Materials

W7 Współczesna architektury procesorów i zestawy instrukcji, część 1 i 2

Omawiane zagadnienia

- addressing modes; the various types of addressing modes common in instruction sets
- number of addresse; issues and trade-offs involved in number of addresses contained in each instruction
- instruction formats; issues and trade-offs involved in designing an instruction format

Addressing modes

The various types of addressing modes common in instruction sets.

The address field or fields in a typical instruction format are relatively small. Probably you would like to be able to reference as large range of locations in main memory as it is only possible.

To achieve this objective, a variety of addressing techniques has been employed. They all involve some trade-off between address range and/or addressing flexibility, on the one hand, and the number of memory references in the instruction and/or the complexity of address calculation, on the other. The most common addressing techniques, or modes, are:

- immediate,
- direct (memory),
- indirect (memory),
- register (direct),
- register indirect,
- base plus displacement,
- stack.

Immediate

Direct

Indirect

Register

Register indirect

Displacement

Stack

Number of addresses

Issues and trade-offs involved in number of addresses contained in each instruction.

One of the traditional ways of describing processor architecture is in terms of the number of addresses contained in each instruction. This dimension has become less significant with the increasing complexity of processor design. Nevertheless, it is useful at this point to draw and analyze this distinction. What is the maximum number of addresses one might need in an instruction?

Evidently, arithmetic and logic instructions will require the most operands. Virtually all arithmetic and logic operations are either unary (one source operand) or binary (two source operands). Thus, you need a maximum of two addresses to reference *source operands*. The result of an operation must be stored, suggesting a third address, which defines a *destination operand*.

This reasoning leads you to instruction with multiple operands (multiple addresses) contained in each. However there is an opposite possibility to reduce the number of operand in favour to use predefined, implicite, locations of opperands. Is it good? It depends.

Now you will see all the variants calculating the same value:

$$v = \frac{a + b}{c \cdot (d - e)}$$

Three-address instructions case

```
ADD a, b, u   ;   u := a + b
SUB d, e, l   ;   l := d - e
MUL c, l, l   ;   l := c * l
DIV u, l, v   ;   v := u / l
```

As you can see in this case you need only four instructions, corresponding exactly to the number of operation required to complete computing given formula. So in terms of the number of instructions, this is the most effective case. At the same time each instruction is the most complex in terms of the number of location (memory or registers) refference. In total

you have 12 references to operands. If all of them are located in memory you need time to get them from or move them to their locations. If you want to put them in internal CPU registers, there must be a decent number of general purposes registers to use during calculations and do not "block" other, preceding or subsequent, operations:

```
ADD  a,  b, r1   ;   r1 := a + b
SUB  d,  e, r2   ;   r2 := d - e
MUL  c, r2, r2   ;   r2 := c * r2
DIV  u, r1, r2   ;    v := r1 / r2
```

In this case you have six (6) references to memory and six (6) references to registers.

Finally, to be able to handle so many operands in each instruction, you need much wider instruction format. *Handle* means in this case coding it reserving a predefined number of bits which makes each instruction longer. Longer instruction requires either more time to get them in fetch CPU working phase (for example you may need to make two or three subsequent memory references to fetch one instruction), or you need wider data bus which complicates system design and increase overall costs.

One-address instructions case

This case is totally opposite to previously discussed. Now you provide as little information as it is possible. You have to keep references to some memory locations -- one for each instruction. If instruction requires second opperand by its formal definition (for example `MUL` or `ADD`) or the way it works (`CPA` -- copy from *source* to *destination*) it uses default, predefined location known as **accumulator**, `acc`:

```
CPA d  ; acc := d           ; acc := d
SUB e  ; acc := acc - e     ; acc := d - e
MUL c  ; acc := acc * c     ; acc := (d - e) * c
STO v  ; v := acc           ; v := acc
CPA a  ; acc := a           ; acc := a
ADD b  ; acc := acc + b     ; acc := a + b
DIV v  ; acc := acc / v     ; acc := (a + b) / ((d - e) * c)
STO v  ; v := acc           ; v := acc
```

Two-address instructions case

This is something in between three-address and one-address instruction case. You can think about this as addressing with multiple accumulators. Because there are multiple "accumulators" you have to specify them explicitly. Those accumulators are **general purpose registers**, for example `r1` or `r2`:

```
LOAD r1, a    ; r1 := a
ADD  r1, b    ; r1 := a + b
LOAD r2, d    ; r2 := d
SUB  r2, e    ; r2 := d - e
MUL  r2, c    ; r2 := (d - e) * c
DIV  r1, r2   ; r1 := r1 / r2
MOV   v, r1   ;  v := r1
```

Zero-address instructions case

It is, in fact, possible to make processing with zero addresses for some instructions. In such a case you use implicite "addresses" with the help of stack.

```
PUSH e   ; e is on the top of the stack.
PUSH d   ; d is on the top of the stack.
SUB      ; Pop two values from a stack (d and e), subtract them
         ; and push the result (d-e) on the stack.
PUSH c   ; c is on the top of the stack.
MUL      ; Pop two values from a stack ((d-e) and c), multiply them
         ; and push the result (d-e)*e on the stack.
PUSH b   ; b is on the top of the stack.
PUSH a   ; a is on the top of the stack.
ADD      ; Pop two values from a stack (a and b), add them
         ; and push the result (a+b) on the stack.
DIV      ; Pop two values from a stack ((a+b) and ((d-e)*e)),
         ; divide them and push the result
         ; (a+b)/((d-e)*e) on the stack.
```

This type of processing (computing) is related to Reverse Polish notation (RPN) where formula:

$$v = \frac{a + b}{c \cdot (d - e)}$$

is written as:

```
e d - c * b a + /
```

To calculate formula given in RPN form you use stack:

```
Operation | Stack contents
          | (grows from left to right)
----------+-----------------------------------
NULL      | empty
----------+-----------------+
PUSH e    | e               |
----------+-----------------+-----+
PUSH d    | e               | d   |
----------+-----------------+-----+
          | (pop two values and use as operands for SUB)
SUB       +-----------------+
          | d-e             |
----------+-----------------+-----+
PUSH c    | d-e             | c   |
----------+-----------------+-----+
          | (pop two values and use as operands for MUL)
MUL       +-----------------+
          | c*(d-e)         |
----------+-----------------+-----+
PUSH b    | c*(d-e)         | b   |
----------+-----------------+-----+-----+
PUSH a    | c*(d-e)         | b   | a   |
----------+-----------------+-----+-----+
          | c*(d-e)         | (pop two values and use as operands for AD
ADD       +-----------------+-----+
          | c*(d-e)         | a+b |
----------+-----------------+-----+
          | (pop two values and use as operands for DIV)
DIV       +-----------------+
          | (a+b)/(c*(d-e)) |
----------+-----------------+
```

Instruction formats

Issues and trade-offs involved in designing an instruction format.

The number of addresses you decide to use in each instruction has great influence on **instruction format** which defines the layout of the bits of an instruction, in terms of its constituent fields.

An instruction format must include at least an opcode and, implicitly or explicitly, zero or more operands. Each explicit operand is referenced using one of the **addressing modes** described in next section. The format must, implicitly or explicitly, indicate the addressing

mode for each operand. For most instruction sets, more than one instruction format is used.

Two different tendencies collide when you design instruction set. From one side there are circuit designers who prefer simpler instructions in terms of its variety and number of fields present in each. Every kind of simplicity on instruction level is translated into simpler circuit which in turn may result on faster and more economic execution. On the other side there are programmers who want more opcodes, more operands, more addressing modes, and greater address range.

More opcodes and more operands make programmers life easier, because they can written shorter code to accomplish given tasks. Compare length of code (in terms of instructions) you obtain in three-address instructions case of preceding section with one-address instructions case. If you consider zero-address instructions case you have to make processing in very specific order (RPN for example) which limits your flexibility and requires additional knowledge and the extra effort you have to put into it.

More addressing modes give the programmer greater flexibility in specifying memory locations and allows to address larger memory ranges. Moreover it makes possible to implementing certain functions, such as table manipulations and multiway branching.

Thus the most basic design issue to be faced is the **instruction format length**. This decision affects, and is affected by, various system components like memory size and organization, bus structure, processor complexity, and overall processing speed. This decision determines the richness and flexibility of the machine as seen by the assembly language programmer.

The most obvious trade-off here is between the desire for a powerful instruction repertoire and a need to save space required to put into it your code.

Of course all of aforementioned elements (opcodes, operands, addressing modes, address range) have to be somehow encoded on bits but greater complexity (which is an effect of greater flexibility) brings longer instruction lengths. The longer instruction you have the more memory you use to store exactly the same code (in terms of sequence of instruction you use). A 64-bit instruction occupies twice the space of a 32-bit instruction but is arguable if they are twice as useful.

On the other side, longer instruction means more "powerful" instruction which in turn may result in less constructions in total. Recall three-address instructions case and zero-address instructions case. In the first case there are four instructions with three operands each. If opcode and operands are encoded on one byte, you obtain 16 bytes (4 opcodes + 12 operands references). In the second case there are nine instructions and five operands in total. This way you obtain 14 bytes. As you can see, you obtain longer code in terms of the number of instruction but shorter in terms of memory usage. Of course this is not a rule, you cannot say that always/newer longer/shorter instructions are more economic, and the devil is

in the details.

When you talk abut length you cannot forget low level details such as "portions" in which you will transfer instruction(s). Either the instruction length should be equal to the memory native transfer length (data bus width) or their integer multiple should fit to this length (either in the form: *integer multiple of instructions fit to this lenght* or in the form: *one instruction fits to length multipled by an integer*) or one should be a multiple of the other if you decide to use a variable-length instructions (see below). This is necessary to optimize time required to complete fetch cycle -- in one fetch cycle you should get an integral number of instructions, at least one or more. If you need more memory acces to "complete" instruction, it will take a toll on performance.

Because the processor does not know the length of the next instruction to be fetched, a typical strategy is to fetch every time a number of bytes or words equal to at least the longest possible instruction. This means that sometimes multiple instructions are fetched and this is a good approach allowing effective cache memory usage.

To offer programmers more elasticity you may consider a **variable-length instructions**. This approach allows you to design a large repertoire of instruction formats with different opcode lengths, very flexible addressing and rich combinations of registers and memory references using different addressing modes. As an instruction designer you will be happy but from processor designer's perspective it heavily affects and complicates all the circuits present in the CPU. Variable-length versus fixed-length approach materialize in two different CPU architectures: CISC (Complex Instruction Set Computing) and RISC (Reduced Instruction Set Computing) -- both will be discussed later.

Note: CISC vs. RISC in short

CISC (Complex Instruction Set Computers)

Characteristics:

- Large number of instructions (100-200 instructions).
- Instructions are very often complex and specialized.
- Low optimization –- instructions require a large number of processor cycles to be execute.
- Large number of addressing modes (from 5 to 20).
- A large number of instructions can directly refer to memory.
- Lower processor clock speed than in the RISC architecture.
- Several low-level operations can be performed in one instruction.
- Microprogrammable control unit

RISC (Reduced Instruction Set Computers)

Characteristics:

- Reduced number of instructions (several dozen instructions).
- Instructions are executed in one clock cycle.
- Reduction of addressing modes.
- Easy to decode, fixed length (32 bit) instruction formats.
- Restriction of communication between memory and processor. The LOAD (load from memory) and STORE (save to memory) instructions are used to transfer data between memory and registers.
- The remaining instructions operate only on the processor's internal registers.
- Increasing the number of registers reduces the number of memory references.
- The control unit is implemented in a system.

As you may conclude, the system designer is faced with a lots of factors to consider and balance. It is really hard to say how critical the various choices are. You should have two almost identical system differs in one factor to be able to judge its importance. There are two, unsolvable, problems. First one is that no one wants to spend a lot of many to make system only to compare it with other solutions. Second is that most factors are not independent. If you change one of them, your choice influence other parts. I would say that if you make a decent, fair and resonable assumption, you will obtain comparable overall performance. Saying *decent, fair and resonable* I have in mind that, for example, for 32-bit length instruction format you will not use system with 8-bit width data bus.

Bibliography

1. William Stallings, *Computer Organization and Architecture*, Pearson, 11th edition (2019)
2. Jim Ledin, *Modern Computer Architecture and Organization*, Packt, April 2020