УДК 004.423 DOI: 10.17587/prin.13.272-276

**А. В. Третьяк,** аспирант, alextretyak2@gmail.com,
**Е. А. Верещагина,** канд. техн. наук, доц., everesh@mail.ru, Дальневосточный федеральный университет, г. Владивосток,
**П. В. Струк,** вед. специалист, pavelstruk@yandex.ru, "Приморский океанариум" — филиал ННЦМБ ДВО РАН, г. Владивосток

# Разработка системы символьных макрокоманд (симкоманд) процессора

*Традиционной формой записи машинного кода является язык ассемблера. Его команды прямо соответствуют отдельным командам машины или их последовательностям. Синтаксис программы на языке ассемблера определяется главным образом системой команд конкретного процессора и системой директив конкретного транслятора. Представленная в статье система символьных макрокоманд процессора является, по мнению авторов, более наглядной формой представления машинного кода, что способствует повышению продуктивности при работе с ней программистом, а также минимизации ошибок.*

***Ключевые слова:*** *система команд процессора, язык ассемблера, макрокоманда*

## Введение

Традиционной формой записи машинного кода является язык ассемблера. Язык ассемблера — машинно-ориентированный язык программирования низкого уровня. Он представляет собой систему обозначений, используемую для представления в удобно читаемой форме программ, записанных в машинном коде.

Команды языка ассемблера прямо соответствуют отдельным командам машины или их последовательностям [1]. Синтаксис программы на языке ассемблера определяется главным образом системой команд конкретного процессора и системой директив конкретного транслятора. Система команд процессора — это набор допустимых для данного процессора управляющих кодов и способов адресации данных [2]. Система команд процессора включает в себя все команды, которые могут быть записаны с использованием различных видов синтаксиса языка ассемблера.

Для некоторых платформ может существовать несколько видов синтаксиса языка ассемблера, не совместимых между собой. Например, наиболее популярные синтаксисы языков ассемблера для Intel-совместимых процессоров — Intel-синтаксис и AT&T-синтаксис. Однако все существующие виды синтаксиса языка ассемблера ориентированы, в первую очередь, на "легкую читаемость" со стороны ЭВМ, а не программиста.

Объединение отдельных команд для выполнения определенной задачи в языке ассемблера называется макрокомандой процессора.

Макрокоманда — это символьное имя, заменяемое при обработке препроцессором на последовательность программных инструкций [3]. Другими словами, макрокоманда — это мнемоническая команда, которая разворачивается более чем в одну машинную инструкцию [4].

Команды процессора в языке ассемблера обозначаются латинскими буквами, например, команда `mov` присваивает значение заданному регистру.

Для записи команд процессора авторами предлагается ввести использование вместо латинских букв символьное обозначение, например, команде ассемблера `mov eax, 2` соответствует символьная команда `eax = 2`. Для обозначения данного способа записи команд предлагается ввести понятие "симкоманда".

Симкоманда — это символьная макрокоманда процессора с Си-подобным синтаксисом. Симкоманды предназначены для наиболее четкого выражения намерения программиста, скрывая незначительные особенности архитектуры процессора. Так, например, очевидно, что при использовании ассемблерной команды `xor eax, eax` намерением программиста было не выполнить операцию "исключающего или" над регистром `eax`, а просто установить его значение равным нулю. На языке симкоманд это действие выражается понятной

симкомандой `eax = 0`, в то время как на языке ассемблера x86 понятная команда `mov eax, 0` не используется в силу того, что есть более эффективная команда, соответствующая этому же действию: `xor eax, eax`. Аналогично и в случае с командой `test eax, eax`, которая равнозначна команде `cmp eax, 0`, однако является более эффективной.

Таким образом, целью введения нового языка симкоманд является повышение эффективности как с аппаратной точки зрения (более эффективное исполнение на процессоре, реализующем систему симкоманд), так и со стороны программиста (минимизация ошибок, более наглядная форма представления машинного кода).

Задача симкоманд — обеспечить понятное отображение на машинный код, причем по возможности самым оптимальным образом. Так, например, симкоманда `edx = eax + ebx*2` транслируется в `lea edx, [eax + ebx*2]` и является допустимой, а симкоманда `edx = eax + ebx*33` уже не является допустимой, так как не может быть транслирована в одну инструкцию. Хотя симкоманды вызова функций являются макрокомандами (т. е. составными, состоящими из нескольких инструкций, командами), они являются допустимыми на основании того, что порождаемые ими ассеблерные инструкции являются оптимальными и не нуждаются в раскрытии/отображении их программисту.

Цель исследования, результаты которого представлены в статье, — разработка системы символьных макрокоманд (симкоманд) процессора как более наглядной формы представления машинного кода для повышения эффективности при работе с ней программиста, включая минимизацию ошибок.

## Симкоманды арифметических и логических операций

В табл. 1 представлены наиболее часто используемые арифметические [5] и логические [6] команды процессора (на примере ассемблера для архитектуры x86) и соответствующие им разработанные авторами симкоманды.

*Таблица 1*

| Команда ассемблера x86 | Симкоманда |
|---|---|
| `add eax, ebx` | `eax += ebx` |
| `sub eax, ebx` | `eax -= ebx` |
| `imul ebx` | `eax *= ebx` |
| `mul ebx` | `eax **= ebx` |
| `imul ecx, edx` | `ecx *= edx` |
| `imul ecx, edx, 10` | `ecx = edx * 10` |
| `idiv ebx` | `eax /= ebx` |
| `div ebx` | `eax //= ebx` |
| `neg eax` | `eax = -eax` |
| `inc eax` | `eax++` |
| `dec eax` | `eax--` |
| `and eax, ebx` | `eax &= ebx` |
| `or eax, ebx` | `eax \|= ebx` |
| `xor eax, ebx` | `eax (+) = ebx` |
| `xor eax, eax` | `eax= 0` |
| `not eax` | `eax = (-)eax` |
| `sal eax, cl` | `eax <<= cl` |
| `sar eax, cl` | `eax >>= cl` |
| `shr eax, cl` | `eax >>>= cl` |
| `rol eax, cl` | `eax (<<)= cl` |
| `ror eax, cl` | `eax (>>)= cl` |

## Симкоманды перехода

В качестве примера использования симкоманд перехода в табл. 2 представлена часть функции для определения того факта, является ли заданная строка палиндромом, т. е. читается ли она

*Таблица 2*

| Ассемблер x86-64 [7] | Симкоманды x86-64 |
|---|---|
| `palindrome_start:`<br>`  cmp rcx, 0`<br>`  jl palindrome_end`<br>`  mov rbx, rdx`<br>`  sub rbx, rcx`<br>`  sub rbx, 1`<br>`  mov bl, byte [rdi + rbx]`<br>`  cmp byte [rdi + rcx], bl`<br>`  jne palindrome_faileddec rcx`<br>`  dec rcx`<br>`  jmp palindrome_start`<br>`palindrome_end:` | `palindrome_start:`<br>`  rcx < 0: palindrome_end`<br><br>`  rbx = rdx`<br>`  rbx -= rcx`<br>`  rbx -= 1`<br>`  bl = [rdi + rbx]`<br>`  [rdi + rcx] != bl: palindrome_failed`<br><br>`  rcx--`<br>`  :palindrome_start`<br>`palindrome_end:` |

одинаково в обоих направлениях — слева направо и справа налево.

Можно заметить, что такая запись делает инструкции безусловного перехода симметричными записи, используемой для обозначения меток, так как запись `метка:` обозначает объявление метки, а `:метка` — переход на эту метку.

Такая запись `[<условие _ или _ команда _ меняющая _ флаги>] :метка` имеет перечисленные далее преимущества.

1. Четко обозначает связь команды, меняющей флаги, с командой перехода. Так, новичкам в ассемблере может быть не очевидно, что команда `jnz label` следующая за командой `dec ecx` опирается на тот факт, что команда `dec ecx` устанавливает или сбрасывает флаг ZF (флаг нуля (zero flag) устанавливается, если при выполнении операции получается число равное нулю). Запись `--ecx != 0 :label` при этом показывает, что операция `--ecx` (`dec ecx`) непосредственно связана с командой перехода на метку `label`. Если посмотреть с позиции машинной логики, то можно отметить, что такая запись показывает, что флаг ZF, изменяемый командой `--ecx`, реально используется. Это означает, что в этом отношении симкоманды выражают намерение более четко как для программиста, так и для машины. Этот момент может быть использован для разработки новых процессорных архитектур, развивая направление, по которому пошли в ARM: "Adding the `s` suffix to `sub` causes it to update the flags itself, based on the result of the operation." [8]. Значит вместо `subs` в `subs r4, r4, #1` в гипотетической новой архитектуре можно было бы написать `subz`, так как реально используется только флаг ZF. Другие флаги при этом (например, флаг переполнения) не используются.

Следует заметить, что предложенная запись для условного перехода лучше, чем просто добавление новых инструкций типа `subz`. Лучше она тем, что дает явное указание не только на тот факт, что используется только флаг ZF, но и на тот факт, что флаг ZF используется **только** для соответствующей инструкции перехода и в дальнейшем он использован не будет. Использование обоих отмеченных фактов представляет возможность для дополнительной оптимизации на уровне процессора, если рассматривать возможность разработки архитектуры процессора, реализующего систему симкоманд.

2. При использовании для сравнения обозначения `==` можно ошибиться и написать например, `eax = 0` вместо `eax == 0`. Эту возможную ошибку можно обойти путем использования другого обозначения для сравнения (например, оставив запись `cmp a, b` вместо `a == b`) либо другого символьного обозначения для присваивания (например, `<-` или `:=` вместо `=`). Другое символьное обозначение для операции присваивания нерационально, так как оператор = уже слишком твердо закрепился в языках программирования и вполне подходит для этой операции. В связи с этим предлагается пойти по другому пути — просто запретить запись `x == 0` в чистом виде, а разрешить ее только как составную часть какой-либо более сложной операции, в составе которой оператор = запрещен. В этом случае для ошибки нужно будет ошибиться как минимум дважды — во-первых, перепутать = и ==, и, во-вторых, использовать неверную запись. Это обстоятельство сводит вероятность случайной ошибки к минимуму. Речь в данном случае идет о том, что запись `eax == 0` оставляет возможность случайной ошибки, а запись `eax == 0 : label` такой возможности практически не оставляет, так как запись `eax = 0 : label` является запрещенной.

Рассмотрим также пример для ARM-архитектуры:

```
subs r4, r5, #1
beq loop _ label
```

Данной паре ARM-инструкций соответствует следующая симкоманда:

```
r4 = r5 - 1 == 0: loop _ label
```

Несмотря на некоторую громоздкость этой записи, здесь также исключается вероятность случайных ошибок, связанных с перепутыванием = и ==, так как следующие три записи не являются действительными симкомандами:

```
r4 = r4 - 1 = 0: loop _ label
r4 == r4 - 1 = 0: loop _ label
r4 == r4 - 1 == 0: loop _ label
```

### Псевдонимы для регистров

Псевдоним для регистра можно объявить как

```
eax'i = 0
```

или:

```
i'eax = 0
```

Первая форма удобна для такой записи:

```
ecx'i = 0
ebx' length = 0
edx' some _ variable _ with _ long _ name = 0
```

Так как это выглядит лучше, чем

```
i'ecx = 0
length'ebx = 0
some _ variable _ with _ long _ name'edx = 0
```

Вторая форма удобна для следующего примера [9]:

```
theta _ l0:
   ; Theta
   lodsd                  ; t = st[i ];
   xor  eax, [esi+5*4-4]  ; t ^= st[i+5];
   xor  eax, [esi+10*4-4] ; t ^= st[i+10];
   xor  eax, [esi+15*4-4] ; t ^= st[i+15];
   xor  eax, [esi+20*4-4] ; t ^= st[i+20];
   stosd                  ; bc[i] = t;
   loop theta _ l0
   popad
   xor  eax, eax
theta _ l1:
   movzx ebp, byte[ebx+eax + 4]  ; ebp = m[(i+4)];
   mov   ebp, [edi+ebp*4]        ; t = bc[m[(i+4)]];
   movzx edx, byte[ebx+eax + 1]  ; edx = m[(i+1)];
   mov   edx, [edi+edx*4]        ; edx = bc[m[(i+1)]];
   rol   edx, 1                  ; t ^= ROTL32(edx, 1);
   xor   ebp, edx
theta _ l2:
```

На языке симкоманд данный ассемблерный код можно написать так:

```
theta _ l0:
   t'eax = [st'esi ++ ]
   t' (+)= st'[ 5*4-4]
   t' (+)= st'[10*4-4]
   t' (+)= st'[15*4-4]
   t' (+)= st'[20*4-4]
   [edi'bc ++ ] = t'.
   --ecx == 0: theta _ l0
   popad
   eax'i = 0
theta _ l1:
   ebp't = byte ebx'm['i+4]
      't = 'bc['t*4]
   edx's = byte 'm['i+1]
      's = 'bc['s*4]
      's (<<)= 1
      't (+)= 's
theta _ l2:
```

Верификатор симкода проверяет, что объявленный псевдоним больше не используется после назначения другого псевдонима на этот же регистр.

Пример:

```
i'eax = 10
//...
|| Здесь можно обращаться к i', например, так:
i' += 2
\\...
c'eax = 0
//...
|| А здесь уже нельзя!
\\...
```

Если псевдоним был объявлен как `регистр'псевдоним`, то обращаться к нему можно только посредством записи `'псевдоним`, а если объявлен как `псевдоним'регистр`, тогда посредством записи `псевдоним'`.

Переназначение псевдонима без явного освобождения (см. далее) на другой регистр не допускается, так как в ассемблере нет понятия областей видимости. Как следствие, можно по ошибке в большой функции добавить в середину новый код, который переназначая псевдоним изменяет таким образом поведение последующего кода.

Псевдоним можно освободить посредством такой записи:

```
i'. // освобождает псевдоним i'
i', r'. // освобождает псевдонимы i' и r'
```

После освобождения этот псевдоним можно назначать на любой регистр.

## Заключение

В результате апробации языка симкоманд на примере, представленном в разд. "Симкоманды перехода", можно заметить повышение эффективности исполнения на процессоре, реализующем систему симкоманд. Эффективность при этом повышается и для программиста, это касается вопросов минимизации ошибок и более наглядной формы представления машинного кода.

Следует отметить, что симкоманды условных переходов спроектированы таким образом, чтобы исключить ошибку спутывания оператора сравнения == и оператора присваивания =, встречающуюся в языках высокого уровня (C++, PHP, JavaScript). Для таких языков, например, конструкция `if (a = 5) c == 7`; является допустимой, хотя содержит сразу две ошибки, так как правильно писать `if (a == 5) c = 7`.

При использовании симкоманд отмеченная ошибка исключена, так как

- запись `eax == 0` в чистом виде не является симкомандой;
- симкоманда `eax = 0 : label` является недействительной.

Для появления ошибки нужно будет ошибиться как минимум дважды — во-первых, перепутать = и ==, и, во-вторых, использовать неверную запись, что сводит вероятность случайной ошибки к минимуму. Учитывая изложенные выше аргументы, можно полагать, что представленная система символьных макрокоманд (симкоманд) процессора является более наглядной формой представления машинного кода для программиста, а также она способствует минимизации ошибок в коде, что повышает продуктивность при создании программного кода.

**Список литературы**


1. **Язык** ассемблера — Википедия. URL: https://ru.wikipedia.org/wiki/Язык_ассемблера
2. **Характеристика** системы команд процессора. URL: https://studfile.net/preview/5055379/page:16/
3. **Макрокоманда** — Википедия. URL: https://ru.wikipedia.org/wiki/Макрокоманда
4. **Васенин В. А., Кривчиков М. А.** Формальные модели программ и языков программирования. Часть 1. Библиографический обзор 1930—1989 гг. // Программная инженерия. 2015. № 5. С. 10—19.
5. **x86** Assembly/Arithmetic — Wikibooks. URL: https://en.wikibooks.org/wiki/X86_Assembly/Arithmetic
6. **x86** Assembly/Logic — Wikibooks. URL: https://en.wikibooks.org/wiki/X86_Assembly/Logic
7. **GitHub** — jlhonora / asm-examples. URL: https://github.com/jlhonora/asm-examples/blob/master/palindrome/palindrome.asm
8. **Condition** Codes 1: Condition flags and codes—Processors blog—Processors—Arm ommunity. URL: https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/condition-codes-1-condition-flags-and-codes
9. **GitHub**—odzhan / tinycrypt /.../ k800x.asm. URL: https://github.com/odzhan/tinycrypt/blob/master/permutation/keccak/old/k800x.asm


# Developing a System of Symbolic Macro Commands (Symcommands) of the Processor


**A. V. Tretiak**, alextretyak2@gmail.com, **E. A. Vereschagina**, everesh@mail.ru, Far Eastern Federal University, Vladivostok, 690091, Russian Federation,
**P. V. Struk**, pavelstruk@yandex.ru, Primorsky Aquarim, Vladivostok, 690922, Russian Federation

*Corresponding author:*

**Tretiak Alexander V.,** Assistant, Far Eastern Federal University, Vladivostok, 690091, Russian Federation
E-mail: alextretyak2@gmail.com



*Received on March 23, 2022*
*Accepted on April 15, 2022*



*The traditional form of recording machine code is assembly language. Its commands correspond directly to individual machine commands or sequences of commands. The syntax of an assembly language program is determined mainly by the instruction system of the specific processor and the directive system of the specific translator. For some platforms, there may be several kinds of assembly language syntax that are not compatible with each other. For example, the most popular assembly language syntaxes for Intel-compatible processors are Intel syntax and AT&T syntax. However, all existing types of assembly language syntax are primarily focused on easy readability by the computer, not the programmer. The system of symbolic macro commands of the processor presented in the article is a more clear form of representation of machine code, which promotes productivity when the programmer works with it, and also minimization of errors.*



***Keywords:*** *processor instruction set, assembly language, macro command*

*For citation:*

**Tretiak A. V., Vereschagina E. A., Struk P. V.** Developing a System of Symbolic Macro Commands (Symcommands) of the Processor, *Programmnaya Ingeneria,* 2022, vol. 13, no. 6, pp. 272—276.

DOI: 10.17587/prin.13.272-276


**References**


1. **Assembly** language — Wikipedia, available at: https://ru.wikipedia.org/wiki/Язык_ассемблера (in Russian).
2. **Characteristics** of the processor instruction set, available at: https://studfile.net/preview/5055379/page:16/ (in Russian).
3. **Macro** command — Wikipedia, available at: https://ru.wikipedia.org/wiki/Макрокоманда (in Russian).
4. **Krivchikov M. A., Vasenin V. A.** Formal Models of Programming Languages and Programs. Part 1. Literature Review: 1930—1989, *Programmnaya Ingeneria,* 2015, no. 5, pp. 10—19 (in Russian).
5. **x86** Assembly/Arithmetic — Wikibooks, available at: https://en.wikibooks.org/wiki/X86_Assembly/Arithmetic
6. **x86** Assembly/Logic — Wikibooks, available at: https://en.wikibooks.org/wiki/X86_Assembly/Logic
7. **GitHub** — jlhonora / asm-examples, available at: https://github.com/jlhonora/asm-examples/blob/master/palindrome/palindrome.asm
8. **Condition** Codes 1: Condition flags and codes—Processors blog—Processors—Arm Community, available at: https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/condition-codes-1-condition-flags-and-codes
9. **GitHub** — odzhan / tinycrypt /.../ k800x.asm, available at: https://github.com/odzhan/tinycrypt/blob/master/permutation/keccak/old/k800x.asm

DOI: 10.17587/prin.13.277-285

# Software Engineering Education: Discovering Future Skills[1]

**S. M. Avdoshin,** Professor, savdoshin@hse.ru, **E. Y. Pesotskaya,** Associate Professor, epesotskaya@hse.ru, **D. M. Kuruppuge,** Graduate Student, dkuruppuge_1@edu.hse.ru, HSE University, Moscow, 101000, Russian Federation

*Corresponding author:*

**Pesotskaya Elena Y.,** Associate Professor, Faculty of Computer Science / School of Software Engineering, HSE University, Moscow, 101000, Russian Federation
E-mail: epesotskaya@hse.ru

*Received on August 30, 2021*
*Accepted on March 17, 2022*

*In the contemporary digital age the majority of industries become very much software dependent, require automation, digitization, and deployment of new technologies. The need for specialists, skilled in software engineering, is in high demand. In this study, we set out to investigate what challenges may arise in IT students' skill sets and new competences in the software engineering context and answer the question "how is skills demand evolving and where the gaps will be?". The paper analyzes what specific knowledge needs to be developed during university education and identifies their evolving skills demand. The paper recommends how students and young specialists in software engineering need to change themselves to adapt to the changing world.*

***Keywords:*** *software engineering, technologies, IT, education, skills*

*For citation:*

**Avdoshin S. M., Pesotskaya E. Y., Kuruppuge D. M.** Software Engineering Education: Discovering Future Skills, *Programmnaya Ingeneria*, 2022, vol. 13, no. 6, pp. 277—285.

УДК 004

**С. М. Авдошин,** канд. техн. наук, проф., savdoshin@hse.ru, **Е. Ю. Песоцкая,** канд. экон. наук, доц., epesotskaya@hse.ru, **Д. М. Куруппуге,** студент, dkuruppuge_1@edu.hse.ru, Национальный исследовательский университет "Высшая школа экономики", Москва

# Программные инженеры будущего: требования к навыкам

*В современную цифровую эпоху большинство отраслей становятся очень программно-зависимыми, требуют автоматизации, цифровизации и внедрения новых технологий. Потребность в специалистах, разбирающихся в программной инженерии, высока. В статье приведены результаты исследования того, какие могут возникнуть проблемы в наборе навыков при обучении программных инженеров и потребности в новых компетенциях при разработке программного обеспечения. Дан ответ на вопрос "как меняется спрос на навыки и где будут пробелы в знаниях?". Приведен анализ того, какие конкретные знания необходимо развивать во время обучения в вузе, и определена потребность в растущих навыках студентов. Даны рекомендации, как студентам и молодым специалистам в области программной инженерии необходимо изменить себя, чтобы адаптироваться к меняющемуся миру.*

***Ключевые слова:*** *программная инженерия, технологии, ИТ, образование, навыки*

## Introduction

*The future of work will be a race between education and technology.*
*Mauricio Macri, President of Argentina in 2018*

With the transition of business to the Internet, many vacancies opened up, hence, employees with IT skills and software engineers are in high demand. Employment requirements became more stringent as the workforce needs to understand and become familiar with new technologies and how they affect the industry landscape. Specific knowledge needs to be continuously updated, accordingly, workers need to adapt. In the future, employers will seek out candidates that can continuously acquire and perfect new skills.

[1] The article is based on the materials of the report at the Seventh International Conference "Actual problems of Systems and Software Engineering" APSSE 2021.