Code Generation

2012003686
임준수

1. Code Generation Design overview

1) Used Register

pc : program counter
현재 진행 되고 있는 instruction number 에 대해 저장하고 있다

sp : stack pointer
현재 stack의 끝을 가리키고 있다. Stack은 아래 방향으로 자란다. Tmp variable 저장 및 함수 호출 시 stack control 에 사용된다.

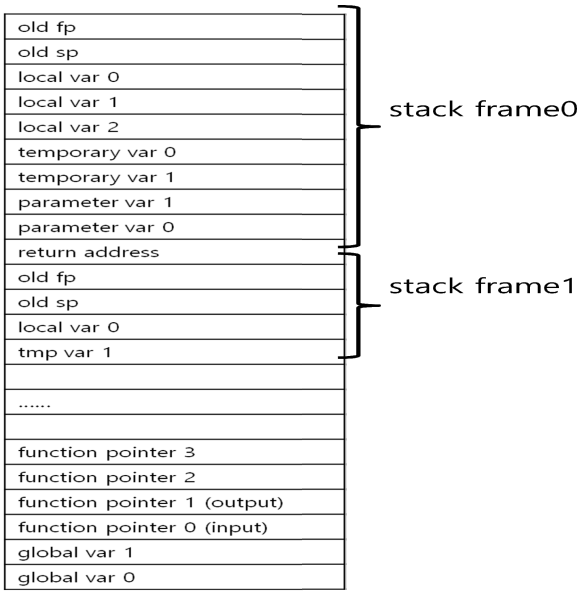
gp : global pointer
memory 의 맨 밑을 가리키고 있다. Global variable 들은 밑에서부터 쌓인다. Global variable과 function pointer 접근에 사용된다.

fp: frame pointer
현재 stack frame의 시작점을 가리키는 pointer 를 저장한다. Local variable 및 parameter 접근에 사용된다.

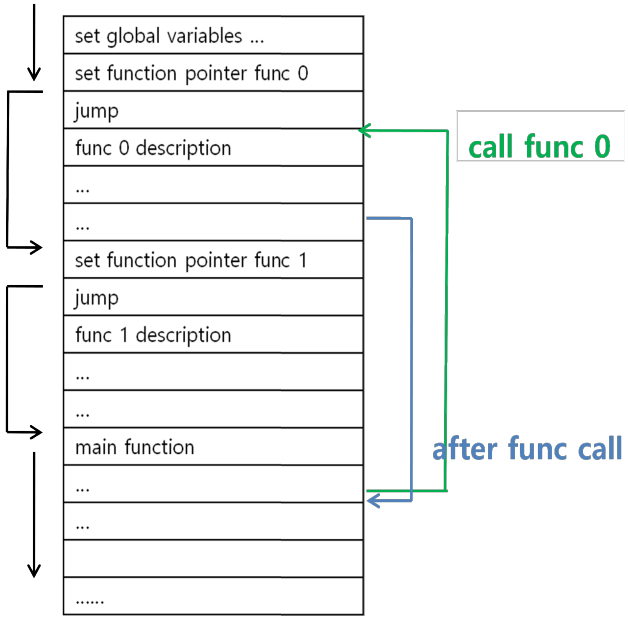
zero: have zero in this register
minus 값으로 바꾸는 연산을 register 간의 연산으로 용이하게 처리하기 위해서

ac: accumulator
ac1: accumulator1
연산 및 임시 저장을 위한 두개의 register

2) How stack are pushed



3) Tm code flow



2. Important Implementation.

Arithmetic operation, while loop, if statement는 tiny와 비슷한 구조를 띄고 있어 거의 고치지 않았기 때문에 code 설명은 생략하도록 하겠다.

가장 많은 차이를 보인 부분은 function과 array 이다.

1) Function

Function declaration

(beforeFuncDecl)
Function pointer를 setting 해주고, function을 실행하면 안 되기 때문에 jump를 넣을 공간을 남겨놓는다.

```
/* decl part */
void beforeFuncDecl(char *name) {
  BucketList l;
  int loc = emitSkip(0);
  emitRM("LDC", ac, loc+3, 0, "get function location");
  l = st_lookup(sc_top(), name);
  printf("in beforeFuncDecl : l->memloc - %d l->name - %s\n", l->memloc, l->name);
  emitRM("ST", ac, l->memloc, gp, "set function pointer");
  /* to do not execute function - change pc val */
  functionSkip = emitSkip(1);
}
```

(getFunc)
실제로 function이 declare 되는 부분이다. Function tree 는 child[0]: type child[1]: param child[2]: body 이기 때문에 아래와 같이 generate 가능하다. 이와 같이 하면 function 내의 statement들이 declare 된다.

```
/* tree->child[0] : type
 * tree->child[1] : parameters
 * tree->child[2] : body
 */
cGen(tree->child[2]);
```

(afterFuncDecl)
Function call 해서 function 이 실행된 이후 할 일들을 써주는 함수이다. 해당 함수가 하는 일은 아래와 같다.

```
/* after function done .. callee part */
void afterFuncDecl() {

  int loc = 0;
  /* restore sp */
  emitRM("LD",ac1,-1,fp,"get old sp");
  emitRM("LDA",sp,0,ac1,"restore old sp");
  /* save return addr in mp stack */
  emitRM("LD",ac1,1,fp,"get return addr");
  spController("ST",ac1,"save return addr in sp stack");
  /* restore fp */
  emitRM("LD",ac1,0,fp,"get old fp");
  emitRM("LDA",fp,0,ac1,"restore old fp");
  /* get return addr from stack and goto return addr */
  spController("LD",ac1,"get return addr from stack");
  emitRM("LDA",pc,0,ac1,"jump to return addr");

  /* set function skip command */
  loc = emitSkip(0);
  emitBackup(functionSkip);
  emitRM("LDC",pc,loc,0,"function skip");
  emitRestore();
}
```

Function call

- Save return location
- Set old fp and sp
- Move fp sp
- pc move

다음과 같은 작업을 진행한다.

```
void beforeFuncCall(TreeNode *tree) {
  TreeNode *params;
  Scope scope;
  int param_num;
  int mem_size;
  int param_offset = 0;
  int loc = 0;

  BucketList l;
  params = tree->child[0];

  scope = search_in_all_scope(tree->attr.name);

  param_num = scope->max_param_num;
  mem_size = scope->mem_size;

  setParamReverseOrder(params, param_num, 0);

  /* save return location */
  loc = emitSkip(0);

  emitRM("LDC",ac1,loc+10,0,"set return addr val");
  emitRM("ST",ac1,-(param_num),sp, "set return address");
  /* control linking ..... */
  emitRM("LDA",ac1,0,fp,"get old fp");
  emitRM("ST",ac1,-(param_num+1),sp, "set control link(old fp)");
  emitRM("LDA",ac1,0,sp,"get old sp");
  emitRM("ST",ac1,-(param_num+2),sp, "set control link2(old sp)");
  /* fp move */
  emitRM("LDA",fp,-(param_num+1),sp,"get new fp");
  /* set new mp */
  printf("scope num :%d\n", scope->mem_size);
  emitRM("LDC",ac,scope->mem_size,0,"set mp offset");
  emitRO("SUB",sp,fp,ac,"get new mp");
  /* pc mov to function call */
  l = st_lookup(sc_top(), tree->attr.name);
  emitRM("LD",pc,l->memloc,gp,"moving pc");
}
```

2) Array

Array offset
Global array variable 과 Local array variable 모두 위에서 시작해서 아래로 인덱스를 훑는 구조를 가지고있다.
해당 code 에서 모든 variable들은 emitHelper를 통해서 접근 한다. emitHelper는 LD 와 LDA 만 사용할 것을 권장한다.

	base	offset
global_var	gp	memloc
global_arr_var	gp + memloc	- array index
local_var	fp	memloc
local_arr_var	fp + memloc	- array index
param_var	fp	- param_opt
param_arr_var	fp – param_opt 에 들어있는 value	- array index

emitHelper 는 위의 표와 비교해 variable 에 접근한다. emitHelper 는 LD와 LDA만 써야된다.

Code는 위의 내용과 동일하게 구현 되어있다.

3. 컴파일러 과제를 마치며

소스코드가 어떻게 machine code로 바뀌는지 직접 체험해 볼 수 있는 좋은 기회였다. 보람찬 실습과제였다.