Министерство цифрового развития, связи и массовых коммуникаций
Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Сибирский государственный университет телекоммуникаций и информатики»
Кафедра Вычислительных систем (ВС)

ОТЧЕТ
о практической работе
«Генерация и анализ ассемблерного кода программы на Си под разные архитектуры.»
Вариант №2

Работу выполнил:
студент 1 курса
группы ИС-541
Устюжанин Д.К.

г. Новосибирск
2025 г.

Содержание

Цель работы

Научиться генерировать и анализировать ассемблерный код на языке Си под разные архитектуры (x86-64, ARM64, RISC-V). Понять, как высокоуровневые конструкции языка Си реализуются на уровне машинных инструкций, и как компилятор адаптирует код под разные платформы.

Листинг 1.1 – program.c

```
int fac(int N) {
     int x=1;
     for (int i=1; i<=N; i++) x*=i;
     return x;
}

/*
// Пример вызова:
int main() {
     fac(10); // !10 = 3628800
}
*/
```

Процесс получения ассемблерных листингов

Операционная система и ее версия: Arch Linux (rolling) x86_64.

Установка компилятора RISC-V 64 для Arch Linux:

```
$ pacman -S riscv64-linux-gnu-gcc
```

Версии всех трех компиляторов:

```
$ gcc --version
```

Вывод: `gcc (GCC) 15.2.1 20251112`

```
$ x86_64-w64-mingw32-gcc --version
```

Вывод: `x86_64-w64-mingw32-gcc (GCC) 15.2.0`

```
$ riscv64-linux-gnu-gcc --version
```

Вывод: `riscv64-linux-gnu-gcc (GCC) 15.1.0`

Команды компиляции для каждого (одна строка – одна команда компиляции):

```
$ gcc -S -O0 -o program_gcc.s program.c
$ x86_64-w64-mingw32-gcc -S -O0 -o program_mingv.s program.c
$ riscv64-linux-gnu-gcc -S -O0 -o program_riscv64.s program.c
```

Стоит отметить, что команды практически идентичны друг другу, а различается только название компилятора.

Листинг 2.1 – (x86_64 (CISC)) GCC (–S –O0)

```
     .file     "program.c"
     .text
     .globl    fac
     .type     fac, @function
fac:
.LFB0:
     .cfi_startproc
     pushq     %rbp
     .cfi_def_cfa_offset 16
     .cfi_offset 6, -16
     movq %rsp, %rbp
     .cfi_def_cfa_register 6
     movl %edi, -20(%rbp)
     movl $1, -8(%rbp)
     movl $1, -4(%rbp)
     jmp  .L2
.L3:
     movl -8(%rbp), %eax
     imull     -4(%rbp), %eax
     movl %eax, -8(%rbp)
     addl $1, -4(%rbp)
.L2:
     movl -4(%rbp), %eax
     cmpl -20(%rbp), %eax
     jle  .L3
     movl -8(%rbp), %eax
     popq %rbp
     .cfi_def_cfa 7, 8
     ret
     .cfi_endproc
.LFE0:
     .size     fac, .-fac
     .ident    "GCC: (GNU) 15.2.1 20251112"
     .section  .note.GNU-stack,"",@progbits
```

Листинг 2.2 - (x86_64 (CISC)) MinGW (–S –O0)

```
     .file     "program.c"
     .text
     .globl    fac
     .def fac; .scl 2;   .type     32;  .endef
     .seh_proc fac
fac:
     pushq     %rbp
     .seh_pushreg   %rbp
     movq %rsp, %rbp
     .seh_setframe  %rbp, 0
     subq $16, %rsp
     .seh_stackalloc     16
     .seh_endprologue
     movl %ecx, 16(%rbp)
     movl $1, -4(%rbp)
     movl $1, -8(%rbp)
     jmp  .L2
.L3:
     movl -4(%rbp), %eax
     imull     -8(%rbp), %eax
     movl %eax, -4(%rbp)
     addl $1, -8(%rbp)
.L2:
     movl -8(%rbp), %eax
     cmpl 16(%rbp), %eax
     jle  .L3
     movl -4(%rbp), %eax
     addq $16, %rsp
     popq %rbp
     ret
     .seh_endproc
     .ident    "GCC: (GNU) 15.2.0"
```

Листинг 2.3 - (RISC) RISC-V 64 (-S –O0)

```
     .file     "program.c"
     .option pic
     .attribute arch,
"rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0_zifencei2p0_zmmul1p0_z
aamo1p0_zalrsc1p0_zca1p0_zcd1p0"
     .attribute unaligned_access, 0
     .attribute stack_align, 16
     .text
     .align    1
     .globl    fac
     .type     fac, @function
fac:
.LFB0:
     .cfi_startproc
     addi sp,sp,-48
     .cfi_def_cfa_offset 48
     sd   ra,40(sp)
     sd   s0,32(sp)
     .cfi_offset 1, -8
     .cfi_offset 8, -16
     addi s0,sp,48
     .cfi_def_cfa 8, 0
     mv   a5,a0
     sw   a5,-36(s0)
     li   a5,1
     sw   a5,-20(s0)
     li   a5,1
     sw   a5,-24(s0)
     j    .L2
.L3:
     lw   a5,-20(s0)
     mv   a4,a5
     lw   a5,-24(s0)
     mulw a5,a4,a5
     sw   a5,-20(s0)
     lw   a5,-24(s0)
     addiw     a5,a5,1
     sw   a5,-24(s0)
.L2:
     lw   a5,-24(s0)
     mv   a4,a5
     lw   a5,-36(s0)
     sext.w    a4,a4
     sext.w    a5,a5
     ble  a4,a5,.L3
     lw   a5,-20(s0)
     mv   a0,a5
```

```
     ld   ra,40(sp)
     .cfi_restore 1
     ld   s0,32(sp)
     .cfi_restore 8
     .cfi_def_cfa 2, 48
     addi sp,sp,48
     .cfi_def_cfa_offset 0
     jr   ra
     .cfi_endproc
.LFE0:
     .size     fac, .-fac
     .ident    "GCC: (GNU) 15.1.0"
     .section  .note.GNU-stack,"",@progbits
```

Анализ и сравнение

Архитектуры демонстрируют принципиально разные подходы к организации регистров. В x86-64 вариантах используются традиционные регистры с префиксом '%', унаследованные от долгой эволюции архитектуры. Linux следует System V ABI с использованием %edi для передачи аргументов, тогда как Windows применяет Microsoft x64 ABI с регистром %ecx.

RISC-V представляет современную философию именования регистров с простыми мнемоническими именами: a0 для аргументов, s0 для сохраненных значений, sp для указателя стека, ra для адреса возврата. Эта регулярность отражает современный подход к проектированию архитектуры.

Все три реализации единодушны в использовании регистров для передачи аргумента и возврата результата. Однако способы управления стеком существенно различаются. Windows проявляет строгость, требуя явного выделения 16 байт для выравнивания стека. Linux обходится минимальным вмешательством, а RISC-V выделяет 48 байт, демонстрируя консервативный подход.

Структура пролога и эпилога также отражает архитектурные особенности. x86-64 варианты используют классическую пару push/pop %rbp, тогда как RISC-V требует явного сохранения и восстановления нескольких регистров.

Реализация цикла for ярко иллюстрирует философские различия архитектур. В x86-64 вариантах цикл реализован компактно с арифметическими операциями, работающими напрямую с ячейками стека, и эффективной проверкой условия через пару compare-jump.

RISC-V демонстрирует совершенно иную картину: каждая операция требует явной загрузки значений в регистры, выполнения операции и сохранения результата обратно в память. Инструкция умножения mulw и условного перехода ble работают исключительно с регистрами, что соответствует load/store архитектуре. Знаковое расширение перед сравнением подчеркивает внимание к деталям типизации в 64-битной среде.

Сравнение ABI Linux System V и Windows Microsoft x64 выявляет существенные различия в философии проектирования. Linux придерживается минималистичного подхода, передавая аргументы в %edi и экономно используя стек. Windows демонстрирует более структурированный подход с обязательным выравниванием стека и выделением shadow space для аргументов. Директивы SEH в Windows против CFI в Linux отражают разные подходы к обработке исключений и отладке.

RISC-V воплощает современные принципы RISC-архитектуры в чистой форме. Load/store природа проявляется в требовании явных операций загрузки и сохранения для любой работы с памятью. Регулярность инструкций выражена в последовательном трехадресном формате и логичной системе именования регистров. Отсутствие флагового регистра приводит к встроенным проверкам условий в инструкциях перехода, что отличает RISC-V от традиционного подхода x86. Знаковое расширение и явное управление 32-битными операциями в 64-битной среде демонстрируют тщательно продуманную систему типизации.

Вывод

Изучение ассемблерного кода показало глубокую связь между языком C и аппаратной архитектурой. Один и тот же алгоритм на C компилируется в существенно разные машинные инструкции в зависимости от целевой платформы. Это демонстрирует, что C является не просто высокоуровневым языком, а своеобразным мостом между логикой программиста и возможностями процессора.

Компилятор выполняет сложную работу по адаптации кода, учитывая регистровые соглашения, организацию стека и систему инструкций каждой архитектуры. Он преобразует абстрактные конструкции C в конкретные машинные команды, соблюдая особенности ABI и принципы работы целевой платформы. Это обеспечивает переносимость кода без изменения исходников.

Знание ассемблера остается критически важным для системного программиста. Оно позволяет понимать реальную производительность кода, эффективно отлаживать сложные проблемы и создавать оптимизированные кросс платформенные решения. В условиях разнообразия современных архитектур это знание помогает выбирать оптимальные подходы для каждой платформы и понимать компромиссы при переносе программного обеспечения.