Procedure Calls and Address Parameters in PeANUt

- ref: [PeANUt Spec, sect 4]
- procedure calls
  - with local variables
  - with return values (non-void functions)
- address parameters
  - ability to modify data
  - indirect memory reference via pointers
- assembly coding style

- other issues:
  - Peanut@Home news!
Procedure Calls: Revision

- recall the procedure call convention determines the order of contents of the stack frame
- example C function declaration:

```
int P(int p1, int p2, ...) {
    int l1, l2, ...;
    ...
}
```

| →1 | SP just before (explicitly) pushing parameters & just after popping them |
|    | return value (if any) |
| →2 | SP just before executing call P and just after executing P’s ret |
|    | parameters |
| →3 | SP just before executing 1st instruction in P and just before executing P’s ret |
|    | return address |
| →4 | SP inside body of P, after (explicitly) allocating space for local variables (references to parameters, return value & local variables are relative to this position) |
|    | local variables (if any) |
Procedure Calls: Key Ideas

- call convention is needed so that caller and procedure can agree (on where to exchange data)
- role of the stack for RV, parameters & RA
- symmetric use of SP
- common errors:
  - using the wrong mode when pushing parameters
  - forgetting to allocate RV slot for non-\texttt{void} functions
  - having SP 1 beyond the last parameter before executing call
  - not popping same number of slots after the \texttt{call} instruction
- different from macros: procedure definition & calls remain in \texttt{.img} file
- for procedures, machine needs SP, !, \texttt{call} and \texttt{ret}
Procedure call – Example with Two Local Variables (1)

; /* WriteInt(x, 4) = printf( "%4d", x ) */
; /* WriteInt(x, -4) = printf("%–4d", x ) */
void WriteInt(

x = -4 ; int x,
n = -3 ; unsigned int n) {
    /* what is at !–2? */

NSp = -1 ; int NSp; /* # of ’ ’s to print */
aX = 0 ; unsigned int aX; /* = |x| */
Nlocs = 2 ;

WriteInt:
    incsp #Nlocs ; /* SP=SP+2 */
load !n ; NSp = n - 1; /* AC=Mem[SP–3] */
sub #1 ; /* AC=AC–1 */
store !NSp ; /* Mem[SP–1]=AC */
    ...
incsp #–Nlocs ; } /*WriteInt()*//* SP=SP–2 */
ret ; /* PC=Mem[SP]; SP=SP–1 */
    ...

Procedure WriteInt() is available in module InOut.ass

(WriteHex() can be used similarly to print x as an unsigned hex number)
Procedure Call – Example with Two Local Variables (2)

call-writeint.ass:

; WriteInt(n, 6); /* printf("%6d", n); */
load n ; /* Push(n) */ /* AC=Mem[n] */
incsp #1 ; /* SP=SP+1 */
store !0 ; /* Mem[SP]=AC */
load #6 ; /* Push(#6) */ /* AC=6 */
incsp #1 ; /* SP=SP+1 */
store !0 ; /* Mem[SP]=AC */
call WriteInt;
; /* SP=SP+1; Mem[SP]=PC; PC=WriteInt */
incsp #-2 ; /* Pop(2) */ /* SP=SP–2 */

PC= call (WriteInt) ret PC=

COMP2300 P8: PeANUt Assembler: Procedures 2009
Non-void Functions in PeANUt

- we implement local variables via the stack
  - good: economy, privacy, recursion and sharable, re-entrant code and multithread-safe
  - bad: we don’t have stack-indexed (or stack-indirect) addressing modes in PeANUt
- inside procedures:
  - we have to increment the SP by number of local variables just after entry (first instruction within procedure)
  - and decrement SP by number of local variables just before (each) `ret`
  - parameters, local variables and return values (RVs) have stack offsets
- non-void function call is similar, but it must first make (empty) slot for the return value, which gets accessed after call before it is popped
Non-void Function – Example with One Return Value (1)

```
RV    = -3 ;  int Log10(
x    = -2 ;      unsigned int x) {
      ;
Logx  = 0 ;  unsigned int Logx;
NLocs = 1 ;

Log10:
      ;
     incsp #NLocs ; /* SP=SP+1 */
     load  !x ;     if (x != 0) { /* AC=Mem[SP-2] */
     cmp   #0 ;                /* compare AC,0 */
     beq   Lendif ;

Lendif: ; } /* if */
     load  !Logx ;     return Logx; /* AC=Mem[SP] */
     store !RV ;    /* Mem[SP-3]=AC */
     incsp #−NLocs ; } /*Log10()*/ /* SP=SP−1 */
     ret ;      /* PC=Mem[SP]; SP=SP−1 */
      ;
```

Procedure Log10() is available in module InOut.ass
Non-void Function – Example with One Return Value (2)

call-log10.ass:

```
incsp #1 ; /* SP=SP+1 */ /* make RV slot */
Push (#511) ; /* AC=511; SP=SP+1; Mem[SP]=AC */
call Log10 ; /* SP=SP+1; Mem[SP]=PC; PC=Log10 */
Pop (1) ; /* SP=SP−1 */
load !0 ; /* AC=Mem[SP] */ /* store RV */
store log ; /* Mem[Log]=AC */
incsp #−1 ; /* SP=SP−1 */ /* pop RV slot */
```

---

COMP2300 P8: PeANUt Assembler: Procedures 2009
Address Parameters in PeANUUt

- in general there are two different types of parameters: value (local copy) and reference (pointers)
- full power of the procedure concept requires the ability to modify data (parameters)
- do via passing address (not value) on stack when calling
- procedures thus reference the *real* memory location indirectly via this pointer
Address Parameters – Example (1)

```c
; void Sum(
a = -3 ; int a,
b = -2 ; int b,
c = -1 ; int *c) {
Sum:
    load !c ; *c = a + b;
    storexr ; /* XR=Mem[SP+c] */
    load !a ; /* AC=Mem[SP+a] */
    add !b ; /* AC=AC+Mem[SP+b] */
    store *0 ; /* Mem[XR]=AC */
    ret ; } /* Sum() */
```

Code is in address-procedure-example.ass
Address Parameters – Example (2)

p: 

q: 

main:

load #59 ; p = 59;
store p ; Sum(5, p, &q);
load #5 /* Push(5); */
incsp #1 ;
store !0 ;
load p ; /* Push(p) */
incsp #1 ;
store !0 ;
loada q ; /* Pusha(q) */
incsp #1 ;
store !0 ;
call Sum ;
incsp #–3 ; /* Pop(3) */
trap #1 ; return 0;
Copy address parameter stack slot to XR, then access memory location via *0
Address Parameters – Arrays

- for array parameters, put address plus index value in XR
- an address parameter is a pointer to a memory location that the procedure may modify (unfortunately (?), we cannot use @ indirect mode)
- example (from InOut.ass):

```assembly
; void WriteString(
    s       = -2 ; char s[]) { /* same as char *s */
    i       = 0 ;
    NumLocs = 1 ;
    WriteString:
        incsp  #NumLocs ;
        ...
        load   !s ;     printf("%c", s[i]);
        add    !i ;
        storexr ;      /* XR = Mem[SP+s]+Mem[SP+i] */
        load    *0 ;
        trap   #3 ;     /* Put */
        ...
```
Coding Style

- Use C code to make things clearer
  - Often line-by-line correspondence with assembly code
  - Short comment for each important variable declaration
  - Properly indented (2 spaces)
  - Syntactically correct
  - Use symbolic constants for both

- Macros: sensibly chosen; well commented when defined

- Algorithm:
  - Simple is beautiful
  - No unnecessary output (like prompts)
  - Own procedures (if any) should not use global variables

- Assembly code:
  - Properly indented
  - Keep identifiers as meaningful as possible (in 6 letters)
  - Beware of optimisations
  - No spaghetti code!
Appendix: Calling from Procedures with a Single Stack Pointer

● a problem! See printshex.ass. Soln 1: see WriteInt() from InOut.ass. Alt.:

```plaintext
NSp = −1 ;       /* Q: how does given code ...
aX = 0 ;         /* ... in InOut.ass avoid ...
NSpm1 = −2 ;     /* this problem? */
aXm1 = −1 ;
...
incsp #1 ;       Nsp = Nsp − Log10( aX );
load !aXm1 ;     /* !aX−1 points to same
incsp #1 ;       mem location as !aX
store !0 ;       did before 1st incsp */
call Log10 ;
incsp #−1 ;
load !NSpm1 ;    /* similarly for NSp */
sub !0 ;         /* RV of Log10 */
store !NSpm1 ;
incsp #−1 ;      /* Q: what if this was omitted? */
```

● alt., we can get the required value into AC before moving SP:

```plaintext
load !aX ;       Nsp = Nsp + −1 * Log10( aX );
incsp #2 ;       /* works because we only
store !0 ;       have to push one value */
call Log10 ;
load !−1 ;       /* RV */
mul #−1 ;
incsp !−2 ;
add !NSp ;
store !NSp ;
```