Second part of topic 7

„Implementation of control flow statements in assembly (conditional- and unconditional jump, branching, loops, subroutine call)”

During operation, the processors execute the instructions of the machine code program loaded into memory. During program execution, the program counter register of the CPU (PC, EIP) always contains the memory address of the next instruction to be executed. This register in the control unit is an important tool for fetch-execute loops. In most cases, when an instruction is executed, the value of the program counter register is increased with the length of that instruction (sequential execution).

However, there are cases (branching, looping, subroutine calling, etc.) where after the current instruction, not the consecutive instruction in the memory needs to be executed, but an instruction that is somewhere else in the RAM. In this case, the program counter is not simply "incremented", but it is overwritten with another memory address specified in the program. This can be done by the programmer with control transfer instructions, thus creating different control structures.

At the level of assembly programming, conditional and unconditional instructions can be used to directly change the value of the program counter, and control transfer instructions of high-level programming languages (for example in C: `if-else`, `switch`, `while`, `for`, `break`, `continue`, `goto`, `return`) are also translated to these assembly instructions. (*The assembly instructions below are based on the Intel syntax x86 instruction set architecture.*)

Unconditional jump

In this case, the program counter is always overwritten by a specified value during the execution of the given instruction. This can be a value defined in the program as an operand or, for example, a value previously placed into the system stack. The most important instructions are:

- `jmp`
  The address specified as the operand is always saved into the program counter. Of course, the operand may be specified as a label in assembly, but it already refers to a (relative or absolute) memory address in the machine code. This statement is behind the C language `goto`, `break`, and `continue` statements.
- `call`
  Subprogram invocation instruction. Its operand is the memory address of the first instruction of the procedure or function to be called, so control is transferred to the called program unit. Before the program counter is overwritten with this address, its current value is saved to the top of the stack as a return address.
- `ret`
  Used to return from a subroutine. During execution, the value at the top of the stack

(the return address saved at the time of the call) is entered into the program counter register, i.e., the next instruction to be executed will be the instruction in memory after the last calling instruction. This is used by the C compiler when implementing the `return` statement.

- Other instructions of this category are `leave`, `int` or `iret` assembly instructions.

Conditional jump

During the execution of a given jump instruction, the system decides how the content of the program counter register changes. This is determined by the state of the processor as a kind of condition, therefore the status register (SR, FLAGS) plays an essential role during execution. Either the contents of the program counter register are "incremented" in the traditional sense (switched to the instruction at the next memory address) or overwritten with a memory address specified as an operand (i.e., program execution continues somewhere else).

- `jz/jc/jo/js`
  The second letter of the instruction mnemonic defines the condition, which depends on the current value of the given bit of the status register (ZF, CF, OF, SF, respectively). If the given flag bit is set (i.e., its value is 1), the address specified as the operand is loaded into the program counter (jump). Otherwise, the operand is not used and the contents of the program counter register increase with the length of the conditional jump instruction.
- `jnz/jnc/jno/jns`
  These instructions work similarly to the previous ones, with the exception that the conditions are negated, i.e., the value of the specific flag bit is 0 resulting in a jump.

There are also conditional jump statements that are combinations of the former (status register bit based) statements or simple alias names. The `cmp` comparative assembly instruction changes the status register's zero (ZF), carry (CF), overflow (OF), and sign (SF) bits based on the relationship between the operand values. Thus, the following assembly mnemonics can also be used for easier code readability:

- `je/jne`
  For a description of necessary jumps in case of equality or inequality (instead of `jz/jnz`).
- `jl/jle/jg/jge`
  For signed values to manage the $<, \leq, >, \geq$ cases.
- `jb/jbe/ja/jae`
  For unsigned values to manage the $<, \leq, >, \geq$ cases.

Often the above conditional statements are collectively referred to as `jcc`.

Special conditional jump instructions are `loop`, `loope`, and `loopne`, which are used for a kind of loop organization, where the implementation of the control transfer depends (also) on the current value of the general-purpose register ECX.

Branching

In high-level programming languages, you can use several branching structures (for example, `if`, `if-else`, and `switch` structures in C). At the assembly level, these should be implemented with the conditional and unconditional jump instructions discussed above.

The "do it or skip it" operation of an `if` branch without an `else` branch can be implemented using a single unconditional control transfer instruction. After the instructions that implement the condition evaluation, we need to write a conditional jump instruction. This is followed by instructions that must be executed optionally (i.e., instructions in the if branch). Finally, all other statements (required in all cases), the first of them must have a label to be specified as the operand of the conditional jump statement. The `jcc` statement must specify by the negation of the condition, as it specifies when to skip the optional block contrary to the high-level languages, where we specify when to execute it.

For the full if-else structure, we first write condition evaluation steps in assembly, then a properly selected `jcc` statement, and then the "true" block. The instructions for the "false" block will only be written after that. A label must be placed at the beginning of the latter, which will be the operand of the `jcc` statement. An unconditional jump instruction (`jmp`) must be placed between the instructions of the two branches (i.e., at the end of the "true" branch), the operand of this will be the label of the first instruction after the "false" branch. This prevents the control from flowing to the "false" branch after the "true" branch is executed. So, you need to specify a `jcc` and a `jmp` statement and two labels.

Loops

I would only present conditional loops. In C, `for` and `while` are preconditional loops (there is no difference between them at all in assembly), while the `do-while` structure is a postconditional loop.

For optimization reasons, during the programming of the preconditional assembly loop, we write the instructions of the trunk first and only then come the steps of evaluating the condition, as well as a `jcc` instruction. The operand of this conditional transfer instruction will be the label of the first instruction in the trunk. An unconditional jump instruction is placed at the very beginning of the entire loop structure, the operand of which must be the label of the first instruction of the condition evaluation. So, the execution starts with the condition evaluation, and we can even get an empty loop.

The assembly code of a postconditional loop differs from the precondition only in that it does not include the `jmp` statement, so that the statements in the body are executed at least once.

Subroutine call

The `call` subroutine invocation instruction and the `ret` instruction to return from the call have been presented previously. Strictly, control transfer in the narrow sense does not include parameter passing and return value management, so they are only briefly mentioned. You can use registers (`edi`, `esi`, `edx`, `ecx`) and in some cases the system stack to pass the parameters. A register (`eax`) is usually used to return the return value of the function to the

caller. In the meantime, we need to take care of the consistency of the registers and my system stack.

Comment:

Wider knowledge in the field and practice of assembly programming are advantageous.

Related subjects:

- *Assembly programming*
- *Computer architectures*

Further suggested readings:

- R. E. Bryant, D. R. O'Hallaron: Computer Systems – A programmer's perspective (Pearson, 2016)
  chapter 3.
- Joseph Cavanagh: X86 Assembly Language and C Fundamentals, (CRC Press, 2013)
  chapter 6.
- Richard Blum: Professional Assembly Language, (Wiley, 2005)
  chapter 6.