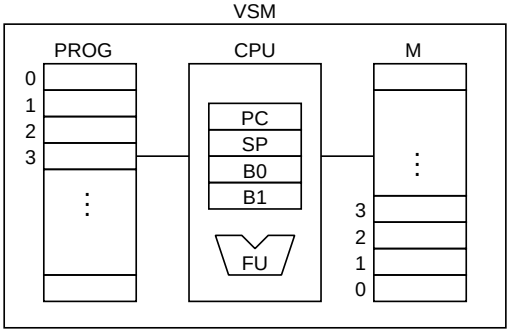
5 仮想スタック機械とその演習

♣ mini-C コンパイラのターゲットの計算機である「仮想スタック機械」 VSM がどんなもので, mini-C や C 言語のプログラムがどのような機械語にコンパイルされ実行されるかの感覚をつかむことを目的とする.
この中で「フレーム」を用いた関数呼出しの仕組みは, 他のプログラミング言語にも共通する非常に重要な概念なので, よく理解して欲しい.

5.1 VSM の構成



1. PROG: プログラムを格納するメモリ.
2. M: データを格納するメモリ. 基本的にはスタックとしてアクセスする.
3. CPU: 演算器 FU (functional unit) といくつかのレジスタを持つ
   (a) PC (program counter) 次に実行する命令を指すカウンタ
   (b) SP (stack pointer) M をアクセスするのに用いる
   (c) B0 (base register 0) global 変数の先頭アドレスを指すのに用いる
   (d) B1 (base register 1) 関数のフレームの先頭アドレスを指すのに用いる

5.2 式の計算

5.2.1 演算

演算は, M スタック上で行う

LC x (load constant 命令)
定数 x をスタックトップにロードする. `SP++; M[SP]=x;`

ADD (add 命令)
スタックトップの 2 数の加算を行う. 2 数をスタックからポップ (除去) し, 結果をスタックトップに置く. `SP--; M[SP]=M[SP]+M[SP+1];`

SUB (subtract 命令)
スタックトップの 2 数の減算を行う. スタックトップの数を 2 番目の数から引き, 2 数をスタックからポップし, 結果をスタックトップに置く. `SP--; M[SP]=M[SP]-M[SP+1];`

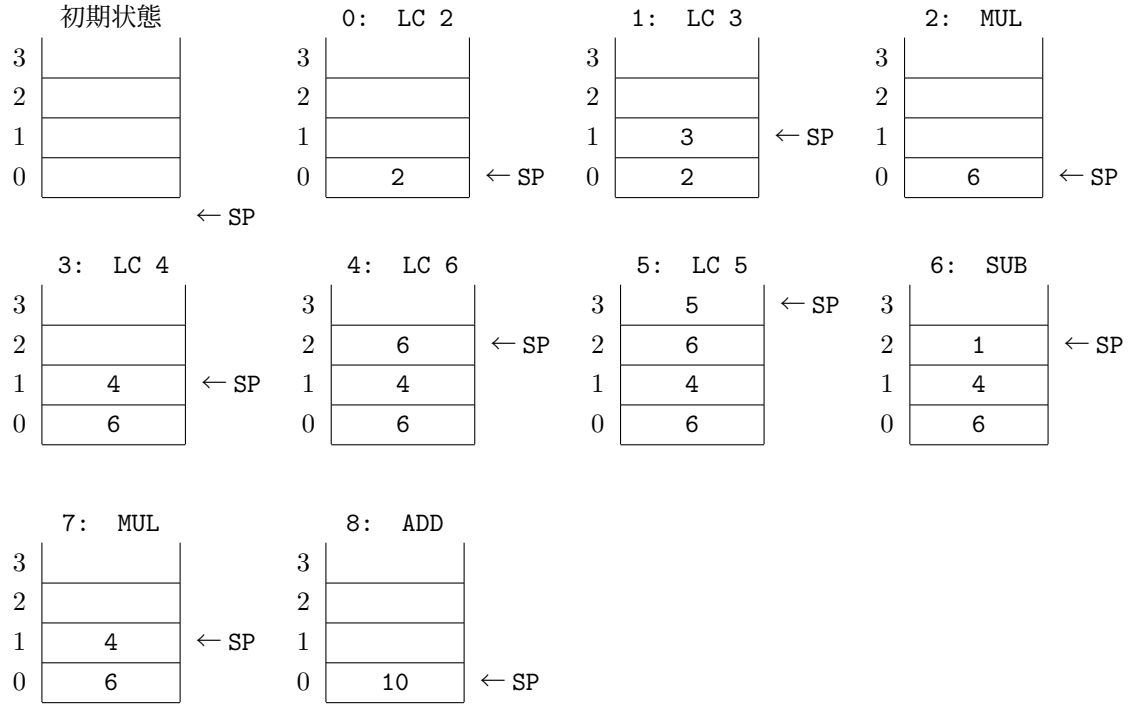
MUL　　(multiply 命令)
スタックトップの 2 数の乗算を行う. 2 数をスタックからポップし, 結果をスタックトップに置く. `SP--; M[SP]=M[SP]*M[SP+1];`

EXIT　　(exit 命令)
実行を終了する. スタックトップの値を return value として OS に返す. `exit(M[SP]);`

(例 5.1) `2*3+4*(6-5)` の計算は次の VSM コードにより実行できる.

```
0: LC 2
1: LC 3
2: MUL
3: LC 4
4: LC 6
5: LC 5
6: SUB
7: MUL
8: ADD
9: EXIT
```

実行の様子 (`M` スタックの内容)



演算命令にはこの他に

`DIV` (整数除算), `MOD` (剰余), `INV` (符号反転)

`EQ` (==), `NE` (!=), `GT` (>), `LT` (<), `GE` (>=), `LE` (<=)

がある. (詳細は **付録 5.3** 参照)

5.2.2　入出力

整数の入出力は `GETI`, `PUTI` 命令で, 一文字の入出力は `GETC`, `PUTC` 命令で行う.

GETI　(get integer 命令)
整数を一つ入力し, スタックトップに置く. SP++; M[SP]=(入力した整数)

GETC　(get character 命令)
一文字を入力し, そのコードをスタックトップに置く. SP++; M[SP]=(入力した文字のコード)

PUTI　(put integer 命令)
スタックトップの整数を出力する. 出力後, その数はスタックからポップされる. M[SP] を整数として出力; SP--;

PUTC　(put character 命令)
スタックトップにあるコードの文字を出力し, ポップする. M[SP] を文字として出力; SP--;

(例 5.2) 3 つの整数 (それぞれ a, b, c とする) を入力し, a+2+b*(3+c) を計算し, その結果を x=... と出力する.

0: GETI	
1: LC 2	
2: ADD	
3: GETI	
4: LC 3	
5: GETI	
6: ADD	
7: MUL	
8: ADD	
9: LC 120	'x' のコード
10: PUTC	
11: LC 61	'=' のコード
12: PUTC	
13: PUTI	
14: LC 10	'\n' のコード
15: PUTC	
16: EXIT	

課題 V.1 (1+2)*(3-18/(4+9)) を計算し, その結果を表示する VSM コードを作成せよ. VSM のシミュレータでこれを実行し, 正しい結果 (6) が得られるかどうか確認せよ. VSM の実行系のダウンロードとコンパイルの方法については**付録 5.1** を参照のこと.

5.3 変数の扱い

1. global 変数の配置と参照

※ C 言語の global 変数と local 変数の違いが分からない人は, **付録 5.2** を参照のこと.

- global 変数はスタックの「底」に配置される.
  - 3 つの global 変数 a, b, c があるとき, スタック M のデータ配置は次のようになる.

初期状態

5		
4		
3		
2	c の領域	← SP
1	b の領域	
0	a の領域	

– 変数領域の確保は ISP 命令により行う

ISP x　　(increment stack pointer 命令)
SP の値を x だけ増やす. SP+=x;

例えば global 変数が 3 つあるときには, プログラムの冒頭に「ISP 3」と書けばよい.

- global 変数の参照は, LV 命令により, ベースレジスタ B0 を用いて行う.
  - 変数参照の際の番地指定は, 変数を格納する領域の先頭番地からの相対番地により行われる.
  - ベースレジスタは, 変数を格納する領域の先頭番地を記憶するレジスタである. ベースレジスタに格納された先頭番地に, 変数の相対番地を足すことにより, 変数の絶対番地 (実効番地) が決定される.
  - VSM は B0, B1 という 2 つのベースレジスタを持つ. B0 は global 変数用, B1 は local 変数用である. global 変数はスタックの「底」に配置されて変動しないので, B0 の値は常に 0 である. (B1 については後述する.)

LV b a　　(load variable 命令)
ベースレジスタ Bb ($b = 0$ または 1) を用いて a 番地の値をスタックトップにロードする. SP++; M[SP]=M[Bb+a];

(例 5.3) (a+b)*c の計算は次のように書ける.

0: ISP 3	3 変数の領域確保 (&a=0, &b=1, &c=2)
1: LV 0 0	a のロード
2: LV 0 1	b のロード
3: ADD	
4: LV 0 2	c のロード
5: MUL	
6: EXIT	

2. global 変数への代入

代入文「x = 式;」は, 次の (a), (b) いずれかのコードに変換する.

(a) SV (store variable) を使う

```
(式のコード)
SV 0 (x の番地)
```

(b) LA (load address) と SI (store indirect) の組合せを用いる

```
LA 0 (x の番地)
(式のコード)
SI
```

命令の説明

SV *b* a　　(store variable 命令)
ベースレジスタ B*b* を用いて, スタックトップの変数を相対番地 a に書き込む (「ストアする」という). ストア後, スタックトップの値はポップされる. `M[Bb+a]=M[SP]; SP--;`

LA *b* a　　(load address 命令)
ベースレジスタ B*b* を用いて相対番地 a の絶対番地を求め, スタックトップにロードする. `SP++; M[SP]=Bb+a;`

SI　　(store indirect 命令)
スタックトップ M[SP] の値を, M[SP-1] の番地にストアする. ストア後, データとアドレスはともにポップされる. `M[M[SP-1]]=M[SP]; SP-=2;`

【注意】(a) と (b) を比べると, 当然 (a) の方が効率は良いが, 1 パスコンパイラで構文解析しながらコード生成を行う際には (b) の方が処理を行いやすい. 作成する mini-C のコンパイラは 1 パスなので, **以下では (b) を標準とすることにする**.

(例 5.4) 代入文「c=a+b;」の VSM コードは次の通り.

0: ISP 3	変数領域の確保 (&a=0, &b=1, &c=2)
1: LA 0 2	c のアドレスのロード
2: LV 0 0	a+b の計算
3: LV 0 1	〃
4: ADD	〃
5: SI	c への代入
6: EXIT	

(例 5.5) a と b を入力し, その平均 c を出力するプログラム

```
1:  int a; int b; int c;
2:  a=getint();
3:  b=getint();
4:  c=(a+b)/2;
5:  putint(c);
6:  putchar('\n');
```

は次の VSM コードに変換できる

0: ISP 3	変数領域の確保 (&a=0, &b=1, &c=2)
1: LA 0 0	a=getint();
2: GETI	〃
3: SI	〃
4: LA 0 1	b=getint();
5: GETI	〃
6: SI	〃
7: LA 0 2	c=(a+b)/2;
8: LV 0 0	〃
9: LV 0 1	〃
10: ADD	〃
11: LC 2	〃
12: DIV	〃
13: SI	〃
14: LV 0 2	putint(c);
15: PUTI	〃
16: LC 10	putchar('\n');
17: PUTC	〃
18: EXIT	

3. local 変数への参照/代入

   global 変数と同様. 違いは, global 変数がベースレジスタ B0 を用いていたのに対し, B1 を用いるということだけ.

課題 V.2 次の計算をする VSM コードを作成し, 実行して正しい結果 (25 と表示) が得られることを確認せよ.

```
1:  int a; int b; int c;
2:  a=3;
3:  b=4;
4:  c=a*a+b*b;
5:  putint(c);
6:  putchar('\n');
```

- 変数 a, b, c は global 変数とする.
- '\n' の文字コードは 10 である.

5.4 分岐

if 文や while 文など実行に分岐が生じる場合は, B (branch) 命令や BZ (branch if zero) 命令を用いる.

B a (branch 命令)
無条件にジャンプする, すなわち PC の値を変更する. a は相対番地であり, 分岐が無かった場合の番地 (すなわち, この B 命令の次の番地) に a をプラスした番地に分岐する. PC+=a;

BZ a (jump if zero 命令)
スタックトップが 0 ならジャンプする (ジャンプ先の番地は B 命令と同じ). スタックトップが 0 でなければ, 次の命令を実行する. if (M[SP]==0) {PC+=a;} SP--;

【注意】例えば, 9 番地の B 命令で 13 番地に分岐したい場合には 「B 4」 ではなく 「B 3」 と指定する. 「分岐が無かった場合の番地」との差が 3 だからである. ループ等で後方に分岐したい場合は負の数を指定する. 9 番地の B 命令で 3 番地に分岐したい場合には, 「B -7」 と指定する.

一般に, プロセッサの命令の動作仕様の記述でプログラムカウンタの値を言うときは, 「その命令の番地」ではなく, 「その命令の番地プラス 1 ワード」あるいは「その命令の番地プラス 1 命令」を意味することが多い. 1 命令が複数のクロックサイクルで実行されるプロセッサでは, 命令のフェッチ (命令メモリから次の命令をロードすること) は先頭のクロックサイクルで行われるが, 通常これと同時にプログラムカウンタの値が増やされる. (特に命令パイプライン制御方式の場合には, 後のクロックサイクルで増やすことができない). したがって, 命令の計算や分岐の処理が行われる時点では, プログラムカウンタは更新されており, 次のワードや次の命令を指しているのである.

1. if 文

else のない if 文

```
if (式) 文
```

は次の VSM コードに変換できる.

```
        「式」のコード
        BZ (L1)
        「文」のコード
(L1)    (次の命令)
```

else のある if 文

```
if (式) 文1 else 文2
```

は次の VSM コードに変換できる.

```
        「式」のコード
        BZ (L1)
        「文1」のコード
        B (L2)
(L1)    「文2」のコード
(L2)    (次の命令)
```

(例 5.6)

```
1:  int exam;
2:  exam=getint();
3:  exam=exam+20; /* 下駄 */
4:  if (100<exam) exam = 100;
5:  putint(exam);
6:  putchar('\n');
```

に対する VSM コードは次の通り.

```
 0: ISP 1    (exam=>0 番地)
 1: LA 0 0   exam=getint();
 2: GETI     〃
 3: SI       〃
 4: LA 0 0   exam=exam+20;
 5: LV 0 0   〃
 6: LC 20    〃
 7: ADD      〃
 8: SI       〃
 9: LC 100   (100<exam)
10: LV 0 0   〃
11: LT       〃
12: BZ 3     条件不成立なら 16 へ分岐
13: LA 0 0   exam = 100;
14: LC 100   〃
15: SI       〃
16: LV 0 0   putint(exam);
17: PUTI     〃
18: LC 10    putchar('\n');
19: PUTC     〃
20: EXIT
```

課題 V.3 次のような四捨五入を行うプログラムの VSM コードを作成せよ.

```
1:  int i; int m;
2:  i = getint();
3:  m = i%10;
4:  if (m<5) i = i-m;
5:  else i = i-m+10;
6:  putint(i);
7:  putchar('\n');
```

– 例えば 123 → 120, 95 → 100 のように, 四捨五入が行えることを確認せよ.

2. while 文

while 文の一般形

```
while (式) 文
```

は次のような VSM コードに変換できる.

```
(L1)   「式」のコード
       BZ (L2)
       「文」のコード
       B [L1]
(L2)   (次の文)
```

(例 5.7) 1 から 10 までの和を求めるプログラム

```
1:  int s; int i;
2:  s=0; i=1;
3:  while(i<=10) {
4:     s=s+i;
5:     i=i+1;
6:  }
7:  putint(s);
8:  putchar('\n');
```

の VSM コードは次のようになる

0: ISP 2	変数領域の確保 (s=>0, i=>1)
1: LA 0 0	s=0;
2: LC 0	〃
3: SI	〃
4: LA 0 1	i=1;
5: LC 1	〃
6: SI	〃
7: LV 0 1	i<=10
8: LC 10	〃
9: LE	〃
10: BZ 11	条件不成立なら 22 番地へ分岐
11: LA 0 0	s=s+i;
12: LV 0 0	〃
13: LV 0 1	〃
14: ADD	〃
15: SI	〃
16: LA 0 1	i=i+1;
17: LV 0 1	〃
18: LC 1	〃
19: ADD	〃
20: SI	〃
21: B -15	7 番地へ分岐
22: LV 0 0	putint(s);
23: PUTI	〃
24: LC 10	putchar('\n');
25: PUTC	〃
26: EXIT	

課題 V.4 階乗を計算をする次のプログラムに対する VSM コードを作成せよ.

(動作確認をして, 結果をレポートに記すこと.)

```
 1:  int n; int f; int i;
 2:  putchar('?'); /* '?'  のコードは 63 */
 3:  n=getint();
 4:  f=1;
 5:  i=1;
 6:  while(i<=n){
 7:     f=f*i;
 8:     i=i+1;
 9:  }
10:  putint(f);
11:  putchar('\n');
```

5.5 関数

5.5.1 フレーム

☆ 関数の「フレーム」は, C 言語等のプログラミング言語の処理系で用いられる, 一般的で重要な概念なので, 少し難しいが, ここはしっかり理解して欲しい.

1. フレーム (frame) とは

   各々の関数の計算に必要なデータを格納する領域. 関数の起動レコード (activation record) とも呼ばれる.

2. フレームの生成と消去

   関数が呼び出される毎にその関数のフレームが 1 つ生成される (フレームの領域が確保される).

   関数の実行が終了すると, フレームは消去される (フレームの領域が解放される).

   フレーム領域の確保/解放はスタック状に行われる.

   例) 次のようなプログラムで

```
int a() {
  a1(); a2();
  return 0;
}

int b() {
  b1(); b2();
  return 0;
}

int main {
  a(); b();
  return 0;
}
```

   フレームの生成/消去は次のように行われる.

【注意】上の例のように, 関数の再帰呼び出しが行われていなければ, 同じ関数のフレームは最大で 1 つしか同時には存在しない. その場合には, フレームは実行時に動的に生成/消去する必要は無く, はじめから各関数に 1 つのフレームを静的に割り当てておけばよいことになる. 再帰呼び出しが許されていない言語 (昔の FORTRAN 等) では, このようにフレームは実行前に各関数に対して 1 つだけ割り当てられていた. しかし, C のように再帰呼び出しが行われる言語では, ある関数の終了前に同じ関数が複数回呼び出されることがあるため, フレームをスタック状に動的に割り当てる必要がある.

3. 関数のフレームに保持されるデータ
   (a) この関数で使用する local 変数
   (b) 関数呼出に用いられた引数
   (c) 戻り番地 [RA]

       この関数終了後に実行すべき命令の番地. つまり, この関数を呼び出した命令の次の命令の番地. (これが無いと, 呼び出し元に戻れない.)

   (d) この関数を呼び出した関数のフレームへのポインタ [RF]

       この関数の実行終了後, 呼び出し元の関数のフレームを回復するために用いる.

   (e) 関数の返り値 [RV]

       この関数の計算結果を呼び出し元に渡すのに用いる.

   (f) その他

5.5.2 VSM の関数呼び出しと関数定義のコード生成

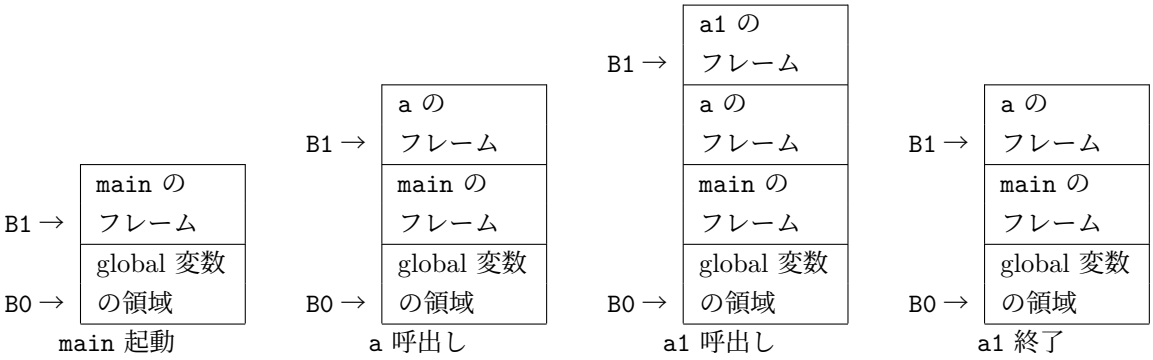
1. VSM のフレームの構成

VSM の記憶はスタック形式なので, global 変数領域の上に, 呼び出される関数のフレームを次々に積み上げていけばよい.

このとき, ベースレジスタの値は次のように設定される.

- B0 には global 変数領域の先頭アドレスが設定される (実際には常に 0).
- B1 には, 現在実行されている関数のフレームの先頭アドレスが設定される.

例) main → a → a1 を呼び出した時の M スタックの様子



2. フレームの内部構成

1 つの関数に対するフレームの内容は次の通り (左の番号は, フレームの起点からの相対番地).

	作業領域
	local 変数
3	引数
2	[RA] 戻り番地
1	[RF] 戻りフレーム番地
0	[RV] この関数の返り値

例) 後述の「成績判定」プログラムの場合

- global 変数は 1 つ (border).
- main の local 変数は 3 つ (report, exam, ok).
- 関数 hantei を呼び出すための引数は 2 つ (x, y).
- 関数 hantei の local 変数は 2 つ (total, ok).

関数 hantei が呼び出された時の M スタックの構成は次のようになる.

15	hantei の作業領域	hantei のフレーム
14	hantei の ok	〃
13	hantei の total	〃
12	hantei の 引数 y	〃
11	hantei の 引数 x	〃
10	hantei の RA	〃
9	hantei の RF	〃
8	hantei の RV	〃 ← B1
7	main の作業領域	main のフレーム
6	main の ok	〃
5	main の exam	〃
4	main の report	〃
3	main の RA	〃
2	main の RF	〃
1	main の RV	〃
0	global 変数 border	← B0

3. 関数の処理に用いる命令

CALL a　　(subroutine call 命令)

呼び出す関数のために RF, RA をセットし, a 番地に分岐する. 番地は絶対番地である.

```
M[SP+2]=B1; M[SP+3]=PC; B1=SP+1; PC=a;
```

RET　　(return 命令)

フレームに記憶された情報を用いて, 関数が呼ばれる前の状態を回復し, 戻り番地に分岐する.

```
SP=B1; B1=M[SP+1]; PC=M[SP+2];
```

SB *b*　　(set base 命令)

B*b* レジスタにスタックトップの値をセットする

```
Bb = M[SP]; SP--;
```

4. 関数呼び出し処理

n 個の引数を持つ関数 abc の呼び出し

```
abc(式 1, 式 2, ..., 式 n);
```

に対する VSM コードは

```
ISP 3
式 1 のコード
式 2 のコード
…
式 n のコード
ISP -(3+n)
CALL (abc の開始アドレス)
```

– 最初の「ISP 3 」で, RV, RF, FA 領域を確保.
– 「式 1 のコード」〜「式 n のコード」の部分で, 引数の値を計算し, スタックに積んでいる.
– 次の「ISP -(3+n)」 命令では, 一旦スタックポインタをもとの位置に戻している. これは, 次の CALL 命令が, 呼び出し時点でのスタックポインタの値を元に RF などを設定するためである.

5. 関数の本体の定義

```
int abc(引数 1, 引数 2, …, 引数 n)
{
  local 変数宣言
  文
}
```

に対するコードは

```
ISP (3 + (引数の個数) + (local 変数領域の大きさ))
(文のコード)
RET
```

– 最初の ISP 命令で, local 変数領域を確保し, abc の作業領域にスタックポインタを移動する.

6. return 文

```
return 式;
```

に対するコードは

```
LA 1 0
(式のコード)
SI
RET
```

– 式の計算結果をフレームの RV にセットし, RET 命令を起動すればよい. RV はフレームの第 0 番地にある. フレームの先頭はベースレジスタ B1 が指しているので, RV のアドレスは「LA 1 0」によりロードできる.

7. main の起動

main も一つの関数として, プログラムの先頭で起動する.

```
0:  ISP (global 変数領域のサイズ)
1:  LC (global 変数領域のサイズ)
2:  SB 1
3:  CALL (main の先頭番地)
4:  EXIT
```

– 最初の ISP 命令で global 変数の領域を確保している.
– 次の LC と SB で, B1 レジスタに main のフレームの先頭番地を設定している.

(例 5.8) 次のような「成績判定」プログラムを考える.

- レポート点 report と試験の成績 exam を入力して, 合格なら OK, 不合格なら NG と出力する.
- 判定は report+exam がボーダライン border 以上なら合格とする. border=60 であるとする.
- 2 数を与えると, その合計が border 以上なら 1, そうでなければ 0 を返す関数 hantei(int, int) を main から呼び出して計算を行うとする.
- border は global 変数とし, main で定義し, hantei で参照できるものとする.

「成績判定」のプログラム

```
 1:  int border;
 2:
 3:  int main()
 4:  {
 5:     int report;
 6:     int exam;
 7:     int ok;
 8:     border = 60;
 9:     report = getint();
10:     exam = getint();
11:     ok = hantei(report,exam);
12:     if (ok) {
13:        putchar('O');
14:        putchar('K');
15:     }
16:     else {
17:        putchar('N');
18:        putchar('G');
19:     }
20:     putchar('\n');
21:     return 0;
22:  }
23:
24:  int hantei(int x, int y)
25:  {
26:     int total;
27:     int ok;
28:     total = x + y;
29:     ok = (total>=border);
30:     return ok;
31:  }
```

「成績判定」のプログラムのコンパイル結果

0: ISP 1	global 変数領域確保 (border=>0)
1: LC 1	B1 を main のフレームに設定
2: SB 1	〃
3: CALL 5	main の呼出し
4: EXIT	終了
5: ISP 6	main の領域の確保
6: LA 0 0	border = 60;
7: LC 60	〃
8: SI	〃
9: LA 1 3	report = getint();
10: GETI	〃
11: SI	〃
12: LA 1 4	exam = getint();
13: GETI	〃
14: SI	〃
15: LA 1 5	ok のアドレスロード
16: ISP 3	hantei(report,exam) の呼び出し
17: LV 1 3	〃
18: LV 1 4	〃
19: ISP -5	〃
20: CALL 39	〃
21: SI	返り値の ok への代入
22: LV 1 5	if (ok)
23: BZ 5	
24: LC 79	OK の出力
25: PUTC	〃
26: LC 75	〃
27: PUTC	〃
28: B 4	
29: LC 78	NG の出力
30: PUTC	〃
31: LC 71	〃
32: PUTC	〃
33: LC 10	改行の出力
34: PUTC	〃
35: LA 1 0	return 0;
36: LC 0	〃
37: SI	〃
38: RET	〃

(次ページへ続く)

39: ISP 7	hantei の領域確保
40: LA 1 5	total=x+y;
41: LV 1 3	〃
42: LV 1 4	〃
43: ADD	〃
44: SI	〃
45: LA 1 6	ok=(total>=border);
46: LV 1 5	〃
47: LV 0 0	〃
48: GE	〃
49: SI	〃
50: LA 1 0	return ok;
51: LV 1 6	〃
52: SI	〃
54: RET	

課題 V.5 次の計算をする VSM コードを作成せよ (動作確認すること).

```
 1:  int K;
 2:
 3:  int main()
 4:  {
 5:      int q;, int a;
 6:      K = 7;
 7:      q = 27; a = round(q); putint(a); putchar('\n');
 8:      q = q*2; a = round(q); putint(a); putchar('\n');
 9:      return 0;
10:  }
11:
12:  int round(int x)
13:  {
14:      int m; int y;
15:      m = x%10;
16:      if (m<K) y = x-m;
17:      else y = x-m+10;
18:      return y;
19:  }
```

【注意】

- 関数 round(x) は四捨五入の拡張で, K-1 捨 K 入を計算する.
- CALL 命令の番地指定は絶対番地 (プログラム全体の先頭から数えての番地) であり, かつ, VSM プログラムの番地は **0 から始まっている**ことに注意せよ.

Nagisa ISHIURA

付録 5.1 VSM 処理系のインストール

1. プログラムとデータのダウンロード
   (a) 講義のホームページ (http://ist.ksc.kwansei.ac.jp/~ishiura/cpl/) の「プログラム」を選択する (ID とパスワード入力が必要).
   (b) 次の 4 つのファイルをダウンロードする
      i. `vsm.c` … VSM シミュレータの本体 (メインプログラム)
      ii. `code.h` … VSM コードの入出力サブルーチンのヘッダ
      iii. `code.c` … VSM コードの入出力サブルーチンの本体
      iv. `test.vsm` … VSM のサンプルコード
2. コンパイル

   次によりシミュレータ `vsm` ができる

   ```
   gcc -o vsm vsm.c code.c
   ```

3. `vsm` の実行
   (a) VSM のコードが入ったプログラムを引数としてコマンドラインから `vsm` を起動する.
   例えば, `test.vsm` の実行には次を入力する.

   ```
   ./vsm test.vsm
   ```

   正しく `vsm` がコンパイルされていれば,

   ```
   8!=40320
   VSM OK
   ```

   が表示される (はず).
   (b) トレースオプション (`-t`) で 1 を指定すると, 命令のトレース (実行した命令列) が表示される. 試してみよ.

   ```
   ./vsm -t 1 test.vsm
   ```

   (c) トレースオプション (`-t`) で 2 を指定すると, 1 命令づつ実行できる.

   ```
   ./vsm -t 2 test.vsm
   ```

   Enter を押す毎に 1 命令を実行し, その時点でのスタックの状態を表示する.
   実行の終了/中止は Ctrl+C.

☆ これらのトレースオプションはデバッグ時に用いるとよい.

付録 5.2 global 変数と local 変数

global 変数 … すべての関数から読み書きできる変数

local 変数 … その変数が宣言されている関数のみから読み書きできる変数

```
 1:  int dollar;   /* global 変数 */
 2:
 3:  int japan()
 4:  {
 5:     int yen;   /* local 変数 */
 6:     int x;     /* local 変数 */
 7:     yen = 1;
 8:     x = 8*dollar+400*yen;   /* この dollar は global 変数 */
 9:  }
10:
11:  int australia()
12:  {
13:     int dollar    /* local 変数 (同じ名前)*/
14:     int y;        /* local 変数 */
15:     dollar = 80;  /* この dollar は local 変数の方 */
16:     y = 100*dollar;
17:  }
18:
19:  int marc;   /* global 変数 */
20:
21:  int main(void)
22:  {
23:     dollar = 100;
24:     marc = 80;
25:     japan();
26:     australia();
27:     return 0;
28:  }
```

- 1 行目の `dollar`, 19 行目の `marc` は global 変数

  `main`, `japan`, `australia` の中から読み書きできる

- 5–6 行目の `yen`, `x` は local 変数

  `japan` の中だけで読み書きできる

- 13–14 行目の `dollar`, `y` は local 変数

  `australia` の中だけで読み書きできる

  この `dollar` のように, global 変数と同じ名前の local 変数を宣言することもできる. `australia` の中で `dollar` を参照すると, それは global 変数の `dollar` ではなく, local 変数の `dollar` を意味する. (global 変数と同名の local 変数を宣言すると, その関数内ではその global 変数にはアクセスする手段がなくなる.)

付録 5.3 VSM の命令セット

レジスタの初期値は, PC=0, B0=0, B1=0, SP=-1 とする.

命　令	意　味	動　　作
EXIT	exit	exit(M[SP]); 終了
LC c	load constant	SP++; M[SP]=c;
LA b a	load address	SP++; M[SP]=Bb+a;
LV b a	load variable	SP++; M[SP]=M[Bb+a];
LI	load indirect	M[SP]=M[M[SP]];
SI	store indirect	M[M[SP-1]]=M[SP]; SP-=2;
SV b a	store variable	M[Bb+a]=M[SP]; SP--;
DUP	duplicate	SP++; M[SP]=M[SP-1];
ISP c	increment sp	SP+=c;
GETC	get character	SP++; M[SP]= 一文字入力;
GETI	get integer	SP++; M[SP]= 空白で区切られた整数を入力;
PUTC	put character	M[SP] の一文字を出力; SP--;
PUTI	put integer	M[SP] の整数を出力; SP--;
ADD	add	SP--; M[SP]=M[SP]+M[SP+1];
SUB	subtract	SP--; M[SP]=M[SP]-M[SP+1];
MUL	multiply	SP--; M[SP]=M[SP]*M[SP+1];
DIV	divide	SP--; M[SP]=M[SP]/M[SP+1];
MOD	modulo	SP--; M[SP]=M[SP]%M[SP+1];
INV	invert	M[SP]=-M[SP];
EQ	equal	SP--; M[SP]=(M[SP]==M[SP+1]);
NE	not equal	SP--; M[SP]=(M[SP]!=M[SP+1]);
GT	greater than	SP--; M[SP]=(M[SP]>M[SP+1]);
LT	less than	SP--; M[SP]=(M[SP]<M[SP+1]);
GE	greater or equal	SP--; M[SP]=(M[SP]>=M[SP+1]);
LE	less or equal than	SP--; M[SP]=(M[SP]<=M[SP+1]);
B a	branch	PC+=a;
BZ a	branch if zero	if (M[SP]==0) {PC+=a;} SP--;
SB b	set base	B[b] = M[SP]; SP--;
CALL a	call	M[SP+2]=B1; M[SP+3]=PC; B1=SP+1; PC=a;
RET	return	SP=B1; B1=M[SP+1]; PC=M[SP+2];

Nagisa ISHIURA