Etapa 6: Geração de Código Assembly

Este relatório descreve a implementação da etapa de geração de código assembly no compilador desenvolvido para a disciplina de Compiladores. O objetivo desta etapa é transformar uma lista de instruções intermediárias (TAC - *Three Address Code*) em código de máquina (*assembly* x86_64 AT&T syntax), permitindo que o programa seja executado em uma arquitetura real após a montagem.

---

1. Ferramentas e Ambiente de Desenvolvimento

O desenvolvimento e a compilação foram realizados em um ambiente **Linux (Ubuntu 22.04 LTS)**, utilizando um conjunto de ferramentas padrão para a criação de compiladores em **C++**. A montagem e linkagem final do código assembly foram realizadas utilizando o **GCC (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0**.

---

2. Processo de Compilação e Execução

A geração de código é realizada na função **generateAsm(Tac* tacList)**, que recebe uma lista de instruções TAC e retorna uma *string* contendo o código assembly correspondente.

Seção de Dados

A abordagem adotada foi gerar código simples e direto. Assim, todos os símbolos da tabela - variáveis, constantes (literais) e temporários - foram declarados nesta seção, tendo o caractere '**_**' adicionado como prefixo. A função **generateDataSection()** percorre toda a tabela de símbolos (**symbolTable**) e define todos os símbolos necessários na seção **.data**:

- **Strings**: nomeadas com prefixo **.str_** e armazenadas com a diretiva **.string**.
- **Variáveis e arrays**: declarados com **.long**, com valor 0 por padrão ou com valores iniciais específicos.
- **Literais inteiros e caracteres**: também armazenados com **.long**, garantindo que todas as variáveis e literais usados no programa tenham um endereço fixo conhecido.

Seção de Código

A função **generateAsm** percorre a lista de instruções TAC e gera o código correspondente para cada tipo de TAC. As instruções tratadas incluem:

- **Atribuições e acesso a arrays**:
  - **MOVE, MOVE_IDX, IDX_ACCESS**: usam movl, leaq e indexação indireta via registradores (%rcx, %rax, %edx).

- **Controle de fluxo**:
  - **BEGIN_FUNC, END_FUNC**: demarcam o início e fim das funções (pushq %rbp, movq %rsp, %rbp, ret).
  - **LABEL, JUMP, IFZ**: geram saltos e rótulos com jmp, jz, e labels locais.

- **Entrada e saída**:
  - **READ**: usa scanf@PLT para leitura de inteiros.
  - **PRINT**: usa printf@PLT para inteiros e strings, e putchar@PLT para caracteres.

- **Operações aritméticas e lógicas**:
  - **ADD, SUB, MULT, DIV**: traduzidas para addl, subl, imull, idivl, com uso de %eax.
  - Comparações como **LESS, GREATER, EQ, DIF, AND, OR, NOT**: usam cmpl, setX, movzbl, e tratam o resultado como valor booleano em %eax.

- **Chamadas de função**:
  - **FUNC_CALL, RETURN, ARG**: realizam chamadas com call, manipulação de retorno via %eax, e passagem de argumentos por variáveis globais.

Decisões de Projeto

- Uso de endereçamento **RIP-relative**, facilitando compatibilidade com o modelo ELF de 64 bits.

- Não foram utilizadas **pilhas** para passagem de parâmetros e retorno. Ao invés disso, na geração da TAC, cada **TacType::ARG** tem uma referência tanto para o symbol que representa o valor da expressão a ser passada como argumento, mas também para o parâmetro da função ao qual lhe é correspondente. Com essa simplificação, a passagem de argumentos é feita através de uma atribuição, como se os parâmetros fossem **variáveis globais**.

- **Strings** são armazenadas com índices dinâmicos (**.str_0**, **.str_1**, etc), mapeadas na geração da seção de dados, via **stringIdxMap**, um **map** que, dada um symbol, retorna a string correspondente na seção de dados.

- O registrador **%eax** é usado como acumulador padrão, consistente com a arquitetura x86.

Considerações Finais

A implementação da geração de código assembly cumpre os requisitos da disciplina, sendo capaz de traduzir instruções TAC para um subconjunto útil da linguagem **assembly x86_64**.

O código gerado é funcional, suportando:

- Chamadas de função
- Controle de fluxo
- Entrada e saída
- Manipulação de vetores (arrays)
- Operações aritméticas

---

3. Testes e Validação

A seguir são apresentados os resultados da execução de programas de teste, demonstrando a corretude do código assembly gerado. Todos os arquivos .txt usados, tanto quanto os arquivos *.s* gerados por cada execução de testes podem ser encontrados na pasta **etapa6/tests/assembly/.**

Teste 1 (func_call.txt): Funções e argumentos

- **Código-fonte do teste:**

```
int var = 0;

int main () {
  print "Valor inicial: var = " var "\n";
  var = func1(func2(1), func3(2));
  print "var = " var "\n";
  var = func2(2);
  print "var = " var "\n";
  var = func0();
  print "Valor final: var =  " var "\n";
}

int func0() { return 0; }

int func1(int x, int y) { return x + y; }

int func2(int w) { return w; }

int func3(int v) { return v; }
```

- **Log de execução:**

```
Valor inicial: var = 0
var = 3
var = 2
Valor final: var =  0
```

Teste 2 (flux_control.txt): Estruturas de controle de fluxo

- **Código-fonte do teste:**

```
int number = 5;

int main () {
  print "number = " number "\n";

  if (number == 5) {
    print "IF1   -> number == 5\n";
  } else {
    print "ELSE1 -> number != 5\n";
  }

  if (number != 5) {
    print "IF2   -> number != 5\n";
  } else {
    print "ELSE2 -> number == 5\n";
  }

  do {
    print "DO number > 3 -> " "number = " number "\n";
    number = number - 1;
  } while (number > 3);

  while (number < 6) do {
    print "WHILE number < 6 -> " "number = " number "\n";
    number = number + 1;
  }
}
```

- **Log de execução:**

```
number = 5
IF1   -> number == 5
ELSE2 -> number == 5
DO number > 3 -> number = 5
DO number > 3 -> number = 4
WHILE number < 6 -> number = 3
WHILE number < 6 -> number = 4
WHILE number < 6 -> number = 5
```

Teste 3 (read.txt): Entrada e saída

- **Código-fonte do teste:**

```
int number = 1;

int main () {
  print "number value is: " number "\n";
  print "enter a new value: ";
  read number;
  print "number now has a new value: " number "
}
```

- **Log de execução:**

```
number value is: 1
enter a new value: 6
number now has a new value: 6
```

Teste 4 (arrays.txt): Manipulação de vetores (arrays)

- **Código-fonte do teste:**

```
int number[3] = 1, 2, 3;
int index = 1;
int value = 5555;

int main () {
  print "\n number[0] = " number[0];
  print "\n number[1] = " number[1];
  print "\n number[2] = " number[2] "\n";

  number[index] = value;

  print "\n number[0] = " number[0];
  print "\n number[1] = " number[1];
  print "\n number[2] = " number[2] "\n";
}
```

- **Log de execução:**

```
number[0] = 1
number[1] = 2
number[2] = 3

number[0] = 1
number[1] = 5555
number[2] = 3
```

Teste 5 (operations.txt): Operações aritméticas

- **Código-fonte do teste:**

```
int number = 5;

int main () {
  print number "\n";      // 5
  number = number + 1;
  print number "\n";      // 6
  number = number - 4;
  print number "\n";      // 2
  number = number * 6;
  print number "\n";      // 12
  number = number / 4;
  print number "\n";      // 3

  if (number < 2) {
    print "< TRUE\n";
  } else {
    print "< FALSE\n";
  }

  if (number > 2) print "> TRUE\n";

  if (number <= 4) print "<= TRUE\n";

  if (number >= 4) print ">= TRUE\n";

  if (number == 3) print "== TRUE\n";

  if (number != 55) print "!= TRUE\n";

  if (number == 3 & number == 3) print "& TRUE\n";

  if (number == 2 | number == 4) print "| TRUE\n";

  if (~(number != 3)) print "~ TRUE\n";
}
```

- **Log de execução:**

```
5
6
2
12
3
< FALSE
> TRUE
<= TRUE
== TRUE
!= TRUE
& TRUE
~ TRUE
```

Teste 6 (sample.txt): Arquivo exemplo disponível no site da disciplina

- **Log de execução**

```
075Digite um numero:
4
...................Dobrando algumas vezes y fica 512
A era=15
OK!
```