Materials

W9 Współczesna architektury procesorów i zestawy instrukcji: RISC i ARM

Omawiane zagadnienia

- Examining instruction execution characteristics
- Why Complex Instruction Set Architectures (CISC)
    - Do complex instructions help compilers to make a better code?
    - Do complex instructions help make programs smaller and execute faster?
- General characteristics of Reduced Instruction Set Architectures (RISC)
- ARM fundamentals

Examining instruction execution characteristics

Today we perceive hardware rarely from the perspective of assembler. More often we use high-level languages (HLLs) which:

- allow the programmer to express algorithms more concisely;
- allow the compiler to take care of details that are not important in the programmer's expression of algorithms;
- and often support naturally the use of structured programming and/or object-oriented design.

However high level languages simplifies a lot daily programmers routines they gave rise to a perceived problem, known as the **semantic gap**, the difference between the operations provided in HLLs and those provided in computer architecture. Symptoms of this gap may result in execution inefficiency, excessive machine program size, and compiler complexity. Fighting with this it may be tempting to design large instruction sets with multiple addressing modes, and various HLL statements implemented directly in hardware.

Meanwhile, a number of studies have been done over the years to determine the characteristics and patterns of execution of machine instructions generated from HLL programs. The results of these studies inspired some researchers to look for a different approach: namely, to make the architecture that supports the HLL simpler, rather than more

complex.

To understand the line of reasoning of the RISC advocates, you should take a closer look at instruction execution characteristics. The aspects of computation of interest are as follows:

- Operations performed: These determine the functions to be performed by the processor and its interaction with memory.

  Among all possible statements, assignment is the most frequent, suggesting that the simple movement of data is of high importance. There is also a massive number of control statements like `if` or `loop`. These statements are implemented in machine language with some sort of *compare* and *branch* instruction. This suggests that the sequence control mechanism of the instruction set is also very important.

  According to studies cited in [1, section 15.1] the relative frequency of occurrence of various HLL statements in a variety of programs is as follow (the data were obtained by observing the occurrences in running programs rather than just the number of times that statements occur in the source code):

  1. 43% – control flow `if`
  2. 38% – assign statement
  3. 12% – procedure call `call`
  4. 3% – loop statement

  However HLL `if` like all other statements requires few instructions in machine code. So you can not consider only assembler `if` but all assemblers instructions allowing you to complete HLL `if`. If you recalculate abowe results having this in mind, you obtain:

  1. 33% – procedure call `call`
  2. 32% – loop statement
  3. 21% – control flow `if`
  4. 13% – assign statement

  The results suggest that **the procedure call/return is the most time- consuming operation in typical HLL programs**. If you consider the number of memory references caused by each HLL statement you obtain:

  1. 45% – procedure call `call`
  2. 26% – loop statement
  3. 15% – assign statement
  4. 13% – control flow `if`

- Operands used: The types of operands and the frequency of their use determine the memory organization for storing them and the addressing modes for accessing them.

- Research shows that most references are to simple scalar variables:

    1. 55% – scalar variable
    2. 25% – array or structure
    3. 20% – integer constant

- Further, more than 80% of the scalars were local (to the procedure) variables.

- In addition, each reference to an array or a structure requires a reference to an index or pointer, which again is usually a local scalar.

Thus, **there is a preponderance of references to scalars, and these are highly localized**.

Studies suggest the importance of an architecture that lends itself to **fast operand accessing**, because this operation is performed so frequently. So as a conclusion it follows that a prime candidate for optimization is the **mechanism for storing and accessing local scalar variables**.

- Execution sequencing: This determines the control and pipeline organization.

  You have seen that procedure calls and returns are an important aspect of HLL programs because these are the most time-consuming operations in compiled HLL programs. Thus, it will be profitable to consider ways of implementing these operations efficiently. Two aspects are significant:

  - the number of parameters and variables that a procedure deals with,
  - and the depth of nesting.

  According to some studies 98% of dynamically called procedures are **passed fewer than six arguments** and that 92% of them used **fewer than six local scalar variables**. This allows us to draw a conclusion that the number of words required per procedure activation is not large.

  Studies on the pattern of procedure calls and returns in HLL programs revealed that **it is rare to have a long uninterrupted sequence of procedure calls** followed by the corresponding sequence of returns.

  These results reinforce the conclusion that:

  - a program remains confined to a rather narrow window of procedure-invocation depth
  - and operand references are highly localized.

Conclusions

1. **Need for optimize operand referencing.**

   The studies show that there are several references per HLL statement and that there is a high proportion of move (assignment) statements. This, coupled with the locality and predominance of scalar references, suggests that **performance can be improved by reducing memory references at the expense of more register references**. Because of the locality of these references, an expanded register set seems practical.

   The researchers varied the number of registers from 16 to 128, and they considered both the use of all general-purpose registers and registers split between integer and floating-point use. Their study showed that **with even simple register optimization, there is little benefit to the use of more than 64 registers**. With **reasonably sophisticated register optimization techniques, there is only marginal performance improvement with more than 32 registers**.

   Note that apart from hardware approach you can also base on software. The software approach is to rely on the compiler to maximize register usage. The compiler will attempt to assign registers to those variables that will be used the most in a given time period. This approach requires the use of sophisticated program-analysis algorithms. However this approach is rather an improvement but not a replacement of hardware solution – if the number of register is not enough any compiler optimization will not bring expected profits.

2. **Careful design of instruction pipelines to make conditional branch and procedure call not so expensive.**

   Because of the high proportion of conditional branch and procedure call instructions, a straightforward instruction pipeline will be inefficient. This manifests itself as a high proportion of instructions that are prefetched but never executed.

3. **Need for better knowledge what is executed and for how long.**

   Instructions should have predictable costs (measured in execution time, code size, and increasingly, in energy dissipation).

Note that all above conclusions characterize RISC architectures.

Why Complex Instruction Set Architectures (CISC)

Thinking behind the Complex Instruction Set Architectures is motivated by a desire to simplify compilers and a desire to improve performance. The means to achieve this goal is to **use richer instruction sets, which include a larger number of instructions and more**

complex instructions.

Do complex instructions help compilers to make a better code?

The task of the compiler writer is to build a compiler that generates good (fast, small, fast and small) sequences of machine instructions for HLL programs. If there are machine instructions that resemble HLL statements, this task is simplified.

This reasoning has been disputed by the RISC researchers. They have found that **complex machine instructions are often hard to exploit because the compiler must find those cases that exactly fit the construct**. The task of optimizing the generated code to minimize code size, reduce instruction execution count, and enhance pipelining is much more difficult with a complex instruction set.

In practice most of the instructions in a compiled program are the relatively simple ones.

Do complex instructions help make programs smaller and execute faster?

Smaller programs

There are two advantages to smaller programs. Because the program takes up less memory, there is a savings in that resource. With memory today being so inexpensive, this potential advantage is no longer compelling. More important, smaller programs should improve performance, and this will happen in three ways.

- **Less to fetch** Fewer instructions means fewer instruction bytes to be fetched.
- **Less page faults** In a paging environment, smaller programs occupy fewer pages, reducing page faults.
- **Better cache utilization** More instructions fit in cache(s).

The problem with this line of reasoning is that it is far from certain that a CISC program will be smaller than a corresponding RISC program.

In many cases, the CISC program, expressed in symbolic machine language, may be shorter (i.e., fewer instructions), but the number of bits of memory occupied may not be noticeably smaller. We saw this in section *Number of addresses* in *Modern processor architectures and instruction sets* chapter. To remind you, I consider there various sets of instructions:

- the most complex, three-address instructions case (4 instructions + 12 operands references),
- the opposite, simplified, one-address instructions case (8 instructions + 8 operands references)
- the simplest, zero-address instructions case (9 instructions + 5 operands references).

In this example, very simplified but not impossible, program expressed in the most simply set of instructions occupies less memory than one expressed in the most complex set of instructions (14 bytes and 16 bytes respectively).

So the expectation that a CISC will produce smaller programs, with the attendant advantages, may not be realized.

Faster programs

The second motivating factor for increasingly complex instruction sets was that instruction execution would be faster. It seems to make sense that a complex HLL operation will execute more quickly as a single machine instruction rather than as a series of more primitive instructions. However at the same time the entire **control unit must be made more complex**, and/or the **microprogram control store must be made larger**, to accommodate a richer instruction set. Either factor increases the execution time of the simple instructions.

General characteristics of Reduced Instruction Set Architectures (RISC)

Although a variety of different approaches to reduced instruction set architecture have been taken, certain characteristics are common to all of them:

- One machine instruction per machine cycle

  A machine cycle is defined to be the time it takes to fetch two operands from registers, perform an ALU operation, and store the result in a register.

  Thus, RISC machine instructions should be no more complicated than, and execute about as fast as, microinstructions on CISC machines. With simple, one-cycle instructions, there is little or no need for microcode; the machine instructions can be hardwired. Such instructions should execute faster than comparable machine instructions on other machines, because it is not necessary to access a microprogram control store during instruction execution.

- Register-to-register operations

Most operations should be register to register, with only simple `load` and `store` operations accessing memory. This design feature simplifies the instruction set and therefore the control unit. For example, a RISC instruction set may include only one or two `add` instructions (e.g., integer add and floating point add); while the VAX (ISA, instruction set architecture, of VAX, Virtual Address eXtension, is considered as the quintessence of CISC: it had a very large number of addressing modes and machine instructions) has 25 different ADD instructions. Another benefit is that such an architecture encourages the optimization of register use, so that frequently accessed operands remain in high-speed storage. This emphasis on register-to-register operations is notable for RISC designs.

Contemporary CISC machines provide such instructions but also include memory-to-memory and mixed register/memory operations.

- Simple addressing modes

  Almost all RISC instructions use simple register addressing. Several additional modes, such as displacement and PC-relative, may be included. Other, more complex modes can be synthesized in software from the simple ones. Again, this design feature simplifies the instruction set and the control unit.

- Simple instruction formats

  Generally, only one or a few formats are used. Instruction length is fixed and aligned on word boundaries. Field locations, especially the opcode, are fixed. This design feature has a number of benefits. With fixed fields, opcode decoding and register operand accessing can occur simultaneously. Simplified formats simplify the control unit. Instruction fetching is optimized because word-length units are fetched. Alignment on a word boundary also means that a single instruction does not cross page boundaries.

CISC or RISC – the winner is...

In more recent years, the RISC versus CISC controversy has died down to a great extent. This is because there has been a gradual convergence of the technologies.

After the initial enthusiasm for RISC machines, there has been a growing realization that:

- RISC designs may benefit from the inclusion of some CISC features and that
- CISC designs may benefit from the inclusion of some RISC features.

As chip densities and raw hardware speeds increase, RISC systems have become more complex. At the same time, in an effort to squeeze out maximum performance, CISC designs have focused on issues traditionally associated with RISC, such as an increased number of general- purpose registers and increased emphasis on instruction pipeline design.

The result is that the more recent RISC designs, notably the PowerPC, are no longer "pure" RISC and the more recent CISC designs, notably the Pentium II and later Pentium models, do incorporate some RISC characteristics.

ARM

The ARM architectures define a family of RISC processors suitable for use in a wide variety of applications. Processors based on ARM architectures are preferred in designs where a combination of high performance, low power consumption, and small physical size is needed.

ARM processors are used across a broad spectrum of applications, from tiny battery-powered devices to supercomputers. ARM processors serve as embedded processors in safety-critical systems, such as automotive anti-lock brakes, and as general-purpose processors in smart watches, portable phones, tablets, laptop computers, desktop computers, and servers.

Virtually all high-end smartphones and portable electronic devices produced since 2016 are designed around processors or SoCs based on the ARMv8-A 32-bit architecture. As of 2017, over 100 billion ARM processors have been manufactured.

ARM basic characteristic

ARM is a *load/store architecture*, requiring data to be loaded from memory to a register before any processing such as an ALU operation can take place upon it. A separate instruction then stores the result back to memory.

ARM processors are bi-endian. A configuration setting is available to select little-endian or big-endian byte order for multi-byte values. The default setting is little-endian, which is the configuration commonly used by operating systems.

The ARM architecture natively supports these data types:

- byte: 8 bits
- halfword: 16 bits
- word: 32 bits
- doubleword: 64 bits

By the way, mind the difference in meaning of "units". In x86 and x64, a *word* is 16 bits. In ARM, a *word* is 32 bits. But it is still in accordance with the meaning:

> *A word is the natural unit of data used by a particular processor design. A word is a fixed-sized datum handled as a unit by the instruction set or the hardware of the*

> *processor.*

First generation of x86 processors were 16-bit and this "definition" remains to this day (to maintain backward compatibility) as it is also used in specifications of many various data structures and formats.

The ARM register set

In `USR` mode, the ARM architecture has 16 general-purpose 32-bit registers named `R0` through `R15`. The first 13 registers are truly general-purpose, while the last three have the following defined functions:

- `R13` is the stack pointer, also named `SP` in assembly code. This register points to the top of the stack.
- `R14` is the link register, also named `LR`. This register holds the return address while in a called function. The use of a link register differs from x86/x64, which pushes the return address onto the stack. The reason for using a register to hold the return address is because it is significantly faster to resume execution at the address in LR at the end of a function than it is to pop the return address from the stack and resume execution at that address.
- `R15` is the program counter, also named `PC`. Due to pipelining, the value contained in `PC` is usually two instructions ahead of the currently executing instruction. **Unlike x86/x64, it is possible for user code to directly read and write the PC register.** Writing an address to `PC` causes execution to immediately jump to the newly written address.

The current program status register (`CPSR`) contains status and mode control bits, similar to `EFLAGS` / `RFLAGS` in the x86/x64 architectures.

Below you have a meaning of some of `CPSR` bits:

```
Bit Name Function
0-3 M    Mode. The current execution privilege level (see note below).
4   T    Thumb. Set if the T32 (Thumb) instruction set is active.
         If clear, the ARM instruction set is active.
9   E    Endianness. Setting this bit enables big-endian mode.
         If clear, little-endian mode is active.
28  V    Overflow flag. Set if an operation resulted in signed overflow.
29  C    Carry flag. Set if addition produced a carry or subtraction
         produced a borrow.
30  Z    Zero flag. Set if the result of an operation is zero.
31  N    Negative flag. Set if the result of an operation is negative.
```

Note: A32 and T32 instruction sets

A32 instructions, known as *ARM instructions* in pre-Armv8 architectures, are 32 bits wide, and are aligned on 4-byte boundaries. A32 was traditionally used in applications requiring the highest performance, or for handling hardware exceptions such as interrupts and processor start-up.

The *T32 instruction* set, known as *Thumb* in pre-Armv8 architectures, is a mixed 32- and 16-bit length instruction set that offers the designer excellent code density for minimal system memory size and cost.

T32 provides enhanced levels of performance, energy efficiency, and code density for a wide range of embedded applications. Designers can use both T32 and A32 instructions sets and therefore have the flexibility to emphasize performance or code size on a subroutine level as their applications require.

Note: ARM execution privilege levels

ARM processors support eight distinct execution privilege levels:

- User (USR)
- Supervisor (SVC)
- Fast interrupt request (FIQ)
- Interrupt request (IRQ)
- Monitor (MON)
- Abort (ABT)
- Undefined (UND)
- System (SYS)

By default, **most instructions do not affect the flags**. The `S` suffix must be used with, for example, an addition instruction (`adds`) to cause the result to affect the flags. Comparison instructions are the exception to this rule; they update the flags automatically.

ARM addressing modes

In true RISC fashion, the only ARM instructions that can access system memory are those that perform register loads and stores. The `ldr` instruction loads a register from memory, while `str` stores a register to memory. A separate instruction, `mov` , transfers the contents of one register to another or moves an immediate value into a register.

ARM addressing arsenal is as simple as it can only be and contains only indispensable modes.

- Immediate

  An immediate value is provided as part of the instruction. The possible immediate values consist of an 8-bit value, coded in the instruction, rotated through an even number of bit positions. A full 32-bit value cannot be specified because the instruction itself is, at most, 32 bits wide (as you can see the instruction length limit forces a certain compromise as we discussed in *Modern processor architectures and instruction sets* chapter).

  To load an arbitrary 32-bit value into a register, the `ldr` instruction must be used instead to load the value from memory:

  ```
  mov r0, #10         // Load the 32-bit value 10 decimal into r0
  mov r0, #0xFF000000 // Load the 32-bit value FF000000 hexadecimal in
  ```

  The second example contains the 8-bit value `FFh` in the instruction opcode. During execution, it is rotated left by `24` bit positions into the most significant 8 bits of the word.

- Register direct

  This mode copies one register to another:

  ```
  mov r0, r1 // Copy r1 to r0
  mvn r0, r1 // Copy NOT(r1) to r0
  ```

- Register indirect

  The address of the operand is provided in a register (the register containing the address is surrounded by square brackets):

  ```
  ldr r0, [r1] // Load the 32-bit value at the address given
               // in r1 to r0
  str r0, [r3] // Store r0 to the address in r3. Unlike most
               // instructions, `str` uses the first operand
               // as the source and the second as the destination.
  ```

- Register indirect with offset

  The address of the operand is determined by adding an *offset* to the *base* register:

```
ldr r0, [r1, #32] // Load r0 with the value at the address
                  // [r1+32]
str r0, [r1, #4]  // Store r0 to the address [r1+4]
```

- Register indirect with offset, pre-incremented

  The address of the operand is computed by adding an offset to the base register. The base register is updated to the computed address and this address is used to load the destination register:

```
ldr r0, [r1, #32]! // Update r1 to (r1+32) and then
                   // load r0 with [r1+32] and
str r0, [r1, #4]! // Store r0 to [r1+4] and update r1 to (r1+4)
```

- Register indirect with offset, post-incremented

  The base address is first used to access the memory location and then is updated to the computed address:

```
ldr r0, [r1], #32 // Load [r1] to r0, then update r1 to (r1+32)
str r0, [r1], #4  // Store r0 to [r1], then update r1 to (r1+4)
```

- Double register indirect

  The address of the operand is the sum of a *base register* and an *increment register* (the register names are surrounded by square brackets):

```
ldr r0, [r1, r2] // Load r0 with [r1+r2]
str r0, [r1, r2] // Store r0 to [r1+r2]
```

- Double register indirect with scaling

  The address of the operand is the sum of a *base register* and an *increment register* shifted left or right by a number of bits (the register names and the shift information are surrounded by square brackets):

```
ldr r0, [r1, r2, lsl #5] // Load r0 with [r1+(r2*32)]
str r0, [r1, r2, lsr #2] // Store r0 to [r1+(r2/4)]
```

ARM instruction categories

Most of the instructions are quite typical and corresponds to instracions you have seen in

case of x86 architecture (except that are designed to work on registers).

Among them conditional execution deserves more attention.

- Conditional execution

  Many ARM instructions support conditional execution, which uses the same condition codes as the branch instructions to determine whether individual instructions are executed. If an instruction's condition evaluates false, the instruction is processed as a no-op. The condition code is appended to the instruction mnemonic. This technique is formally known as predication.

  Consider the following example:

  ```
  mov r1, #0
  cmp r0, #10
  addpl r1, r0, #10
  addmi r1, r0, #20
  ```

  The `cmp` instruction subtracts `10` from the `r0` and sets the `N` flag if `r0` is less than `10`. Otherwise, the `N` flag is clear, indicating the value in `r0` is `10` or greater.

  If `N` is clear, the `addpl` instruction executes (`pl` means "plus," as in "not negative"), and the `addmi` instruction does not execute.

  If `N` is set, the `addpl` instruction does not execute and the `addmi` instruction executes.

  After this sequence completes, `r1` contains either 10 (`r0` is greater or equal 10) or 20 (`r0` is lower than 10).

  The most important consequence of using conditional execution is keeping the instruction pipeline flowing by avoiding branches.

Bibliography

1. William Stallings, *Computer Organization and Architecture*, Pearson, 11th edition (2019)
2. Jim Ledin, *Modern Computer Architecture and Organization*, Packt, April 2020
3. ARM architecture family, https://en.wikipedia.org/wiki/ARM*architecture*family, Retrieved 2023-12-10.
4. AVR microcontrollers, https://en.wikipedia.org/wiki/AVR_microcontrollers, Retrieved 2023-12-10.

5. Navigating the Cortex Maze, https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/navigating-the-cortex-maze, Retrieved 2023-12-10.
6. MIPS architecture, https://en.wikipedia.org/wiki/MIPS_architecture, Retrieved 2023-12-10.
7. SPARC, https://en.wikipedia.org/wiki/SPARC, Retrieved 2023-12-10.
8. RISC-V, https://en.wikipedia.org/wiki/RISC-V, Retrieved 2023-12-10.
9. RISC-V official webpage, https://riscv.org, Retrieved 2023-12-10.
10. PowerPC, https://en.wikipedia.org/wiki/PowerPC, Retrieved 2023-12-10.