TM and Runtime Environment

무엇을 공부하나?

- 본 장에서는 Tiny Machine 혹은 TM이라고 불리는 작은 target machine에 대하여 공부한다.
- 우리가 제작할 C-Minus 컴파일러는 C-Minus 프로그램을 Tiny Machine이 수행하는 프로그램으로 변환한다.
- Tiny Machine은 명령어 메모리 `iMem[]`, 데이타 메모리 `dMem[]` 및 레지스터 `reg[]`로 구성되어 있고, 25개의 명령어를 수행한다.
- 컴파일러가 생성한 Tiny Machine 프로그램은 Tiny Machine 시뮬레이터(simulator)를 사용하여 수행할 수 있다.

강의 내용

- Tiny Machine(TM) 구조
- TM 수행기
- TM 명령어 집합
- TM 프로그램 보기
- TM 시뮬레이터(Simulator)

교재

– Kenneth C. Louden, *Compiler Construction Principles and Practice*, PWS Publishing Company, 1997. (8.7)

Tiny Machine(TM) 구조

- 명령어 메모리(instruction memory)
  - 주소 하나에 하나의 명령어(instruction)를 저장한다.
  - 배열 iMem[IADDR_SIZE]로 표현된다.
  - 기계 수행 전 컴파일된 프로그램 명령어가 여기에 저장된다.
- 데이타 메모리(data memory)
  - 주소 하나에 하나의 integer(32bit)를 저장한다.
  - 배열 dMem[DADDR_SIZE]로 표현된다.
  - 기계 수행 전 dMem[0]에는 (DADDR_SIZE-1)이 저장되어 있고, 그 외 데이타 메모리는 0으로 초기화되어 있다.
- 레지스터(register)
  - 하나의 레지스터는 하나의 integer(32bit)를 저장한다.
  - 배열 reg[32]로 표현된다.
  - 현재 32개(0~31)의 레지스터를 가지고 있다.
  - 레지스터 번호에서 pc=31, gp=30, fp=29, sp=28를 의미한다.
  - 기계 수행 전 모든 레지스터의 값은 0으로 초기화되어 있다.

TM 수행기

- 기계 초기화 및 프로그램 load
  ① 레지스터 reg[]를 0으로 초기화 한다.
  ② 명령어 메모리 iMem[]을 0으로 초기화 한다.
  ③ 데이타 메모리 dMem[]을 0으로 초기화 하고 dMem[0] = DADDR_SIZE - 1
  ④ 수행할 프로그램을 iMem[]에 load 한다.
- Fetch and execute loop
  ⑤ current_inst = iMem[reg[pc]]
  ⑥ current_inst가 HALT이면 기계를 종료시킨다.
  ⑦ reg[pc] = reg[pc] + 1
  ⑧ current_inst의 종류에 따라 각 명령어를 수행한다.
  ⑨ goto ⑤

TM 명령어 집합 - 1/4

- Arithmetic 명령어

명령어 형식	설명
`add r, s, t`	`reg[r] = reg[s] + reg[t]`
`sub r, s, t`	`reg[r] = reg[s] - reg[t]`
`mul r, s, t`	`reg[r] = reg[s] * reg[t]`
`div r, s, t`	`reg[r] = reg[s] / reg[t]` (/0 신호 발생)

TM 명령어 집합 - 1/4

- Relational 명령어

명령어 형식	설명
lt r, s, t	reg[r] = ? (reg[s] < reg[t]) : 1 : 0
le r, s, t	reg[r] = ? (reg[s] <= reg[t]) : 1 : 0
gt r, s, t	reg[r] = ? (reg[s] > reg[t]) : 1 : 0
ge r, s, t	reg[r] = ? (reg[s] >= reg[t]) : 1 : 0
eq r, s, t	reg[r] = ? (reg[s] == reg[t]) : 1 : 0
ne r, s, t	reg[r] = ? (reg[s] != reg[t]) : 1 : 0

TM 명령어 집합 - 2/4

- Load/Store 명령어

명령어 형식	설명
ld r, d(s)	reg[r] = dMem[d + reg[s]]
lda r, d(s)	reg[r] = d + reg[s]
ldc r, d	reg[r] = d
st r, d(s)	dMem[d + reg[s]] = reg[r]

TM 명령어 집합 - 3/4

- Jump 명령어

명령어 형식	설명
jlt r, d(s)	if(reg[r] < 0) reg[pc] = d + reg[s]
jle r, d(s)	if(reg[r] <= 0) reg[pc] = d + reg[s]
jgt r, d(s)	if(reg[r] > 0) reg[pc] = d + reg[s]
jge r, d(s)	if(reg[r] >= 0) reg[pc] = d + reg[s]
jeq r, d(s)	if(reg[r] == 0) reg[pc] = d + reg[s]
jne r, d(s)	if(reg[r] != 0) reg[pc] = d + reg[s]

TM 명령어 집합 - 4/4

- Stack, Input/Output 및 기타 명령어

명령어 형식	설명
push r	dMem[reg[sp]] = reg[r] reg[sp] = reg[sp] - 1
pop r	reg[sp] = reg[sp] + 1 reg[r] = dMem[reg[sp]]
in r	reg[r] <= in
out r	reg[r] => out
halt	기계의 수행을 종료함

TM 프로그램 보기 - fact.tm

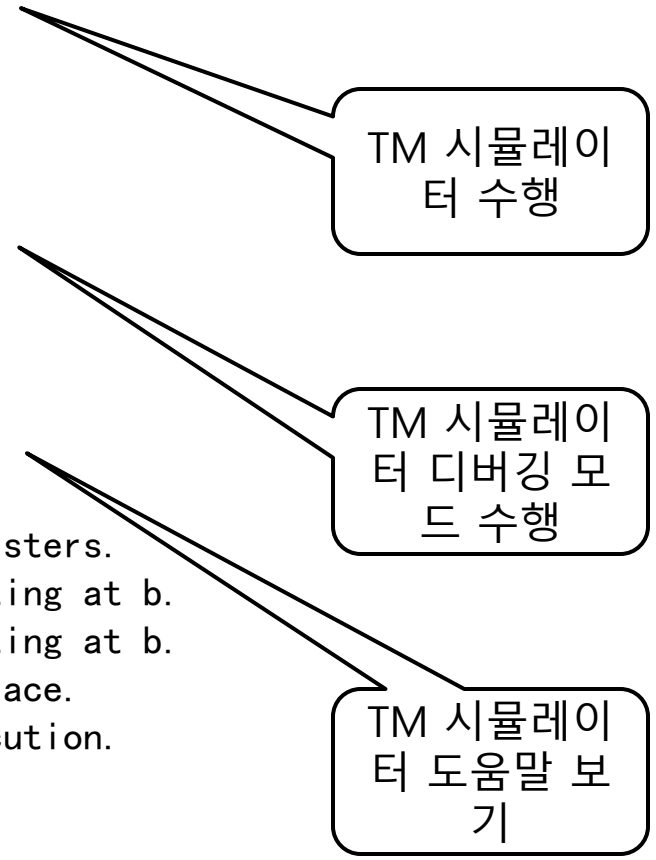
```
// ================================================
// fact.tm

0: in 0                    // r0 <= in
1: jle 0, 6(pc)            // if r0 <= 0 then goto 8
2: ldc 1, 1                // r1 = 1
3: ldc 2, 1                // r2 = 1
4: mul 1, 1, 0             // r1 = r1 * r0
5: sub 0, 0, 2             // r0 = r0 - r2
6: jne 0, -3(pc)           // if r0 != 0 then goto 4
7: out 1                   // r1 => out
8: halt                    // halt

// ================================================
```

TM 시뮬레이터(Simulator) - 시작하기

```
$ tm fact.tm
<= 10
=> 3628800
$ tm -d fact.tm
Tiny Machine 1.0
Programmed by Dongha Shin (dshin@smu.ac.kr)
Type h to print help message.
0> h
Enter           Execute 1 instruction.
s(tep) n        Execute n instruction.
g(o)            Execute until HALT.
r(egs)          Print the contents of all registers.
i(Mem) b n      Print n iMem[] locations starting at b.
d(Mem) b n      Print n dMem[] locations starting at b.
t(oggle)        Toggle printing instruction trace.
c(lear)         Reset the machine for new execution.
h(elp)          Print the help message.
q(uit)          Terminate the machine.
0>
```



TM 시뮬레이터 - 디버딩 모드 명령

```
0> r
0000: 0000000000, 0001: 0000000000, 0002: 0000000000
...
0029: 0000000000, 0030: 0000000000, 0031: 0000000000.
(sp=0028, fp=0029, gp=0030, pc=0031)
0> i 0 5
0: in 0
1: jle 0, 6(pc)
2: ldc 1, 1
3: ldc 2, 1
4: mul 1, 1, 0
0> d 0 6
0000: 0000001023, 0001: 0000000000, 0002: 0000000000
0003: 0000000000, 0004: 0000000000, 0005: 0000000000..
0>
```

레지스터 reg[] 보기

명령어 메모리 iMem[] 보기

데이타 메모리 dMem[] 보기

TM 시뮬레이터 - 디버딩 모드 명령

```
0> Enter
0: in 0
<= 10
1> r
0000: 0000000010, 0001: 0000000000, 0002: 0000000000,
...
0029: 0000000000, 0030: 0000000000, 0031: 0000000001:
(sp=0028, fp=0029, gp=0030, pc=0031)
1> g
1: jle 0, 6(pc)
2: ldc 1, 1
...
7: out 1
=> 3628800
8: halt
8>
```

명령어 1개 수행

레지스터 reg[] 보기

명령어 연속 수행

Runtime Environments
(C-Minus 중심)

무엇을 공부하나?

- 본 장에서는 컴파일러가 생성한 프로그램이 어떻게 메모리를 사용하면서 수행하는 지에 대하여 공부한다.
- 컴파일된 프로그램은 수행할 때 메모리를 코드(code) 메모리와 데이타(data) 메모리로 나누어 사용한다.
- 코드 메모리는 프로그램의 명령어가 저장되는 곳으로 프로그램 수행 시 그 내용이 변화되지 않는다.
- 데이타 메모리는 프로그램의 데이타가 저장되는 곳으로 전역(global) 변수들이 저장되는 정적(static) 영역과 인수(argument)나 지역(local) 변수가 저장되는 스택(stack) 영역 및 동적(dynamic) 데이타가 저장되는 동적 영역으로 구성된다.
- 특히 컴파일러는 스택 영역의 사용을 위하여 함수 호출(call) 및 복귀(return) 부분에 스택 메모리 관리를 위한 코드를 생성한다.
- 본 장에서는 C-Minus 프로그램이 수행할 때 어떻게 메모리를 사용하는 지에 대하여 중점적으로 배운다.

강의 내용

- 수행 시간(Runtime) 메모리 구조
- 정적(Static) 영역과 스택(Stack) 영역
- 전역(Global), 인수(Argument) 및 지역(Local) 변수의 주소
- 함수 부름(Call)에 필요한 코드
- C 프로그램 시작(C Startup) 코드

교재

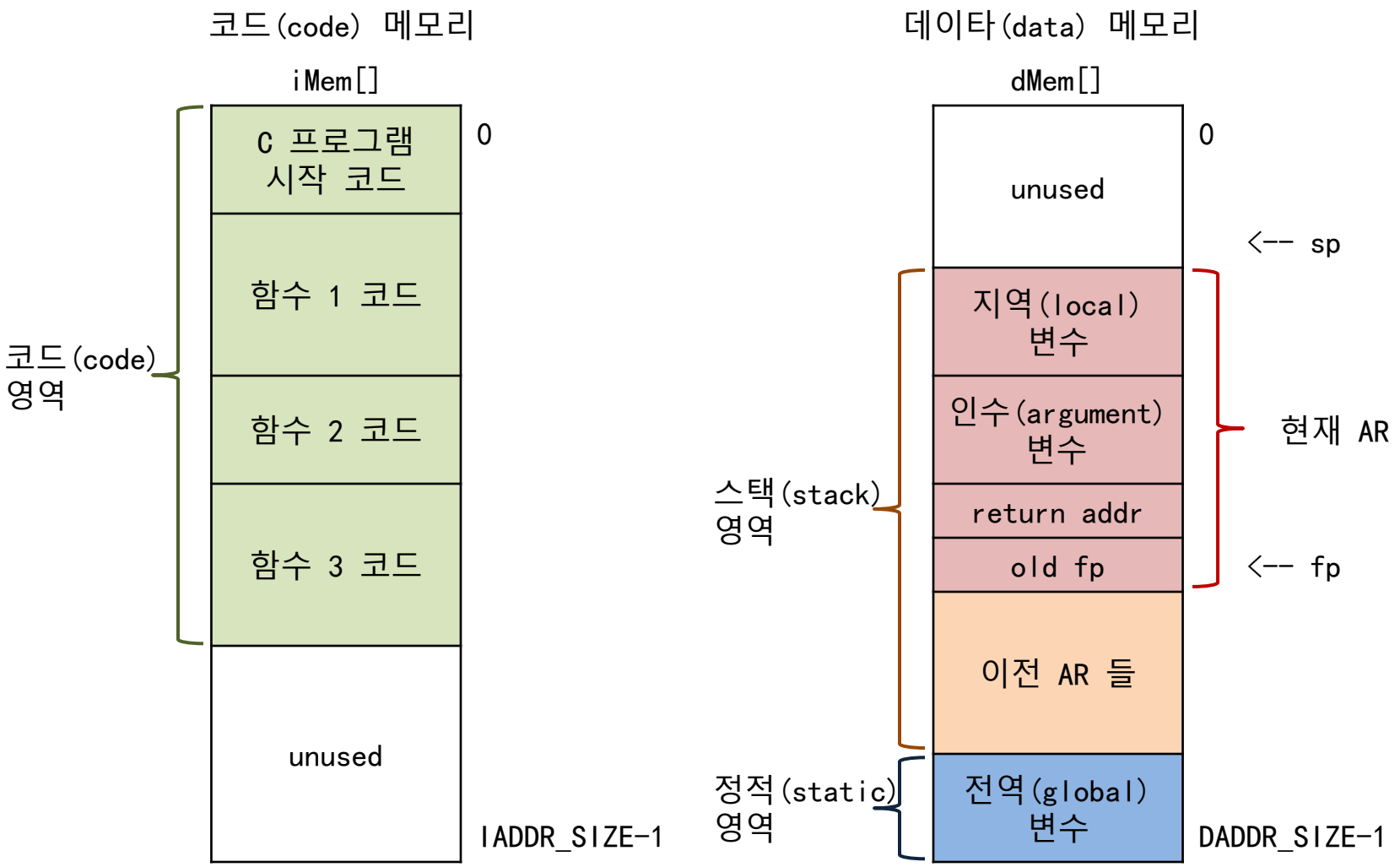
- Kenneth C. Louden, *Compiler Construction Principles and Practice*, PWS Publishing Company, 1997. (7.3, 7.5, A.4)

수행 시간(Runtime) 메모리 구조

메모리	영역	저장 내용	컴파일러의 주소 예측	수행 시간 특징
코드(code) 메모리	코드(code) 영역	명령어 (instruction)	절대 주소 예측	크기 고정 내용 불변
데이타(data) 메모리	정적(static) 영역	전역(global) 변수	절대 주소 예측	크기 고정 내용 변화
	스택(stack) 영역	인수 (argument), 지역(local) 변수 등	fp로 부터 상대 주소 예측	크기 변화 내용 변화
	동적(dynamic) 영역	동적 할당 (dynamic alloc) 변수	예측 불가	크기 변화 내용 변화

주: C-Minus에서 동적 영역은 사용되지 않음.

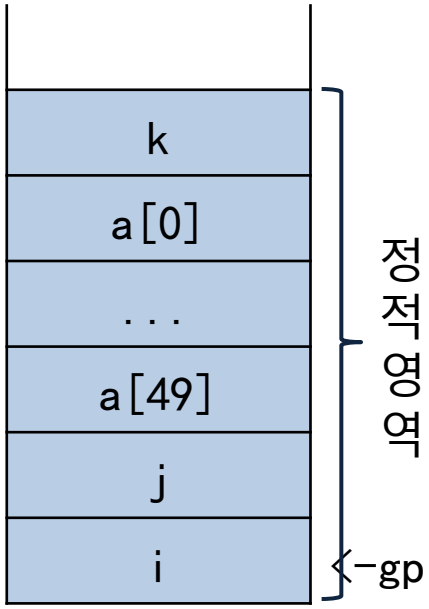
수행 시간 메모리 구조 - 그림



정적(Static) 영역과 스택(Stack) 영역

- 정적 영역
  - 전역 변수가 저장되는 장소이다.
  - 컴파일러는 전역 변수 심볼의 offset 값을 사용하여 수행 시간에 각 변수가 저장되는 주소를 컴파일 시간에 계산할 수 있다.
- 스택 영역
  - 함수 호출 시 호출된 함수의 인수 및 지역 변수가 저장되는 장소이다.
  - 함수 부름(call)을 처리하기 위하여 old fp 및 return addr도 저장한다.
  - 컴파일러는 인수 혹은 지역 변수 심볼의 offset 값과 실행 시간에 값이 변하는 레지스터 fp 값을 사용하여 수행 시간에 각 변수가 저장되는 주소를 컴파일 시간에 계산할 수 있음.

전역(Global) 변수 주소

프로그램 모습	메모리 모습	변수 주소
int i; int j; int a[50]; int k; 심볼의 offset i 0 j 1 a 51 k 52		i gp-0 j gp-1 a[n] gp-51+n k gp-52

인수(Argument) 및 지역(Local) 변수 주소

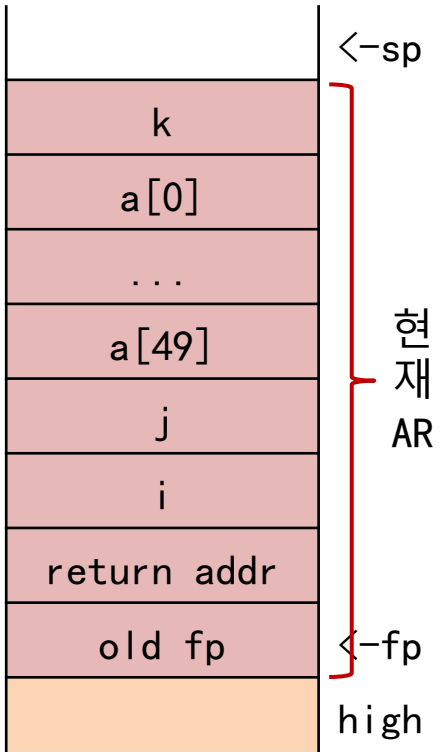
<table>
<tr><th>프로그램 모습</th><th>메모리 모습</th><th>변수 주소</th></tr>
<tr>
<td>

```
void fun(int i) {
  int j;
  int a[50];
  int k;
}
```

심볼의 offset

i	0
j	1
a	51
k	52

</td>
<td>



</td>
<td>

i	(fp-2)-0
j	(fp-2)-1
a[n]	(fp-2)-51+n
k	(fp-2)-52

</td>
</tr>
</table>

함수 부름(Call)에 필요한 코드

	Caller	Callee

소스 코드	fun(exp1, ..., expn);	int fun(int p1, ..., int pn){
		  int l1; ... int im;
		  ...
		}

TM 코드 설명	① exp1 계산하여 push ② ... ③ expn 계산하여 push ④ old fp push하고 새 fp 설정 ⑤ return addr push ⑥ pc에 함수 fun 주소 저장 ⑦ // return 하는 곳	함수 시작 ① sp를 local 변수 크기 만큼 증가 함수 return ① sp에 fp를 저장 ② fp에 dMem[fp] 값 저장 ③ pc에 dMem[sp-1]값 저장

Caller 코드

TM 코드	설명
lda sp, -2(sp)	old fp 및 return addr는 skip
r = 1st 인수 계산 값 push r	1st 인수의 값을 계산하여 r에 저장한 후 이를 stack에 push
...	
r = nth 인수 계산 값 push r	nth 인수의 값을 계산하여 r에 저장한 후 이를 stack에 push
st fp, -n(fp) lda fp, -n(fp)	old fp를 저장한 후 새 fp를 설정 (n은 현재 AR 크기)
lda r, 2(pc) st r, -1(fp)	return addr(=pc+2)를 r에 저장한 후 r을 dMem[fp-1]에 저장
ldc pc, fun	pc=fun (fun은 부르는 함수 주소)

Callee 코드

- 함수 시작

TM 코드	설명
lda sp, -n(sp)	sp를 n 만큼 감소 (n은 local 변수 공간 크기)

- 함수 return

TM 코드	설명

lda 27, 0(r)
lda sp, 0(fp)
ld fp, 0(fp)
	결과 값 r을 r27에 저장
	sp에 fp를 저장
	fp에 dMem[fp] 값 저장

ld pc, -1(sp)	pc에 dMem[sp-1]값 저장

C 프로그램 시작(C Startup) 코드

TM 코드	설명
ld gp, 0(0) st 0, 0(0)	gp = dMem[0] dMem[0] = 0
lda fp, -n(gp) lda sp, -n(gp)	fp = gp - n sp = gp - n (n은 global 변수 공간 크기)

push fp
lda 0, 2(pc)
push 0
ldc pc, main
	함수 main을 call (함수 main은 argument가 없다고 가정)
	(main은 함수 main의 시작 주소)

halt	halt