instruction and do not have to be sourced from a general purpose register.

(2) The JSR instruction has two addressing modes for obtaining the starting address of a subroutine call, the contents of a register or an address formed by adding an offset from the instruction to the PC.

(3) Some of the 24 instructions listed are special cases of the actual 14. For example, the opcode of both NOT and XOR is 1001, that is, NOT is a special case of XOR, i.e., "A XOR 1 = NOT(A)." Also, the opcode for both JMP and RET is 1100, i.e., RET is a special case of JMP, where the address the program jumps to is contained in Register 7, That address is the linkage from the subroutine back to the location after its corresponding JSR instruction.

**Operate Instructions**

Operate instructions operate on data present in the general purpose registers. ADD, AND, and XOR source two general purpose registers or a general purpose register and an immediate operand, perform the operation specified by the opcode and store the result in a general purpose register.

The SHF instruction shifts its one source operand left or right up to 15 bits, and fills in the vacant bits with 0 if the instruction performs a logical shift or the sign bit if the instruction is an arithmetic right shift.

The LEA instructions constructs a memory address and loads it into one of the general purpose registers.

### Data Movement Instructions

The LC-3b has four data movement instructions, load and store byte and load and store word. Data movement between the processor and memory or Input/Output devices use these four instructions. Data transferred from an I/O device to the processor, – “input” – uses the LDB and LDW instructions. Data transferred from the processor to an I/O device – “output” – uses the STB and STW instructions. Each I/O device has some number of registers associated with it. Each register is assigned an address from the $2^16$ addresses in the memory address space. In the case of the LC-3b, those addresses are from the range xFE00 to xFFFF.

### Control Instructions

Unlike the Operate and data movement instructions which execute instructions in the computer program sequentially, the control instructions change the control flow from performing the next sequential instruction in the program to whatever instruction the execution of the program requires. During the instruction cycle of an operate or data movement instruction, the Program Counter is incremented by 2 so the next instruction executed will be the next sequential instruction in the program. During the execution of a control instruction, the PC is loaded with the address of the instruction that the program requires to be executed next.

There are 5 control instructions. BR is a conditional branch instruction which, based on the condition codes that were last set, will load the PC with either the next sequential address or an address specified in the BR instruction.

JSR is a subroutine call that sets PC to the address specified by its addressing mode and provides a linkage back to the instruction following the JSR instruction after the subroutine completes execution.

JMP loads the PC with the address specified by the addressing mode of the JMP instruction. It is the equivalent of a GOTO statement in a high level language.

TRAP is essentially a subroutine call to the operating system to perform a task such that the programmer is not able to specify the code needed to perform the task.

RTI (Return from trap or interrupt) is the last instruction in a TRAP, exception, or interrupt service routine. After the service routine completes execution, RTI provides the linkage back to the point in the program where the TRAP, exception, or interrupt was initiated.

### 2.3.3 Vector architecture

Many ISAs, in particular ISAs in high performance computers, have a set of instructions called vector instructions and registers called vector registers that provide significant performance benefits. The structures are collectively referred to as a vector architecture. The LC-3b does not have a vector architecture. We will discuss vector architectures in Chapter 12.

### 2.3.4 Pragmas

Pragmas (or pseudo-ops as they are sometimes called) are not instructions. They are included here because they are similar to instructions and it is important to understand the difference between pragmas and instructions. Instructions are part of a program and get executed. Pragmas are part of a program that do not get executed. Rather, they are messages to the compiler or assembler, geared to helping the translator produce a "better" program.

## 2.4 Other Example ISAs

There have been many ISAs introduced over the years. Digital Equipment Corporation introduced (among many others) the PDP-11 in 1970, the VAX in 1977, and the Alpha in 1991. IBM introduced many ISAs during the 1950s and early 1960s, until converging on one ISA, the IBM System 360 in 1964. Today, IBM mostly celebrates two ISAs, the Z series (which is the evolution of the System 360) and the PowerISA (which evolved from the Power-PC ISA, introduced by a collaboration of engineers from IBM, Motorola, and Apple, in the 1990s. We will briefly describe characteristics of two ISAs that are widely discussed today.

### 2.4.1 Intel's ISAs

Intel has introduced several ISAs, most notably the x86, starting with the 8086 in 1979. The ISA has evolved over the ensuing 45 years, Intel adding new features from time to time, but never losing its commitment to its customer base, ensuring that old programs can run correctly on new hardware. x86 added virtual memory with the 80386 in the mid-1980s, and conditional execution with the CMOV instruction of the Pentium Pro in the late 1990s. Along the way, new implementations increased the size of data elements and the size of the memory address space. The 8086 featured 16-bit data elements, the VAX increased the size to 32-bits. Today, x86 ISAs support 64-bit data elements.

But x86, although the most well-known of Intel's ISAs, was not Intel's only ISA. Among the others was the i432, an ISA targeted for capability-based computing, where security was the deign point. Unfortunately, performance was found wanting, resulting in the i432 not becoming a winner in the marketplace. In the mid-1980s, Intel offered the i860, one of Intel's first one million transistor microprocessors. In the mid-1990s, Intel offered the EPIC ISA, featuring 64 bit data elements, rather than 32-bit data elements that were being produced on their x86 chips. The intent of EPIC was for server applications.

## The x86 instruction set

The x86 instruction format is variable length, non-uniform decode. Figure 2.2 illustrates the instruction format.

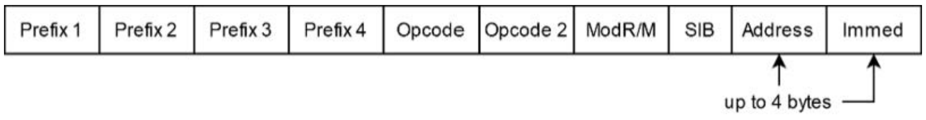


Figure 2.2: Format of the x86 Instruction

Instructions can be as small as a single byte (just the opcode), with no operands. Or it can be as large as 16 bytes, as shown in Figure 2.2. The opcode could be the first byte of the x86 instruction if there are no prefixes, or it could be the second, 3rd, 4th or fifth byte if the instruction had one, two, three or four prefixes. Prefixes can specify special features associated with the instruction, resulting in a richer instruction, such as the REPEAT prefix which directs the hardware to efficiently execute the instruction more than once. Prefixes can also tailor the instruction by recognizing that the program was written for an older implementation so the instruction needs to be modified to run correctly on a more recent implementation.

The x86 instruction consists of from one to four prefixes, followed by an opcode that requires either one or two byte, the ModR/M byte, SIB byte, an absolute address (from one to four bytes, and an immediate operand consisting of from one to 4 bytes. Initially, there were fewer than 256 opcodes so originally one byte for the opcode was enough. When the number exceeded 256, a second opcode byte was required. ModR/M is the workhorse byte of the x86 instruction. It organizes all parts of the instruction and gets it ready for execution. For example, the ModR/M byte knows whether the SIB byte, 4 byte absolute address, and up to 4 byte immediate operand are part of the instruction.

x86 is not Load/Store. For example, A single instruction can perform an operation on source operands and store the result in memory. x86 is a two-address machine. One of the addresses can be used for a source operand and a destination operand in the same instruction. For example, an instruction can perform the following: R1 $\leftarrow$ R1 + memory_operand. If the value in R1 before execution of such an instruction is needed after this instruction, the contents of R1 must be copied to a separate register or memory location before the addition takes place.

x86 registers can be 8 bits, 16 bits, 32 bits, ... 512 bits, depending on their needs, Memory of current x86 processors is byte-addressable. Memory has a 64 bit address space.

### 2.4.2 RISCV

RISCV (pronounced RISC 5) is a recent addition to our collection of ISAs. Its major attraction seems to be it is OPEN SOURCE, which is very attractive to hardware developers tired of paying royalties to Intel and Arm for use of their ISAs. The "5" comes from the fact that it was the 5th ISA from the Berkeley CS Division since its first RISC processor in 1979. It had essentially nothing relating it to RISC 1,2,3, and 4 other than it came from the same research group.

The ISA is actually subset into more than a dozen subsets, each one dealing with a particular set of problems it is tuned to attack. There are three subsets that deal with integer arithmetic, where the integers are 32 bits, 64 bits and 128 bits. There are also three subsets that deal with standard floating point computations, where the float data types are 32 bits, 64 bits, and 128 bits. There is a vector subset, an integer MUL/DIV subset, an atomic instruction subset, a bit-manipulation subset, etc. Developers are free to choose only the subsets that are useful to their design point. The only constraint is that at least one of the three integer subsets must be included in the set chosen.

The 32 bit Integer subset has the following characteristics: It is Load/Store, uniform decode, Little Endian, uses general purpose registers (not condition codes) for dealing with conditional branches, has 32 general purpose registers, R0 is hardwired to 0, R1 is used for subroutine all return linkage. Loads and stores allow unaligned access. The 32-bit integer unit has 47 distinct opcodes.

## 2.5 Tradeoffs

As we have said many times, if computer architecture is a science, it is a science of tradeoffs. With respect to the ISA, we will examine a number of tradeoffs.

### 2.5.1 Dynamic/Static Interface

The Dynamic/Static Interface specifies the point in the translation of a program from its specification in a high-level programming language to the control signals in the microarchitecture that direct the hardware's processing of the program. That point is the ISA; that is, the point where compilation ends and run-time behavior begins. An instruction like AOBLEQ (add one and branch if less than or equal) specifies the iteration control of a for loop. This represents placing that interface close to the high-level language, while an instruction like LD R1,A (load R1 with the contents of memory location A) represents an instruction very close to the control signals of the microarchitecture. Placing the ISA close to the high-level language provides opportunity for the microarchitect to exploit concurrency, thereby decreasing the time it takes to execute the program. Placing the ISA close to the control signals of the microsequencer helps ensure that no unnecessary processing will be done.

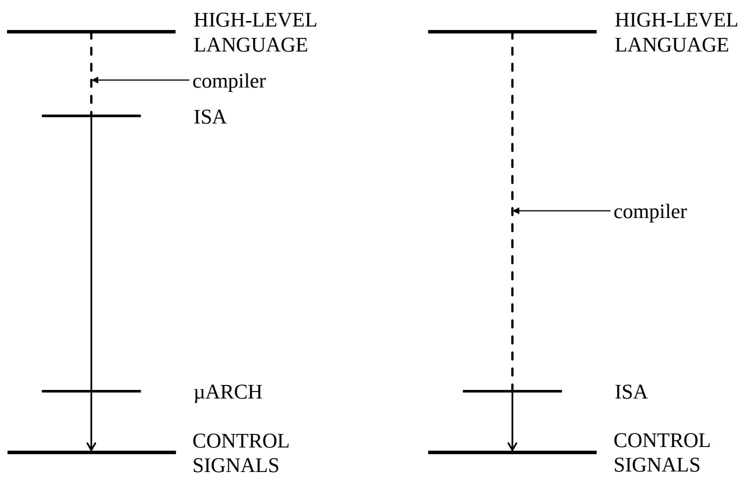


Figure 2.3: Difference in Dynamic/Static interface

### 2.5.2 Complex instructions – part of the ISA or keeping the ISA lean

Every complex instruction added to the ISA provides an opportunity for speeding up the processing at the expense of extra logic, area on the chip, and energy consumed. The following are instructions taken from ISAs where the decision was made that it is worth performing the task as a single heterogeneous instruction, rather than a program segment of more vanilla instructions.

EDITPC: An instruction that is part of the COBOL compiler for handling COBOL's "Picture" construct with a single instruction.

INDEX: An instruction that performs part of the computation needed to find the location of an element in a multi-dimensional subscripted array. It is very useful, for example, when dealing with a three dimensional array, where the three instruction sequences INDEX I, INDEX J, INDEX K can replace 30 or more vanilla instructions that perform bounds checking on the subscripts and locate the memory address of a single element of an array, ARRAY [i,j,k].

AOBLEQ: An instruction that can perform the iteration control of a for loop in a single instruction.

LDCTX/SAVCTX: two instructions to save the context of the process losing control of the computer and load the context of the next process to gain control of the computer.

FIND_FIRST: A single instruction to find the first bit of a bit vector that is set or cleared.

INSQUE/REMQUE: Two instructions that insert and remove nodes from a doubly linked list. The doubly linked list is the data type that is supported by these two instructions.

TRIADS: An instruction that performs more than one micro-op in its execution. The most common example is the Multiply-accumulate instruction that multiplies two source operands and adds the result to a running sum, very useful in performing the inner product of two vectors.

### 2.5.3 Condition codes vs general purpose registers

Condition codes get an extra piece of work done (for example, determining whether a value is positive negative or zero) at no cost in execution time, but it serializes the instruction stream since it is likely the next instruction in the program will change the condition codes. The alternative, using general purpose registers to store the condition removes the need to use the condition information immediately, but it costs the extra time for a compare instructions to evaluate the condition. It is worth noting that John Cocke attempted to get the best of both worlds by including in the ISA of IBM's RS6000 a 32 bit register consisting of 8 sets of four condition codes. This allowed the computer to get the condition code information at no cost of an extra instruction while also removing the need for serializing the instruction stream as long as the program left the specific condition codes alone until they were needed.

### 2.5.4 Unaligned access of data

Allowing unaligned access to data means the programmer can remain clueless as to how memory is actually organized. However, it also means an unaligned access to memory takes two memory cycles instead of one. It is worth noting that Digital Equipment Corporation did not allow unaligned accesses on its PDP-11 ISA in 1970, allowed it on its VAX ISA in 1977, and did not allow it on its Alpha ISA in 1991. Its rationale in 1970 was that serious programmers should be expected know how memory is organized. In 1977, it decided that it was more important to provide support for naive programmers, even though it cost performance. By 1991, they changed their mind again. This time the rationale was that while programmers are mostly programming in a high level programming language and represented all variables symbolically, which means they did not need to think in terms of memory alignment, their programs had to be compiled and it was not unreasonable to expect compiler writers to understand memory organization.

### 2.5.5 Rich sets of memory addressing modes and data types

Some computers have a minimal set of addressing modes and data types. Others have a rich collection. Data types include multiple size integers and floating point numbers, bit vectors, character strings, doubly linked lists, trees, etc.

Addressing modes include indirect, index, auto-increment, auto-decrement, SIB byte, etc. The more there are, the more work for the microarchitect, but the easier job for the programmer. The fewer there are, the more streamlined the processor, and potentially the higher performance.

### 2.5.6 instruction format

We have already discussed the difference between fixed length uniform decode and variable length non-uniform decode. Fixed length uniform decode does less work per instruction, which increases the number of instructions in the program. The cache is less effective because these instructions need to be stored. The plus is that fetching and decoding more instructions each cycle is made much easier. Variable length non-uniform decode gets a lot more work done with each instruction and more effectively utilizes the cache. Fetching and decoding multiple instructions per cycle becomes much harder.

### 2.5.7 Synchronous vs Asynchronous

Synchronous means the inclusion of a clock, asynchronous means operations can operate as fast as they can and do not have to worry about a clock. One would assume asynchronous would be faster since the operations to not have to wait for a clock edge. Actually, the opposite is true since the absence of a clock means operations have to include some handshaking mechanism which is likely to be far more expensive than waiting for a clock edge, especially if the cycle time is small.

### 2.5.8 Security vs. Performance

Up to now, security has come at the expense of performance, the more structure that is involved in improving the security of a computer, the lower the performance. If one does not care about performance, one can design a highly secure system. The problem is that no one wants their computer to run like a dog. That is, everyone cares about performance.

### 2.5.9 PAGE SIZE

Most manufacturers are still specifying the memory page size as 4KB, even though more and more applications these days require very large data sets. Having a large page size means one Page Table Entry for each page. Retaining the 4KB page size means every 4kB requires its own PTE. For example, a page size of $2^18$ bytes requires one PTE if configured as a single entity. Storing it in 4KB increments means 64 PTEs, resulting in much slower translation from virtual to physical memory. A disadvantage of the large page size is the fragmentation of memory, which makes some of the physical memory unusable. Having multiple page sizes can severely reduce the amount of fragmentation.

### 2.5.10 Register set size

The larger the size of the set of general purpose registers, the less often we need to spill a register value to memory. However, the larger the number of registers, the longer a context switch takes if the architecture of the register set has to be saved/restored.