3. Instruction set design

Clearly the design of a new machine is not a smooth process; the designer of the architecture must be aware of the possible hardware limitations when setting up the instruction set, while the hardware designers must be aware of the consequences their decisions have over the software.

It is not seldom that some architectural features cannot be implemented (at a reasonable price, in a reasonable time, using a reasonable surface of silicon, or cannot be implemented at all!); in these cases the architecture has to be redefined. Very often small changes in the instruction set can greatly simplify the hardware; the converse is also true, the process of designing the hardware often suggests improvements in the instruction set.

The topic of the following sections is the design of the instruction set: what should be included in the instruction set (what is a must for the machine), and what can be left as an option, how do instructions look like and what is the relation between hardware and the instruction set are some of the ideas to be discusses.

The coming section tries to answer the question: should we have a rich set of instructions (CISC) or a simple one (RISC)?

3.1 RISC / CISC, where is the difference?

For many years the memory in a computer was very expensive; it was only after the introduction of semiconductor memories that prices began to fall dramatically. As long as the memory is expensive low end computers cannot afford to have a lot; there is a premium in a system with little memory to reduce the size of programs. This was the case in the 60s and 70s.

Lot of effort was invested in making the instructions smaller (tighter encoding) and in reducing the number of instructions in a program. It was observed that certain sequences of instructions occur frequently in programs like the following:

```
loop: ................
      ................
      DEC   r1    # decrement r1 (in r1 is the loop counter)
      BGEZ  loop  # branch if r1 >= 0
```

“Well let's introduce a new instruction that has to do the job of both” said the designers and this happened: the programmers was offered a new instruction which decrements a register and branches if the register is greater of equal to zero.

The same phenomenon happened in many other cases, like instruction that save/restore multiple registers in the case of procedure calls/returns, string manipulating instructions etc.

Another motivation that led to the development of complex instruction sets was the insufficient development of compilers; software was dominated by assembly language in so much that programmers found very pleasant and efficient to have powerful instructions at hand. In this trend many instructions became somehow like procedures, having parameters as the procedures may have, and saving memory in the same way procedures do.

The landmark for **CISC** architectures (Complex Instruction Set Computers) is the VAX family; introduced in 1977, the VAX architecture has more than 200 instructions, some 200 addressing modes and instructions with up to six operands. The instruction set is so powerful that a C program has almost the same number of assembly language instructions as the C source.

The main problem with CISC is that, due to different complexities, instructions require very different number of clock cycles to complete thus making very difficult efficient implementations (like pipelining); long running instructions render the interrupt handling difficult, while uneven sizes of instructions make the instruction decoding inefficient.

Due to the large number of instructions in the instruction set, the control part of a CISC machine is usually **microprogrammed**: the implication is a lower clock rate (a higher T_{ck}) than the one a hardwired implementation would allow.

As the compiler technology developed, people realized how difficult is to figure out what is the best instruction (sequence of instructions) to be generated by the compiler: simply said there are too many combinations of instructions to do the same job, when using a complex instruction set.
To summarize, remember that the CPU performance is given by:

$$CPU_{time} = IC * CPI * T_{ck}$$

where CPU_{time} is the time spent by a CPU to run a program (the effective time), IC is the instruction count, CPI is the average number of clock cycles per instruction, and T_{ck} is the clock cycle (assumed to be constant).

Basically the CISC approach to reducing the CPU_{time} was to reduce the IC as much as possible. As a side effect the CPI has increased (instructions are complex and require many clock cycles to execute), as well as T_{ck} due to microprogramming. Very few new CISC have been introduced in the 80s and many argue that CISC survive only due to compatibility to older successful computers: a prominent example is the Intel's 80x86 family of microprocessors.

In contrast with CISC architectures, the **RISC** (Reduced Instruction Set Computers) ones make an attempt to reduce CPU_{time} by decreasing CPI and T_{ck}. Instructions are simple and only a few basic addressing modes are provided. Simple instructions mean low CPI as well as the possibility of easier hardwired implementation (thus lowering the clock cycle T_{ck}).

However the major point is that simple instructions allow pipelined implementations (all instruction execute in the same number of clock cycles) which dramatically decreases the CPI (the ideal CPI in a pipelined implementation is 1); moreover pipelining also permits higher clock rates.

The disadvantage is an increase in the number of instructions (IC); the same program may have as much as twice the number of instructions (assembly language) as compared with the same program that uses a CISC assembly language.

In a CISC machine the CPI can be in the order of 10, while for a RISC pipelined machine the CPI is some where between 1 and 2: roughly speaking there is an order of magnitude difference between the two approaches.

Example 3.1 RELATIVE PERFORMANCE:

The same program is compiled for a CISC machine (like a VAX) and for a pipelined RISC (like a SPARC based computer). The following data is available:

$IC_{CISC} = 500000$

$IC_{RISC} = 1100000$

$CPI_{CISC} = 6.8$

$CPI_{RISC} = 1.4$

$T_{ck\ CISC} = 25$ ns ($25*10^{-9}$ s)

$T_{ck\ RISC} = 30$ ns

What is the relative performance of the two machines?

Answer:

$$\frac{CPU_{timeCISC}}{CPU_{timeRISC}} = \frac{5 * 10^5 * 6.8 * 25}{11 * 10^5 * 1.4 * 30} = \frac{850}{462} = 1.83$$

The RISC machine is by 83% faster than the CISC one.

It is important to observe that we have only compared the CPU times: however this does not say everything about the two machines; the overall performance is affected by the memory and the I/O. For the user it is the elapsed time which counts not the CPU_{time}.

Practically, at this moment, all major semiconductor producers are offering at least one RISC architecture.

3.2 How many addresses?

After we have introduced, in the previous sections, some general ideas about the hardware-software interaction, we shall be discussing, in more detail, about the instruction set.

Consider the following statement in a high level language (C for instance):
a = a + b + a * c;

For all familiar with a programming language is clear what the meaning of this statement is: take the value of a and multiply it with c, then add a and c to the above result; the result is assigned to the variable a.

We know what are the rules of precedence and associativity; these rules are also included in the high level languages (in the compiler more precisely). It is however obvious that we cannot expect the hardware to directly use these rules and to “understand” the whole sentence at once.

For sake of simplicity we require operations to be carried out in small steps; the result will be obtained after a **sequence of simple steps**. Incidentally this eliminates the need for the machine to know about the grouping rules.

Example 3.2 OPERATION SEQUENCE:

What is the sequence of operations (single statement per operation) that evaluates the statement:

```
a = a + b + a * c;
```

Answer:

```
t = a * c;
a = a + b;
a = a + t;
```

It is important to note that, in order to understand what the above sequence does, one has to know:

- how sequencing works: execute the first instruction, then the second and so on;

- what every operation does.

As an aside, the above sequence points out why our computers are said to be **sequential**.
In the example above t stands for temporary, an intermediate variable that holds the value of the first operation; in this case we could not assign the value of multiplication a * c to a because a is also needed for the second operation.

Which is the operation performed in the first statement of the answer in example 3.2? The spontaneous and yet not very correct answer is multiplication; it is not simple multiplication because the statement specify not only the **operation** to be performed but also **where** the result is to be stored (this is a computer statement, not a a simple mathematical equality). The proper name is therefore **multiply_and_store**, while for the second statement the proper name would be **add_and_store**.

Multiplication and addition are **binary** operations; multiply_and_store and add_and _store are **ternary** operations.

The operands which specify the values for the binary operation (which is a part of the ternary operation) are called **source operands**. The operand that specifies where the result is to be stored is called the **destination operand**.

Operands may be constants like in the following example:

a = b * 4

where we have to multiply the value designed by the operand b with 4 (the other operand) and to assign the result to the operand a. However, in most cases we deal with generic names: we don't know what the value is but where it is stored: we know its **address**. This is the reason we discuss about:

- 3-address machines;
- 2-address machines;
- 1-address machines;
- 0-address machines.

From now on, when we say we have an *n-address machine* we understand that *the maximum number of operands is n*.

Throughout this lecture we shall use the convention that the destination is the first operand in the instruction. This a commonly used convention though not generally accepted. It is consistent with the assignment statements in high level languages. The other used convention, listing the destination after the source operands, is coherent with our verbal description of operations.

3.2.1 Three-address machines

In a 3-address machine all three operands are explicit in each instruction. The general format of an instruction is:

```
operation dest, op1, op2
```

where:

- operation is the name of the operation to be performed;
- dest is the destination operand, the place where the result will be stored;
- op1 and op2 are the two source operands.

Thus the meaning of:

```
ADD r2, r1, r0
```

is to add the value stored in register r1, with the value stored in register r0, and put the result in the register r2.

Let’s see what addresses can specify and what are the implications for the hardware. In the example above all operands were held in registers and you may wonder why we discuss about addresses in this case: the reason is that registers can be seen as a special part of the memory, very fast and very close to the CPU; for historical reasons they have special names instead of being designated with some addresses.

Suppose you are designing an instruction set for a machine with 32 registers; then you need five bits to specify a register and, because the three operands may be different, 15 bits to specify the three registers that hold the three operands.

An instruction must also have a field to specify the operation to be performed, the opcode, probably a few bits depending of the number of instructions you want to have in the instruction set. We end up with an instruction that is 20 to 22 bits wide only for reasons of specifying the operands and the opcode. we shall see there are also other things to be included in a instruction, like an offset, a displacement, or an immediate value, with the possibility of having an even larger instruction size.

Should the machine be a 24 bit machine (24 is the first multiple of eight after 20-22) or not, and the answer is not necessarily:

- if we choose to have a 24 bit (or more) datapath then there is a definite advantage: the instruction is fetched at once, using a single memory access (assuming also that the data bus between the CPU and the memory has the same size as the datapath);

- if we settle for a cheap implementation, an 8 bit machine for instance, then the CPU must perform three memory accesses to memory only to get the whole instruction; then it can execute it. It is easy to see that in this case the performance both because number are processes in 8 bit chunks and because the instruction fetch takes so many clock cycles.

When all instructions (again we refer to the most common arithmetic/logic operations) specify only register operands then we say we have a **register-register** machine or a **load-store** machine; the load-store names comes from the fact that operands have to be loaded in registers from memory, and the result is stored in memory after the operation is performed.

In the following we'll take a different approach, we consider that all operands are in memory.

Example 3.3 UNDERSTANDING INSTRUCTIONS:

What is the meaning of the following instruction?

```
ADD   x, y, z
```

Answer:
Add the value of variable y (this is somewhere in the memory, we don't have to worry about, its address will be known after the translation is done), to the value of variable z (also in memory), and then store the result in the memory location corresponding the variable x.

Example 3.3 shows another way to process operands, when they all reside in the memory. If **all** instructions specify only memory operands then we say we have a **memory-memory** machine. We will next explore the implications for hardware of a memory-memory architecture.
In the case of a memory-memory machine the CPU knows it has to get two operands from memory before executing the operation, and has to store the result in memory. There are several ways to specify the address of an operand: this is the topic of addressing modes; for the following example we will assume a very simple way to specify the address of an operand: the absolute address of every operand is given in the instruction (this addressing mode is called *absolute*).

Example 3.4 MEMORY ACCESSES:

Addresses in an 8 bit machine are 16 bit wide. How many memory accesses are necessary to execute an instruction? Assume that the machine is memory-memory and operands are specified using absolute addresses.

Answer:
The instruction must specify three addresses, which means 3*2 = 6 bytes, plus an opcode (this is the first to be read by the CPU) which, for an 8 bit machine, will probably fit in one byte. Therefore the CPU has to read:

1 + 6 = 7 bytes

only to get the whole instruction. The source operands have to be read from memory and the result has to be stored into memory: this means 3 memory accesses. The instruction takes:

7 + 3 = 10

memory accesses to complete.

So far a register-register machine seems to be faster than a memory-memory machine, mainly because it takes less time to fetch the instruction and there are fewer accesses to the memory. The picture however is not complete: to work within registers operands must be brought in from the memory, and write out to the memory, which means extra instructions. On the other hand intermediate results can be kept in registers thus sparing memory accesses. We'll resume this discussion after introducing the addressing modes.

3.2.2 Two-address machines

We start with an example.

Example 3.5 STATEMENT IMLPEMENTATION:

Implement the high level statement:

```
a = a + b + a * c
```

on a 3-address machine; assume that variables a, b, c are in registers r1, r2, and r3 respectively.

Answer:

```
MUL r4, r1, r3    # use r4 as a temporary
ADD r1, r1, r2    # a + b in r1
ADD r1, r1, r4    # final result in r1
```

Note that if the values of b or c are no longer necessary, one of the registers r2 or r3 could be used as a temporary.

In this example two out of three instructions have only **two distinct addresses**; one of the operands is both a source operand and the destination. This situation occurs rather frequently such that you may think it is worth defining a **2-address machine**, as one in which instructions have only two addresses.

The general format of instructions is:

```
operation dest, op
```

where:

- operation is the name of the operation to be performed
- dest designates the name of one source operand and the name of the destination
- op is the name of the second source operand

Thus the meaning of an instruction like:

```
ADD r1, r2
```

is to add the values stored in the registers r1 and r2, and to store the result in r1.

There is an advantage in having two-address instructions as compared with three-address instructions, namely that instruction are shorter, which is important when preserving memory is at big price; moreover shorter instructions might be fetched faster (in the case instructions are wider than the datapath and multiple accesses to memory are required). There is a drawback however with two-address instructions: one of the source operands is destroyed; as a result extra moves are sometimes necessary to preserve the operands that will be needed later.

Example 3.6 STATEMENT IMLPEMENTATION:

Show how to implement the high level statement

```
a = a + b + a * c
```

on a 3-address machine and then on a 2-address machine. Both machines are 8 bit register-register machines with 32 general purpose registers and a 16 bit addresses. The values of variables a, b, and c are stored in r1, r2 and r3 respectively. In any case calculate the number of clock cycles necessary if every memory access takes two clock cycles and the execution phase of an instruction takes one clock cycle.

Answer:
For the 3-address machine:

```
MUL r4, r1, r3    # 3 * 2 + 1 clock cycles
ADD r1, r1, r2    # 3 * 2 + 1
ADD r1, r1, r4    # 3 * 2 + 1
```

This sequence requires 21 clock cycles to complete; each instruction has a fetch phase that takes three (3 bytes/instruction) times two clock cycles (2 clock cycles per memory access), plus an execution phase which is one clock cycle long.For the 2-address machine:

```
MOV r4, r1        # 2 * 2 + 1 clock cycles
MUL r4, r3        # 2 * 2 + 1
ADD r1, r2        # 2 * 2 + 1
ADD r1, r4        # 2 * 2 + 1
```

The sequence requires 20 clock cycles to complete; it is slightly faster than the implementation of the same statement on the 3-address machine.
The two address machine requires 10 bits (5 + 5) to encode the two operands and the example assumes an instruction is 16 bit wide.

The above example has introduced a new two operands instruction:

```
MOV dest, op
```

which transfers (moves) the value stored at address op to address dest.

Even if the MOV instruction is typical two operands instruction there is no reason to believe it only belongs to 2-address machines: a 3-address machine is one in which instructions specify up to 3 operands, not a machine in which all instructions have precisely 3 operands.

In this section we discussed about a register-register, 2-address machine. Nothing can stop us thinking about a 2-address, memory-memory machine, or even about a register-memory one.

3.2.3 One address machine (Accumulator machines)

In a 1-address machine the accumulator is implicitly both a source operand and the destination of the operation. The instruction has only to specify the second source operand. The format of an instruction is:

```
operation op
```

where:

- operation is the name of the operation to be performed
- op is a source or a destination operand. Example of source or destination operand is the accumulator (in the case of a store op denotes the destination).

Thus the meaning of:

```
ADD a
```

is to add the value of variable a to the content of the accumulator, and to leave the result in the accumulator.The accumulator is a register which has a special position in hardware and in software. Instructions are very simple and the hardware is also very simple. Figure 2.2 is an exemplification of an accumulator machine.

Example 3.7 STATEMENT IMPLEMENTATION:

Show how to implement the statement

```
a = a + b + a * c
```

using an accumulator machine.

Answer:

```
LOAD a      # bring the value of a in accumulator
MUL  c      # a * c
ADD  b      # a * c + b
ADD  a      # a * c + b + a
STO  a      # store the final result in memory
```

Due to its simplicity, only one operand has to be explicitly specified, accumulator machines present compact instruction sets. The problem is that the accumulator is the only temporary storage: memory traffic is the highest for accumulator machines compared with other approaches.

3.2.4 Zero-address machines (stack machines)

How is it possible to have a machine without explicit operands instructions? This is possible if we know where the operands are, and where is the result to be stored. In other words all operands are implicit.

A stack is a memory (sometimes called LIFO = Last In First Out) defined by two operations PUSH and POP: PUSH moves a new item from the memory into the stack (you don't have to care where), while POP gets the last item that was pushed into the stack. The formats of operations on a stack machine are:

```
operation
PUSH op
POP  op
```

where:

- operation indicates the name of the operation to be performed operation always acts on the value(s) at top of the stack

- op is the address in the main memory where the value to be pushed/popped is located.

A stack machine has two memories: an unstructured one, we call it the main memory, where instructions and data are stored, and a structured one, the **stack** where access is allowed only through predefined operations (PUSH/POP). It is worth mentioning here that a stack has usually a very fast part where the top of the stack is located, and which is usually inside CPU, while the large body of the stack is outside the CPU (possibly extensible to disk) and, hence, slower. One may envision the fast part of the stack (inside CPU) as the registers of the machine; as a matter of fact, a stack machine does not have explicit registers.

Example 3.8 STATEMENT IMLPEMENTATION:

Show how to implement the statement

```
a = a + b + a * c
```

using a stack machine.

Answer:

```
PUSH a      # push the value of a;
PUSH c      # push the value of c
MUL         # multiply the two values on top of the stack
PUSH b
ADD
PUSH a
ADD
POP  a      # store the result back in memory at the address
              where a is located.
```

Whenever an operation is performed, the source operands are popped from the stack, the operation is performed, and the result is pushed into the stack.

Example 3.9 STATEMENT IMLPEMENTATION:

Show the content of the stack while implementing the statement:

```
a = a + b + a * c
```

ANSWER:

PUSH a	PUSH c	MUL	PUSH b	ADD	PUSH a	ADD	POP a
	c		b		a		
a	a	a*c	a*c	b+a*c	b+a*c	a+b+a*c	

The stack machine has the most compact encoded instructions possible.

To conclude this section let's see how fast is an expression evaluated on a stack machine as compared with other machines.

Example 3.10 CALCULATION OF CLOCK CYCLES:

Compute the number of clock cycles necessary to evaluate the statement:

```
a = a + b + a * c;
```

on an 8 bit stack machine, with 16 bit addresses. Assume that every memory access takes two clock cycles and that the execution phase of any instruction take only one clock cycle; assume also that addresses are absolute.

Answer:

```
PUSH a      # 2 + 2 * 2 + 2 clock cycles
PUSH c      # 2 + 2 * 2 + 2
MUL         # 2 + 1
PUSH b      # 2 + 2 * 2 + 2
ADD         # 2+ 1
PUSH a      # 2 + 2 * 2 + 2
ADD         # 2 + 1
POP  a      # 2 + 2 * 2 + 2
```

The above sequence of code completes in 49 clock cycles. Every PUSH or POP takes 2 clock cycles to read the opcode (one byte), plus 2 * 2 clock cycles to read the address (2 bytes) of the operand, plus 2 clock cycles to read/store the operand.

It would be unfair to directly compare the performance of the stack machine in example 3.10 with the performance of machine in the previous examples as long as the stack machine has to bring the operands from memory while in the other cases it was assumed that the operands are in registers. The stack has the disadvantage that, by definition, it cannot be randomly accessed thus making difficult to generate *efficient* code; note however that the easiest way to generate code in a compiler is for a stack machine.

3.3 Register or memory?

In classifying architectures we used the number of operands the most common arithmetic instructions specify. As we saw the 3-address and 2-address machines may have operands in registers, memory or both. A question may be naturally asked: which is the best way to design the instruction set? This is to say: should all instructions be register-register or maybe should they all be memory-memory; what happens if we mix registers and memory in specifying operands?

Example 3.11 CLOCK CYCLES AND MEMORY TRAFFIC:

We have two 32 bit machines, a register-register and a memory-memory one. Addresses are 32 bit wide and the register-register machine has 32 general purpose registers. A memory access takes two clock cycles and an arithmetic operation executes in two clock cycles (the execution phase). Addresses of operands are absolute. Show how to implement the statement:
a = b * c + a * d;
and compare the number of clock cycles, and the memory traffic for the two machines. Variables a, b, c, d reside in memory and no one may be destroyed but a. Assume also that instructions are 32 bit wide or multiples of 32 bit.

Answer:

		Words per instruction	Memory accesses	Clock cycles
LOAD	r1, b	2	3	6
LOAD	r2, c	2	3	6
MUL	r3, r1, r2	1	1	3
LOAD	r1, a	2	3	6
LOAD	r2, d	2	3	6
MUL	r1, r1, r2	1	1	3
ADD	r1, r1, r3	1	1	3
STORE	a, r1	2	3	6
Total		13	18	39

For the memory-memory machine:

		Words per instruction	Memory accesses	Clock cycles
MUL	temp, b, c	4	4+3	15
MUL	a, a, d	4	4+3	15
ADD	a, a, temp	4	4+3	15
Total		12	21	45

For the memory-memory machine every instruction require 7 memory accesses: one to get the opcode, plus 3 to get the three addresses of the operands, plus other 3 to access the operands. The code for the memory-memory machine is shorter than the code for the register-register machine, 12 words as compared with 13, but the memory traffic is higher, as well as the execution time.

Even though in the above example we considered two different machines, it is quite common to have hardware that supports both models. This is the case with very successful machines like IBM-360 or DEC VAX; moreover these machines also support the register-memory model.

The most valuable advantage of registers is their use in computing expression values and in storing variables. When variables are allocated to registers, the memory traffic is lower, the speed is higher because registers are faster than memory and code length decreases (since a register can be named with fewer bits than a memory location). If the number of registers is sufficient, then local variables will be loaded into registers when the program enters a new scope; the available registers will be used as temporaries, i.e. they will be used for expression evaluation.

How many registers should a machine have? If their number is too small then the compiler will reserve all for expression evaluation and variables will be kept in memory thus decreasing the effectiveness of a register-register machine. A too large number of registers, on the other hand, may mean wasted resources that could be used otherwise for other purposes.

Most microprocessors from the first generation (8 bit like Intel 8080, or Motorola 6800) had 8 general purpose registers. A very successful 16 bit microprocessor, Intel 8086, had 14 registers though most of them were reserved for special purposes. Almost all 32 bit microprocessors on the market today have 32 integer registers, and many have special registers for floating point operations.

32 registers seem to be sufficient even for a 64 bit architecture; at least this is the point of view of the team who designed the ALPHA chip.

Let's now summarize some of the basic concepts we have introduced so far. The basic criteria of differentiating architectures was the number of operands an instruction can have. Operands may be named explicitly or implicitly:

- **stack architectures**: all operands are implicitly on the stack
- **accumulator architectures**: one source operand and the destination are implicitly the accumulator; the second source operand has to be explicitly named
- **general purpose register (GPR) architectures** have only

explicit operands, either registers or memory locations. GPR architectures dominate the market at this moment, basically for two reasons: first registers are faster than memory and second, compilers use more efficient registers than other forms of temporary storage like accumulators or stacks. The main characteristics that divide GPRs are:

- th**e number of operands** a typical arithmetic operation has we discussed about 2-address machines (one of the operands is both source and destination), and about three address-machines;

- number of operands that may be **memory addresses** in ALU operation. This number may vary from none to three.

Using the two parameters we have to differentiate GPR architectures, there are seven possible combinations in the table below:

Number of operands	Number of memory addresses	Examples
2	0	IBM
	1	PDP 10, Motorola 68000, IBM-360
	2	PDP 11, IBM-360, National 32x32, VAX
3	0	SPARC, MIPS, HP-PA, Intel 860, Motorola 88000
	1	IBM-360
	2	-
	3	VAX

Please observe that IBM-360 appears in several positions in the table, due to the fact that the architecture supports multiple formats. The same is true for the VAX architecture.

Three combinations classify most of the existing machines:

- **register-register** machines which are also called load-store. They are defined by 2-0 or 3-0 (operands-number of memory addresses); in other words all operands are located in registers

- **register-memory** machines: defined by 2-1 (operands-number of memory addresses); one of the operands may be a memory address;

- **memory-memory** machines: 2-2 or 3-3; all operands are memory addresses.

Here are some of the advantages and disadvantages of the above alternatives:

register-register machines

Advantages:
- simple instruction encoding (fixed length)
- simple compiler code generation
- instruction may execute in the same number of clock cycles (for a carefully designed instruction set)
- lower memory traffic as compared with other architectures.

Disadvantages:
- higher instruction count than in architectures with memory operands
- the fixed format wastes space in the case of simple instructions that do not specify any register (NOP, RETURN etc.)

register-memory machines

Advantages:
- more compact instructions than in register-register machines
- in many cases loads are spared (when an operand is in register and the other one is in memory)

Disadvantages:
- operands are not equivalent: in an arithmetic or logic operation one of the source operands is destroyed (this complicates the code generation in the compiler).

memory-memory machines

Advantages:
- compact code
- easy code generation

Disadvantages:
- different length instructions
- different running times for instructions
- high memory traffic.

3.4 Problems in instruction set design

The instruction set is the collection of operations that define how data is transformed/moved in the machine. An architecture has an unique set of operations and addressing modes that form the **instruction set**.

The main problems the designer must solve in setting up an instruction set for an architecture are:

- which are the operations the instruction set will implement;
- relationship between operations and addressing modes;
- relationship between operations and data representation.

3.4.1 Operations in the instruction set

We obviously need **arithmetic and logic operations** as the purpose of most applications is to perform mathematical computation. We also need **data transfer** operations as data has to be moved from memory to CPU, from CPU to memory, between registers or between memory locations.

The instruction set must provide some instructions for the **control flow** of programs; we need, in other words, instructions that allow us to construct loops or to skip sections of code if some condition happens, to call subroutines and to return from them.

Arithmetic and logic instructions

Integer operations:

- add and subtract are a must and they are provided by all instruction sets. At this point it must be very clear how integers are represented in the machine: unsigned or signed and, in the latter case what signed representation is used? The most frequent used representation for signed integers is the two's complement. Some architectures provide separate instructions for unsigned integers and for signed integers.

- multiply and divide: the early machines had no hardware support for these operations. Almost all new integrated CPUs, introduced since the early 80s provide hardware for multiplication and division. Again the integer representation must be very clear because it affects the hardware design. Integer multiplication and division are complicated by the results they generate;

multiplication of two n bit integers may result in a 2*n bit number. If all registers are n bit wide, then some provisions must be made in hardware to accommodate the result.

- compare instructions: compare two integers and set either a condition code (a bit in a special register called the Status Register), or a register (in the new architectures); EQ, NE, GT, LT, GE, LE may be considered for inclusion in the instruction set. Set operations are used in conjunction with branches, so they must be considered together at the design time.

- others: like the remainder of a division, or increment/decrement register, add with carry, subtract with borrow etc.; most of these instructions can be found in the instruction sets of earlier architectures (8 and 16 bit).

Floating point operations:

While the IEEE 754 standard is currently adopted by all new designs, there are other formats still in use, like those in the IBM 360/370 or in the DEC's machines.

- add, subtract, multiply, divide: are provided sometimes in an additional instruction set; most of the new architectures have special registers to hold floating point numbers.

- compare instructions: compare two floating point numbers and set either a condition code or a register: EQ, NE, GT, LT, GE, LE may be considered.

- convert instructions: convert numbers from integer to float and vice-versa, conversion between single precision and double precision representations might be considered.

Logical instructions

It is not necessary that all possible logical operations are provided, but care must be taken that the instruction set provides the minimum necessary such that the operations that are not implemented can be simulated.

Decimal operations

Decimal add, subtraction, multiply, divide are useful for machines running business applications (usually written in COBOL); they are sometimes primitives like in IBM-360 or VAX, but in most cases they are simulated using other instructions.

String operations

String move, string compare, string search, may be primitive operations as in IBM-360 or VAX, or can be simulated using simpler instructions.

Shifts

Shift left, right, arithmetic/logic are useful in manipulating bit strings, in implementing multiplication /division when not provided by hardware; all new architectures provide shifts in the instruction set.

- arithmetic shift: whenever the binary configuration in a register is shifted towards the least significant bit (shift right) the most significant positions are filled with the value of the sign bit (the most significant bit in register before shifting);

- logical shift: positions emptied by shift are filled with zeros.

Example 3.12 LOGICAL AND ARITHMETIC SHIFTS:

The binary configuration in a register is:
11010101
Show the register's configuration after the register is shifted one and two positions, logical and arithmetic.

Answer:

1 1 0 1 0 1 0 1	The initial configuration
1 1 1 0 1 0 1 0	arithmetic right one bit
1 1 1 1 0 1 0 1	arithmetic right two bits
0 1 1 0 1 0 1 0	logical right one bit
0 0 1 1 0 1 0 1	logical right two bits
1 0 1 0 1 0 1 **0**	left one bit
0 1 0 1 0 1 **0 0**	left two bits

Note that it does not make sense to discuss about left arithmetic shifts.

Data transfer instructions

As long as the CPUs were 8 bit wide the problem of transfers was simple: loads and stores had to move *byte*s from memory to CPU, from CPU to memory or between registers. With a 32 bit architecture we must consider the following transfer possibilities:

- **byte**s (8 bits);
- **half-word**s (16 bits);
- **word**s (32 bits);
- **double word**s (64 bits).

While allowing all transfers provides maximum flexibility, it also poses some hardware complications that may slow down the clock rate (this will be discussed again in chapter 4).

If transfers of items narrower than a word (for the today 32 bit architectures) are included in the instruction set, then two possibilities are to be considered for **loads:**

- **unsigned:** data is loaded in a register at the least significant positions while the most significant positions are filled with zeros;

- **signed:** data is loaded into a register at the least significant positions while the most significant positions in the register are filed with the sign of the data being loaded (i.e. the MSB of data).

Example 3.13 EFFECT OF INSTRUCTIONS ON REGISTERS:

Show the content of the register r1 after executing the instructions:

```
LB    r1,   0xa5        # load bite (signed)
```
and
```
LBU   r1,   0xa5        # load byte unsigned
```

All registers are 16 bit wide.

Answer:

```
  MSB                        LSB

 1111  1111  1010  0101       after executing LB  r1, 0xa5


 0000  0000  1010  0101       after executing LBU r1, 0xa5
```

Moves

Allow for transfer of data between registers; if the architecture provides floating point registers, then the instruction set should include also the possibility to transfer between the two sets of registers (integers & floats).

Control flow instructions

In the followings we shall use the names:

- **branch:** when the change in control is conditional
- **jump:** when the change in control is unconditional (like a goto).

The four categories of control flow instructions are:

- **branches**
- **jumps**
- **subroutine calls**
- **subroutine returns**

The destination address of a branch, jump or call can always be specified in the instruction; this can be done because the *targe*t (i.e. the instruction where control must be transferred) is known at compile time. In the case of a return the target is not known at the compile time; a subroutine can be called from different points in a program.

Specifying the destination

There are two basic ways to specify the target:

- use the **absolute address** of the target. This has two main drawbacks, first it requires the full address of the target to be provided in the instruction which is impossible for fixed length instructions, and second, programs using absolute addresses are difficult to relocate.

- use **PC-relative** computation of the target. This is the most used way to specify the destination in control-flow; a displacement is added to the program counter (PC) to get the target address. The displacement is usually represented as a two's complement integer, thus allowing to specify targets that are after the current instruction (positive displacement), or before the present instruction (negative displacement). The main advantages in using the PC-relative way to compute the target address are:

    - **fewer bits** necessary to specify the target. Targets are usually not very far away from the present instruction, so that it is not worth specifying the target as an absolute address;

- **position independence:** because addressing is PC-relative, the code can run independent of where it is loaded in memory. The linking and loading of programs is thus simplified.

What happens with returns

For returns the target address is not known at the compile time; the target must be somehow specified dynamically so that it can change at run time:

- use the **run time stack** to hold the return address: the address of the next address to be executed after a call instruction (at the return from the subroutine) is pushed into the run time stack when the subroutine is called; at the end of the subroutine the address that was saved in the stack is popped into the PC;

- use a **register** that contains the target address (of course the register must be wide enough to can accommodate an address); most of the new architectures use this mechanism for the returns; a register from the set of general purpose registers is dedicated to this purpose.

A major concern in the design of an instruction set is the size of the displacement; if the size is too small then we must very often resort to absolute addresses which mean larger instructions. Too much space for displacement may be detrimental for other useful functions.

Exercises

3.1 What is the value of variables x and y after the following operations are performed:

x = 3 - 2 - 1;
y = 12/6/3;

If you are unsure about the result, then write a small program in the language you like, and see which are the rules for the associativity.

3.2 Addresses in a 32 bit machine are 32 bit wide. How many memory accesses are necessary to execute an instruction? Assume that the machine is memory-memory and operands are specified using absolute addresses.

3.3 What was the need for the early microprocessors (8 bit) to provide instructions like add with carry or subtract with borrow? Modern 32 bit CPUs do not provide such instructions.