Bài 12
Sinh mã đích

Nguyễn Thị Thu Hương

Nội dung

- Tổng quan về sinh mã đích
- Máy ngăn xếp
  - Tổ chức bộ nhớ
  - Bộ lệnh
- Sinh mã cho các lệnh cơ bản
- Xây dựng bảng ký hiệu
  - Biến
  - Tham số
  - Hàm, thủ tục và chương trình

Chương trình đích

- Viết trên một ngôn ngữ trung gian
- Là dạng Assembly của máy giả định (máy ảo)
- Máy ảo làm việc với bộ nhớ stack
- Việc thực hiện chương trình thông qua một interpreter
- Interpreter mô phỏng hành động của máy ảo thực hiện tập lệnh assembly của nó

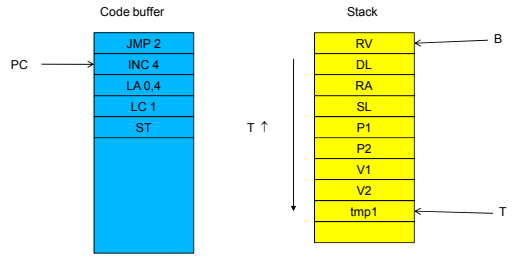
Chương trình đích được dịch từ

- Mã nguồn
- Mã trung gian

Máy ngăn xếp

- Máy ngăn xếp là một hệ thống tính toán
  - Sử dụng ngăn xếp để lưu trữ các kết quả trung gian của quá trình tính toán
  - Kiến trúc đơn giản
  - Bộ lệnh đơn giản
- Máy ngăn xếp có hai vùng bộ nhớ chính
  - Khối lệnh: chứa mã thực thi của chương trình
  - Ngăn xếp: sử dụng để lưu trữ các kết quả trung gian

Máy ngăn xếp



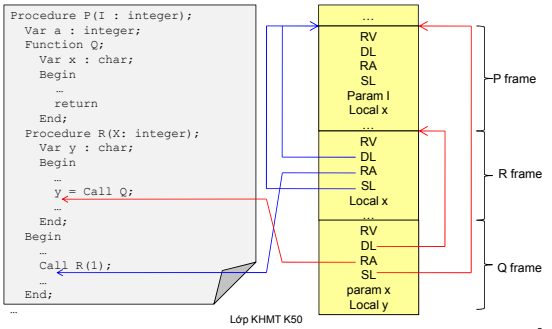
Máy ngăn xếp

- Thanh ghi
  - PC (program counter): con trỏ lệnh trỏ tới lệnh hiện tại đang thực thi trên bộ đệm chương trình
  - B (base) : con trỏ trỏ tới địa chỉ gốc của vùng nhớ cục bộ. Các biến cục bộ được truy xuất gián tiếp qua con trỏ này
  - T (top); trỏ tới đỉnh của ngăn xếp

Máy ngăn xếp

- Bản hoạt động (activation record/stack frame)
  - Không gian nhớ cấp phát cho mỗi chương trình con (hàm/thủ tục/chương trình chính) khi chúng được kích hoạt
    - Lưu giá trị tham số
    - Lưu giá trị biến cục bộ
    - Lưu các thông tin khác
      - Giá trị trả về của hàm – RV
      - Địa chỉ cơ sở của bản hoạt động của chương trình con gọi tới (caller) – DL
      - Địa chỉ lệnh quay về khi kết thúc chương trình con – RA
      - Địa chỉ cơ sở của bản hoạt động của chương trình con bao ngoài – SL
  - Một chương trình con có thể có nhiều bản hoạt động

Máy ngăn xếp



Máy ngăn xếp

- RV (return value): Lưu trữ giá trị trả về cho mỗi hàm
- DL (dynamic link): Sử dụng để hồi phục ngữ cảnh của chương trình gọi (caller) khi chương trình được gọi (callee) kết thúc
- RA (return address): Sử dụng để tìm tới lệnh tiếp theo của caller khi callee kết thúc
- SL (static link): Sử dụng để truy nhập các biến phi cục bộ

Máy ngăn xếp

- Bộ lệnh

op	p	q

LA	Load Address	t:=t+1; s[t]:=base(p)+q;
LV	Load Value	t:=t+1; s[t]:=s[base(p)+q];
LC	Load Constant	t:=t+1; s[t]:=q;
LI	Load Indirect	s[t]:=s[s[t]];
INT	Increment T	t:=t+q;
DCT	Decrement T	t:=t-q;

Máy ngăn xếp

- Bộ lệnh

op	p	q

J	Jump	pc:=q;
FJ	False Jump	if s[t]=0 then pc:=q; t:=t-1;
HL	Halt	Halt
ST	Store	s[s[t-1]]:=s[t]; t:=t-2;
CALL	Call	s[t+2]:=b; s[t+3]:=pc; s[t+4]:=base(p); b:=t+1; pc:=q;
EP	Exit Procedure	t:=b-1; pc:=s[b+2]; b:=s[b+1];
EF	Exit Function	t:=b; pc:=s[b+2]; b:=s[b+1];

Máy ngăn xếp

- Bộ lệnh

op	p	q

RC	Read Character	read one character into s[s[t]]; t:=t-1;
RI	Read Integer	read integer to s[s[t]]; t:=t-1;
WRC	Write Character	write one character from s[t]; t:=t-1;
WRI	Write Integer	write integer from s[t]; t:=t-1;
WLN	New Line	CR & LF

Máy ngăn xếp

- Bộ lệnh

op	p	q

AD	Add	t:=t-1; s[t]:=s[t]+s[t+1];
SB	Subtract	t:=t-1; s[t]:=s[t]-s[t+1];
ML	Multiply	t:=t-1; s[t]:=s[t]*s[t+1];
DV	Divide	t:=t-1; s[t]:=s[t]/s[t+1];
NEG	Negative	s[t]:=-s[t];
CV	Copy Top of Stack	s[t+1]:=s[t]; t:=t+1;

Máy ngăn xếp

- Bộ lệnh

op	p	q

EQ	Equal	t:=t-1; if s[t] = s[t+1] then s[t]:=1 else s[t]:=0;
NE	Not Equal	t:=t-1; if s[t] != s[t+1] then s[t]:=1 else s[t]:=0;
GT	Greater Than	t:=t-1; if s[t] > s[t+1] then s[t]:=1 else s[t]:=0;
LT	Less Than	t:=t-1; if s[t] < s[t+1] then s[t]:=1 else s[t]:=0;
GE	Greater or Equal	t:=t-1; if s[t] >= s[t+1] then s[t]:=1 else s[t]:=0;
LE	Less or Equal	t:=t-1; if s[t] <= s[t+1] then s[t]:=1 else s[t]:=0;

Xây dựng bảng ký hiệu

- Bổ sung thông tin cho biến
  - Vị trí trên frame
  - Phạm vi
- Bổ sung thông tin cho tham số
  - Vị trí trên frame
  - Phạm vi
- Bổ sung thông tin cho hàm/thủ tục/chương trình
  - Địa chỉ bắt đầu
  - Kích thước của frame
  - Số lượng tham số của hàm/thủ tục

Xây dựng bảng ký hiệu

- Bổ sung thông tin cho biến
  - Vị trí trên frame (vị trí tính từ base của frame)
  - Phạm vi

```
struct VariableAttributes_ {
  Type *type;
  struct Scope_ *scope;
  int localOffset;
};
```


Xây dựng bảng ký hiệu

- Bổ sung thông tin cho tham số
  - Vị trí trên frame
  - Phạm vi

```
struct ParameterAttributes_ {
  enum ParamKind kind;
  Type* type;
  struct Scope_ *scope;
  int localOffset;
};
```


Xây dựng bảng ký hiệu

- Bổ sung thông tin cho phạm vi
  - Kích thước frame

```
struct Scope_ {
  ObjectNode *objList;
  Object *owner;
  struct Scope_ *outer;
  int frameSize;
};
```


Xây dựng bảng ký hiệu

- Bổ sung thông tin cho hàm
  - Vị trí
  - Số lượng tham số

```
struct FunctionAttributes_ {
  struct ObjectNode_ *paramList;
  Type* returnType;
  struct Scope_ *scope;

  int paramCount;
  CodeAddress codeAddress;
};
```

Xây dựng bảng ký hiệu

- Bổ sung thông tin cho thủ tục
  - Vị trí
  - Số lượng tham số

```
struct ProcedureAttributes_ {
  struct ObjectNode_ *paramList;
  struct Scope_* scope;

  int paramCount;
  CodeAddress codeAddress;
};
```


Xây dựng bảng ký hiệu

- Bổ sung thông tin cho chương trình
  - Vị trí

```
struct ProgramAttributes_ {
  struct Scope_ *scope;
  CodeAddress codeAddress;
};
```


int sizeOfType(type* t)

- Trả về số ngăn nhớ trên stack mà một biến thuộc kiểu đó sẽ chiếm.

void declareObject(Object* obj)

- Đối tượng toàn cục
  - Đưa vào symtab->GlobalObjectList
- Đối tượng khác:
  - Đưa vào symtab->currentScope->objList
  - Variable:
    - Cập nhật scope = currentScope
    - Cập nhật localOffset = currentScope->frameSize
    - Tăng kích thước frameSize

void declareObject(Object* obj)

- Đối tượng khác:
  - Parameter
    - Cập nhật scope = currentScope
    - Cập nhật localOffset = currentScope->frameSize
    - Tăng kích thước frameSize
    - Cập nhật paramList của owner
    - Tăng paramCount của owner.

void declareObject(Object* obj)

- Đối tượng khác:
  - Function
    - Cập nhật outer = currentScope
  - Procedure
    - Cập nhật outer = currentScope

kplrun

- Là bộ thông dịch cho máy ngăn xếp

```
$ kplrun <source> [-s=stack-size] [-c=code-size] [-debug] [-dump]
```

- Tùy chọn –s: định nghĩa kích thước stack
- Tùy chọn –c: định nghĩa kích thước tối đa của mã nguồn
- Tùy chọn –dump: In mã ASM
- Tùy chọn –debug: chế độ gỡ rối

kplrun

- Tùy chọn –debug: chế độ gỡ rối
  - a: địa chỉ tuyệt đối của địa chỉ tương đối (level, offset)
  - v: giá trị tại địa chỉ tương đối (level,offset)
  - t: giá trị đầu ngăn xếp
  - c: thoát khỏi chế độ gỡ rối

Instructions.c

```
enum OpCode {
  OP_LA,   // Load Address:
  OP_LV,   // Load Value:
  OP_LC,   // load Constant
  OP_LI,   // Load Indirect
  OP_INT,  // Increment t
  OP_DCT,  // Decrement t
  OP_J,    // Jump
  OP_FJ,   // False Jump
  OP_HL,   // Halt
  OP_ST,   // Store
  OP_CALL, // Call
  OP_EP,   // Exit Procedure
  OP_EF,   // Exit Function
```

```
  OP_RC,   // Read Char
  OP_RI,   // Read Integer
  OP_WRC,  // Write Char
  OP_WRI,  // Write Int
  OP_WLN,  // WriteLN
  OP_AD,   // Add
  OP_SB,   // Substract
  OP_ML,   // Multiple
  OP_DV,   // Divide
  OP_NEG,  // Negative
  OP_CV,   // Copy Top
  OP_EQ,   // Equal
  OP_NE,   // Not Equal
  OP_GT,   // Greater
  OP_LT,   // Less
  OP_GE,   // Greater or Equal
  OP_LE,   // Less or Equal
  OP_BP    // Break point.
};
```


Instructions.c

```
struct Instruction_ {
  enum OpCode op;
  WORD p;
  WORD q;
};


struct CodeBlock_ {
  Instruction* code;
  int codeSize;
  int maxSize;
};
```

```
CodeBlock* createCodeBlock(int maxSize);
void freeCodeBlock(CodeBlock* codeBlock);
void printInstruction(Instruction* instruction);
void printCodeBlock(CodeBlock* codeBlock);

void loadCode(CodeBlock* codeBlock, FILE* f);
void saveCode(CodeBlock* codeBlock, FILE* f);

int emitLA(CodeBlock* codeBlock, WORD p, WORD q);
int emitLV(CodeBlock* codeBlock, WORD p, WORD q);
int emitLC(CodeBlock* codeBlock, WORD q);
….
int emitLT(CodeBlock* codeBlock);
int emitGE(CodeBlock* codeBlock);
int emitLE(CodeBlock* codeBlock);

int emitBP(CodeBlock* codeBlock);
```


codegen.c

```
void initCodeBuffer(void);
void printCodeBuffer(void);
void cleanCodeBuffer(void);
int serialize(char* fileName);

int genLA(int level, int offset);
int genLV(int level, int offset);
int genLC(WORD constant);
….
int genLT(void);
int emitGE(void);
int emitLE(void);
```


Sinh mã lệnh gán

V := exp

```
<code of l-value v>  // đẩy địa chỉ của v lên stack
<code of exp>        // đẩy giá trị của exp lên stack
ST
```

Sinh mã lệnh if

If <dk> Then statement;

```
  <code of dk>     // đẩy giá trị điều kiện dk lên stack
  FJ L
  <code of statement>
L:
  …
```

If <dk> Then st1 Else st2;

```
  <code of dk>     // đẩy giá trị điều kiện dk lên stack
  FJ L1
  <code of st1>
  J L2
L1:
  <code of st2>
L2:
  …
```


Sinh mã lệnh while

While <dk> Do statement

```
L1:
  <code of dk>
  FJ L2
  <code of statement>
  J L1
L2:
  …
```


Sinh mã lệnh for

For v := exp1 to exp2 do statement

```
  <code of l-value v>
  CV   // nhân đôi địa chỉ của v
  <code of exp1>
  ST   // lưu giá trị đầu của v
L1:
  CV
  LI   // lấy giá trị của v
  <code of exp2>
  LE
  FJ L2
  <code of statement>
  CV;CV;LI;LC 1;AD;ST;  // Tăng v lên 1
  J L1
L2:
  DCT 1
  …
```


Lấy địa chỉ/giá trị biến

- Khi lấy địa chỉ/giá trị một biến cần tính đến phạm vi của biến
  - Biến cục bộ được lấy từ frame hiện tại
  - Biến phi cục bộ được lấy theo các StaticLink với cấp độ lấy theo "độ sâu" của phạm vi hiện tại so với phạm vi của biến

```
computeNestedLevel(Scope* scope)
```

Lấy địa chỉ của tham số hình thức

- Khi **`LValue`** là tham số
- Cũng cần tính độ sâu như biến
  - Nếu là tham trị: địa chỉ cần lấy chính là địa chỉ của tham trị
  - Nếu là tham biến: vì giá trị của tham biến chính là địa chỉ muốn truy nhập, địa chỉ cần lấy chính là giá trị của tham biến.

Lấy giá trị của tham số hình thức

- Khi tính toán giá trị của **`Factor`**
- Cũng cần tính độ sâu như biến
  - Nếu là tham trị: giá trị của tham trị chính là giá trị cần lấy.
  - Nếu là tham biến: giá trị của tham số là địa chỉ của giá trị cần lấy.

Lấy địa chỉ của giá trị trả về của hàm

- Giá trị trả về luôn nằm ở offset 0 trên frame
- Chỉ cần tính độ sâu giống như với biến hay tham số hình thức

Sinh lời gọi hàm/thủ tục

- Lời gọi
  - Hàm gặp trong sinh mã cho **`factor`**
  - Thủ tục gặp trong sinh mã lệnh **`CallSt`**
- Trước khi sinh lời gọi hàm/thủ tục cần phải nạp giá trị cho các tham số hình thức bằng cách
  - Tăng giá trị T lên 4 (bỏ qua RV,DL,RA,SL)
  - Sinh mã cho k tham số thực tế
  - Giảm giá trị T đi 4 + k
  - Sinh lệnh CALL

Sinh mã cho lệnh CALL (p, q)

CALL (p, q)

```
s[t+2]:=b;                  // Lưu lại dynamic link
s[t+3]:=pc;                 // Lưu lại return address
s[t+4]:=base(p); // Lưu lại static link
b:=t+1;          // Base mới và return value
pc:=q;           // địa chỉ lệnh mới
```

Giả sử cần sinh lệnh CALL cho hàm/thủ tục A
Lệnh CALL(p, q) có hai tham số:

p: Độ sâu của lệnh CALL, chứa static link.
Base(p) = base của frame chương trình con chứa khai báo của A.

q: Địa chỉ lệnh mới
q + 1 = địa chỉ đầu tiên của dãy lệnh cần thực hiện khi gọi A.

Hoạt động khi thực hiện lệnh CALL(p, q)

1. Điều khiển `pc` chuyển đến địa chỉ bắt đầu của chương trình con `/* pc = p */`
2. `pc` tăng thêm 1 `/* pc ++ */`
3. Lệnh đầu tiên thông thường là lệnh nhảy `J` để bỏ qua mã lệnh của các khai báo hàm/ thủ tục cục bộ trên `code buffer`.
4. Lệnh tiếp theo là lệnh `INT` tăng `T` đúng bằng kích thước frame để bỏ qua frame chứa vùng nhớ của các tham số và biến cục bộ.

Hoạt động khi thực hiện lệnh CALL(p, q)

5. Thực hiện các lệnh và stack biến đổi tương ứng.
6. Khi kết thúc
   1. Thủ tục (lệnh `EP`): toàn bộ frame được giải phóng, con trỏ `T` đặt lên đỉnh frame cũ.
   2. Hàm (lệnh `EF`): frame được giải phóng, chỉ chừa giá trị trả về tại `offset 0`, con trỏ `T` đặt lên đầu frame hiện thời (`offset 0`).

Sinh mã đích từ mã ba địa chỉ

- Bộ sinh mã trung gian đưa ra mã ba địa chỉ
- Tối ưu trên mã ba địa chỉ
- Từ mã ba địa chỉ đã tối ưu sinh ra mã đích phù hợp với một mô tả máy ảo