Universidade de São Paulo
Escola de Artes, Ciências e Humanidades
Sistemas de Informação

ACH2034: Organização e Arquitetura de Computadores I

Professor Doutor Fabio Nakano

Semestre 2024-01

27/06/2024

EP#2

Adryelli Reis dos Santos
(Número USP: 14714019)

Gabriel Monteiro de Souza
(Número USP: 14746450)

Introdução

Este relatório apresenta uma análise detalhada da tradução de códigos em linguagem C para instruções em Assembly x86-64. A arquitetura x86-64 é amplamente utilizada em sistemas modernos devido à sua compatibilidade e desempenho eficiente. O estudo envolve a geração de código em Assembly a partir de exemplos simples em C, explorando como operações de alto nível são convertidas em instruções de baixo nível que o processador pode executar diretamente.

A geração de código em Assembly foi realizada utilizando compiladores e ferramentas de desenvolvimento que permitem visualizar o fluxo de controle e a manipulação de dados em nível de registrador. Cada exemplo examinado neste relatório demonstra a alocação de variáveis na pilha, o uso de registradores para operações aritméticas e lógicas, e a interação com funções do sistema por meio de chamadas de função.

O objetivo deste estudo é proporcionar uma compreensão prática e aplicada de como programas em C são otimizados e implementados em um nível mais baixo, contribuindo para uma melhor compreensão da interação entre software e hardware em sistemas computacionais baseados em x86-64.

Exercício 1 (EP2-1)

Código em C

Listing 1: Código em C

```c
#include <stdio.h>


void main () {
    int a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s;
    s = a + b + c + d + e + f + g + h + i + j + k + l + m + n + o + p + q + r;
    // Intel i7 em modo 64-bits tem 16 registradores de uso geral
    // (https://www.quora.com/How-many-registers-are-there-in-modern-64-bit-
        CPUs-like-intel-core-i5-or-i7#:~:text=The%20Intel%20Core%20i7%20
        processors,registers%20in%2064%2Dbit%20mode.)
}
```

Código em Assembly

Listing 2: Código em Assembly

```
    .file "ep2-1.c"
    .text
    .globl main
    .type main, @function
main:
.LFB0:
    .cfi_startproc
    endbr64
    pushq %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq %rsp, %rbp
    .cfi_def_cfa_register 6
    movl -76(%rbp), %edx
    movl -72(%rbp), %eax
    addl %eax, %edx
    movl -68(%rbp), %eax
    addl %eax, %edx
    movl -64(%rbp), %eax
    addl %eax, %edx
    movl -60(%rbp), %eax
    addl %eax, %edx
    movl -56(%rbp), %eax
    addl %eax, %edx
    movl -52(%rbp), %eax
    addl %eax, %edx
    movl -48(%rbp), %eax
    addl %eax, %edx
    movl -44(%rbp), %eax
    addl %eax, %edx
    movl -40(%rbp), %eax
```

```
    addl %eax, %edx
    movl -36(%rbp), %eax
    addl %eax, %edx
    movl -32(%rbp), %eax
    addl %eax, %edx
    movl -28(%rbp), %eax
    addl %eax, %edx
    movl -24(%rbp), %eax
    addl %eax, %edx
    movl -20(%rbp), %eax
    addl %eax, %edx
    movl -16(%rbp), %eax
    addl %eax, %edx
    movl -12(%rbp), %eax
    addl %eax, %edx
    movl -8(%rbp), %eax
    addl %edx, %eax
    movl %eax, -4(%rbp)
    nop
    popq %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0:
    .size main, .-main
    .ident "GCC:␣(Ubuntu␣11.4.0-1ubuntu1~22.04)␣11.4.0"
    .section .note.GNU-stack,"",@progbits
    .section .note.gnu.property,"a"
    .align 8
    .long 1f - 0f
    .long 4f - 1f
    .long 5
0:
    .string "GNU"
```

```
1:
    .align 8
    .long 0xc0000002
    .long 3f - 2f
2:
    .long 0x3
3:
    .align 8
4:
```

Correspondências entre C e Assembly

Declaração de Variáveis

No código em C, as variáveis são declaradas como inteiros:

Listing 3: Código em C

```
int a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s;
```

No assembly, as variáveis são alocadas na pilha do frame de ativação da função. Por exemplo:

Listing 4: Código em Assembly

```
movl -76(%rbp), %edx
```

Explicação:

- **movl**: Instrução assembly que move dados de uma origem para um destino. Neste caso, move um valor de 32 bits.
- **-76(%rbp)**: Indica que estamos acessando um endereço na pilha, deslocado -76 bytes a partir da base do quadro de pilha atual (%rbp). Isso sugere que -76(%rbp) provavelmente está armazenando algum tipo de variável local ou parâmetro.
- **%edx**: Registrador de dados de 32 bits que será usado para armazenar o valor lido do endereço -76(%rbp).

O %rbp é o registrador de base da pilha, que aponta para o início do quadro de pilha atual na execução de uma função. O %edx é um dos registradores de propósito geral usados para operações aritméticas e de manipulação de dados em assembly x86-64.

Este trecho de código pode ser parte de uma função assembly que está lendo um valor específico localizado na pilha para ser usado posteriormente no programa. `rbp` (base pointer).

Atribuições

A atribuição no C:

Listing 5: Código em C

```
s = a + b + c + d + e + f + g + h + i + j + k + l + m + n + o + p + q + r;
```

No assembly, isso é desdobrado em múltiplas instruções de soma e movimentação. Cada operação de soma é realizada individualmente e acumulada em registradores:

Listing 6: Código em Assembly

```
movl -76(%rbp), %edx
movl -72(%rbp), %eax
addl %eax, %edx
movl -68(%rbp), %eax
addl %eax, %edx
...
movl -8(%rbp), %eax
addl %edx, %eax
movl %eax, -4(%rbp)
```

Explicação:

Cada deslocamento como '-76(%rbp)', '-72(%rbp)', etc., incrementa de 4 em 4 devido ao tamanho dos dados na arquitetura x86-64. Cada variável de 32 bits (4 bytes) na pilha é acessada com um deslocamento de 4 bytes. As instruções `movl` movem valores das variáveis locais para registradores, enquanto as instruções `addl` realizam as somas acumulativas desses valores. O registrador `%edx` é usado para acumular os resultados parciais das somas até a última operação, onde o resultado final é movido para a variável 's' na pilha (`-4(%rbp)`). Esse processo exemplifica como o compilador traduz a operação de

soma em C para uma série de operações individuais em assembly, otimizando a utilização dos registradores disponíveis para as operações aritméticas.

Comandos e Chamadas de Função

A função principal `main()` em C:

Listing 7: Código em C

```
void main() {
    ...
}
```

No assembly, a definição e o início da função são indicados por:

Listing 8: Código em Assembly

```
.globl main
.type main, @function
main:
.LFB0:
        .cfi_startproc
```

- `.globl main`: Declaração de que `main` é um símbolo global, permitindo que seja acessado de outros módulos ou arquivos.
- `.type main, @function`: Indica que `main` é uma função.
- `main:` Define o início da função `main`.

O prólogo da função, que configura o stack frame, é representado por:

Listing 9: Código em Assembly

```
.LFB0:
        .cfi_startproc
endbr64
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
```

- `endbr64`: Instrução de prevenção de supressão de retorno.

- `pushq %rbp`: Salva o valor atual do registrador de base da pilha (`%rbp`) na pilha.

- `movq %rsp, %rbp`: Move o ponteiro de pilha atual (`%rsp`) para o registrador de base da pilha (`%rbp`), estabelecendo o novo frame de pilha.

- Diretivas `.cfi_*`: Instruções de informações de quadro de pilha para o depurador.

O epílogo da função, que limpa o stack frame e retorna, é representado por:

Listing 10: Código em Assembly

```
popq %rbp
.cfi_def_cfa 7, 8
ret
.cfi_endproc
```

- `popq %rbp`: Restaura o valor anterior do registrador de base da pilha (`%rbp`) a partir da pilha.

- `ret`: Retorna da função, transferindo o controle de volta para onde a função foi chamada.

- Diretivas `.cfi_*`: Instruções de informações de quadro de pilha para o depurador, indicando o fim do procedimento.

Essas instruções e diretivas são cruciais para a definição e execução de funções em assembly, garantindo a correta manipulação do quadro de pilha e o fluxo de controle adequado durante a execução do programa.

Exercício 2 (EP2-2)

Código em C

Listing 11: Código em C

```
#include <stdio.h>


void main() {
    int a, b, c, d;
    if (a > (b + c)) {
        d++;
        printf("algo");
```

```
    } else {
        c++;
        puts("outro");
    }
    a += 5;
    printf("xeque");
}
```

Código em Assembly

Listing 12: Código em Assembly

```
.file "ep2-2.c"
.text
.section .rodata
.LC0:
        .string "algo"
.LC1:
        .string "outro"
.LC2:
        .string "xeque"
.text
.globl main
.type main, @function
main:
.LFB0:
        .cfi_startproc
        endbr64
        pushq %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq %rsp, %rbp
        .cfi_def_cfa_register 6
        subq $16, %rsp
        movl -16(%rbp), %edx
```

```
        movl -12(%rbp), %eax
        addl %edx, %eax
        cmpl %eax, -8(%rbp)
        jle .L2
        addl $1, -4(%rbp)
        leaq .LC0(%rip), %rax
        movq %rax, %rdi
        movl $0, %eax
        call printf@PLT
        jmp .L3
.L2:
        addl $1, -12(%rbp)
        leaq .LC1(%rip), %rax
        movq %rax, %rdi
        call puts@PLT
.L3:
        addl $5, -8(%rbp)
        leaq .LC2(%rip), %rax
        movq %rax, %rdi
        movl $0, %eax
        call printf@PLT
        nop
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size main, .-main
        .ident "GCC:␣(Ubuntu␣11.4.0-1ubuntu1~22.04)␣11.4.0"
        .section .note.GNU-stack,"",@progbits
        .section .note.gnu.property,"a"
        .align 8
        .long 1f - 0f
        .long 4f - 1f
```

```
        .long 5
0:
        .string "GNU"
1:
        .align 8
        .long 0xc0000002
        .long 3f - 2f
2:
        .long 0x3
3:
        .align 8
4:
```

Correspondências entre C e Assembly

Declaração de Variáveis

No código em C, as variáveis são declaradas como inteiros:

Listing 13: Código em Assembly

```
int a, b, c, d;
```

No assembly, as variáveis são alocadas na pilha do frame de ativação da função. Por exemplo:

Listing 14: Código em Assembly

```
subq $16, %rsp
```

Explicação:

- `subq $16, %rsp`: Esta instrução subtrai 16 bytes do registrador de ponteiro de pilha (`%rsp`). Isso reserva espaço na pilha para as variáveis locais `a, b, c` e `d`. Em arquiteturas x86-64, a pilha cresce para baixo (do endereço de memória mais alto para o mais baixo), portanto, subtrair do ponteiro de pilha (`%rsp`) move o topo da pilha para baixo, aumentando o espaço disponível para as variáveis locais dentro do frame de ativação da função.

Essa alocação de espaço na pilha é essencial para armazenar variáveis locais durante a execução da função em assembly, garantindo que elas tenham espaço suficiente para armazenar seus valores temporários e permitindo acesso eficiente durante a execução do programa.

Condicional e Atribuições

A condicional no C:

Listing 15: Código em Assembly

```
if (a > (b + c)) {
    d++;
    printf("algo");
} else {
    c++;
    puts("outro");
}
```

No assembly, isso é desdobrado em múltiplas instruções de comparação e salto. Por exemplo:

Listing 16: Código em Assembly

```
movl -16(%rbp), %edx
movl -12(%rbp), %eax
addl %edx, %eax
cmpl %eax, -8(%rbp)
jle .L2
addl $1, -4(%rbp)
leaq .LC0(%rip), %rax
movq %rax, %rdi
movl $0, %eax
call printf@PLT
jmp .L3
.L2:
addl $1, -12(%rbp)
leaq .LC1(%rip), %rax
```

```
movq %rax, %rdi
call puts@PLT
.L3:
```

Explicação:

- `movl -16(%rbp), %edx`: Move o valor da variável `a` para o registrador `%edx`.
- `movl -12(%rbp), %eax`: Move o valor da variável `b` para o registrador `%eax`.
- `addl %edx, %eax`: Soma os valores de `a` e `b`, armazenando o resultado em `%eax`.
- `cmpl %eax, -8(%rbp)`: Compara o valor em `%eax` (resultado de `a + b`) com o valor da variável `c`.
- `jle .L2`: Salta para o rótulo `.L2` se `a` não for maior que `b + c`.
- `addl $1, -4(%rbp)`: Incrementa a variável `d` se a condição for verdadeira (`a > b + c`).
- `leaq .LC0(%rip), %rax`: Carrega o endereço da string "algo" para o registrador `%rax`.
- `movq %rax, %rdi`: Passa o endereço da string como argumento para a função `printf`.
- `call printf@PLT`: Chama a função `printf` para imprimir "algo".
- `jmp .L3`: Salta para `.L3` para evitar a execução do bloco `else`.
- `.L2`: Rótulo para o início do bloco `else`, caso `a <= b + c`.
- `addl $1, -12(%rbp)`: Incrementa a variável `c` se a condição for falsa (`a <= b + c`).
- `leaq .LC1(%rip), %rax`: Carrega o endereço da string "outro" para o registrador `%rax`.
- `movq %rax, %rdi`: Passa o endereço da string como argumento para a função `puts`.
- `call puts@PLT`: Chama a função `puts` para imprimir "outro".
- `.L3`: Rótulo para o final do bloco condicional.

Esse conjunto de instruções exemplifica como o compilador transforma uma estrutura condicional simples em C em operações de comparação, saltos condicionais e chamadas de função em assembly, mantendo o controle de fluxo e realizando as operações necessárias com eficiência na arquitetura x86-64.

Atribuições e Funções

A atribuição no C:

Listing 17: Código em Assembly

```
a += 5;
```

```
printf("xeque");
```

No assembly, isso é representado por:

Listing 18: Código em Assembly

```
addl $5, -8(%rbp)
leaq .LC2(%rip), %rax
movq %rax, %rdi
movl $0, %eax
call printf@PLT
```

Explicação:

- `addl $5, -8(%rbp)`: Esta instrução adiciona o valor 5 à variável `a`, que está localizada no endereço `-8(%rbp)` dentro do frame de ativação da função.
- `leaq .LC2(%rip), %rax`: Carrega o endereço da string "xeque" para o registrador `%rax`.
- `movq %rax, %rdi`: Passa o endereço da string como argumento para a função `printf`, armazenando-o no registrador `%rdi` conforme exigido pela convenção de chamada da função em sistemas x86-64.
- `movl $0, %eax`: Limpa o registrador `%eax` para indicar que não há argumentos de ponto flutuante passados para `printf`.
- `call printf@PLT`: Chama a função `printf` para imprimir a string "xeque".

Essas instruções demonstram como o compilador traduz operações simples de atribuição e chamadas de função em C para operações específicas em assembly, garantindo que as variáveis sejam manipuladas corretamente e que as chamadas de função sejam feitas de acordo com a convenção de chamada da plataforma.

Exercício 3 - (EP2-3)

Código em C

Listing 19: Código em C

```
#include <stdio.h>
```

```
void main() {
    int a, b, c, d;
    while (a > (b + c)) {
        a++;
        printf("algo");
    }
    c++;
    puts("outro");
}
```

Código em Assembly

Listing 20: Código em Assembly

```
.file "ep2-3.c"
.text
.section .rodata
.LC0:
        .string "algo"
.LC1:
        .string "outro"
.text
.globl main
.type main, @function
main:
.LFB0:
        .cfi_startproc
        endbr64
        pushq %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq %rsp, %rbp
        .cfi_def_cfa_register 6
        subq $16, %rsp
        jmp .L2
```

```
.L3:
        addl $1, -12(%rbp)
        leaq .LC0(%rip), %rax
        movq %rax, %rdi
        movl $0, %eax
        call printf@PLT
.L2:
        movl -8(%rbp), %edx
        movl -4(%rbp), %eax
        addl %edx, %eax
        cmpl %eax, -12(%rbp)
        jg .L3
        addl $1, -4(%rbp)
        leaq .LC1(%rip), %rax
        movq %rax, %rdi
        call puts@PLT
        nop
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size main, .-main
        .ident "GCC: (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0"
        .section .note.GNU-stack,"",@progbits
        .section .note.gnu.property,"a"
        .align 8
        .long 1f - 0f
        .long 4f - 1f
        .long 5
0:
        .string "GNU"
1:
        .align 8
```

```
        .long 0xc0000002
        .long 3f - 2f
2:
        .long 0x3
3:
        .align 8
4:
```

Correspondências entre C e Assembly

Declaração de Variáveis

No código em C, as variáveis são declaradas como inteiros:

Listing 21: Código em Assembly

```
int a, b, c, d;
```

No assembly, as variáveis são alocadas na pilha do frame de ativação da função. Por exemplo:

Listing 22: Código em Assembly

```
subq $16, %rsp
```

Explicação:

- `subq $16, %rsp`: Esta instrução subtrai 16 bytes do registrador de ponteiro de pilha (`%rsp`). Em arquiteturas x86-64, a pilha cresce para baixo (do endereço de memória mais alto para o mais baixo). Portanto, subtrair do ponteiro de pilha move o topo da pilha para baixo, reservando espaço para as variáveis locais `a, b, c` e `d` no frame de ativação da função.

Essa alocação na pilha é essencial para que as variáveis locais possam ser acessadas e manipuladas durante a execução da função em assembly, seguindo as convenções de chamada e garantindo o correto gerenciamento da memória durante a execução do programa.

Laço While e Atribuições

O laço `while` no C:

Listing 23: Código em Assembly

```
while (a > (b + c)) {
    a++;
    printf("algo");
}
```

No assembly, isso é desdobrado em múltiplas instruções de comparação e salto. Por exemplo:

Listing 24: Código em Assembly

```
jmp .L2
.L3:
        addl $1, -12(%rbp)
        leaq .LC0(%rip), %rax
        movq %rax, %rdi
        movl $0, %eax
        call printf@PLT
.L2:
        movl -8(%rbp), %edx
        movl -4(%rbp), %eax
        addl %edx, %eax
        cmpl %eax, -12(%rbp)
        jg .L3
```

Explicação:

- `jmp .L2`: Salto inicial para verificar a condição do laço.

- `.L3`: Rótulo para o início do bloco do laço.

- `addl $1, -12(%rbp)`: Incrementa a variável `a` dentro do laço se a condição `a > (b + c)` for verdadeira.

- `leaq .LC0(%rip), %rax`: Carrega o endereço da string "algo" para o registrador `%rax`.

- `movq %rax, %rdi`: Passa o endereço da string como argumento para a função `printf`.

- `movl $0, %eax`: Limpa o registrador `%eax` para indicar que não há argumentos de ponto flutuante passados para `printf`.

- `call printf@PLT`: Chama a função `printf` para imprimir a string "algo".

- `movl -8(%rbp), %edx`: Move o valor da variável `a` para o registrador `%edx`.

- `movl -4(%rbp), %eax`: Move o valor da expressão `b + c` para o registrador `%eax`.
- `addl %edx, %eax`: Soma os valores de `a` e `b + c`, armazenando o resultado em `%eax`.
- `cmpl %eax, -12(%rbp)`: Compara o valor em `%eax` (resultado de `a + b + c`) com o valor da variável `a`.
- `jg .L3`: Salta de volta para `.L3` se `a` ainda for maior que `b + c`, repetindo assim o laço.

Essas instruções exemplificam como o compilador transforma um laço `while` em C em uma estrutura de comparação, salto condicional e chamadas de função em assembly, mantendo o controle de fluxo e realizando operações iterativas conforme necessidade do programa.

Atribuições e Funções

A atribuição no C:

Listing 25: Código em Assembly

```
c++;
puts("outro");
```

No assembly, isso é representado por:

Listing 26: Código em Assembly

```
addl $1, -4(%rbp)
leaq .LC1(%rip), %rax
movq %rax, %rdi
call puts@PLT
```

Explicação:

- `addl $1, -4(%rbp)`: Esta instrução incrementa a variável `c` dentro do contexto do frame de ativação da função. O valor 1 é adicionado ao conteúdo de `-4(%rbp)`, que é onde a variável `c` está localizada.
- `leaq .LC1(%rip), %rax`: Carrega o endereço da string "outro" para o registrador `%rax`.
- `movq %rax, %rdi`: Passa o endereço da string como argumento para a função `puts`. O registrador `%rdi` é usado para armazenar o primeiro argumento para chamadas de função em sistemas x86-64.
- `call puts@PLT`: Chama a função `puts`, que imprime a string "outro" seguida por uma nova linha.

Essas instruções exemplificam como o compilador traduz operações simples de incremento e chamadas de função em C para operações específicas em assembly, garantindo que as variáveis sejam manipuladas corretamente e que as chamadas de função sejam feitas de acordo com a convenção de chamada da plataforma.

Conclusão

A análise dos códigos em C e Assembly nos três exercícios fornece insights valiosos sobre como o código fonte em C é traduzido para instruções em Assembly, destacando a complexidade das operações e a manipulação dos dados em nível de baixo nível.

No Exercício 1, observamos como as variáveis em C são alocadas na memória e manipuladas através de registradores em Assembly. A correspondência entre as operações em C e suas representações em Assembly demonstra a tradução direta das instruções de alto nível para operações de baixo nível.

O Exercício 2 apresenta um exemplo de estruturas de controle em C, mostrando como as condicionais e os loops são representados em Assembly através de instruções de comparação e salto condicional. A análise dessas estruturas revela a implementação eficiente de operações de controle de fluxo em nível de Assembly.

Por fim, o Exercício 3 destaca a importância da otimização do código em C para reduzir a complexidade das operações em Assembly. A comparação entre os dois níveis de linguagem ressalta a necessidade de compreensão profunda do funcionamento interno do processador para escrever códigos eficientes e otimizados.

Em resumo, a tradução de códigos em C para Assembly oferece uma visão detalhada do funcionamento interno do sistema computacional, destacando a interação entre software e hardware e a importância da otimização de código para melhorar o desempenho e a eficiência do sistema computacional como um todo.

---

*End of **EP#2***