

Bit Operations and Traps in PeANUt

- ref: [PeANUt Spec, sect 2.8.3 and Appendix B]
- bitwise operations
- traps:
 - concepts
 - in PeANUt;
 - ◆ predefined and user-definable
 - trap handler and trap table
- debugging
- other issues:
 - overall feedback on Assignment 1, Tute/Labs and the MSE (comments, marks)
 - wed11-13 tute/lab session arrangements for Apr 25

Bitwise Operations in PeANUt

- need to get at bit-level of data in many applications
- example: what does the following code do?

```

load    x        ;
not     ; /* AC = not AC */
add     #1       ;
store   y        ;
    
```

- bit masks are useful:

```

Msk:    data    %00000 111111 00000    ; int Msk = 2016;
CMsk:   data    %11111 000000 11111    ; int CMsk = ~Msk; /* -
tmp:    block   1                      ; int tmp;
    
```

(bitop-example.ass)

- to get a range of bits:

```

AC: [xxxxx | yyyyyy | zzzzz]
    and Msk
AC: [00000 | yyyyyy | 00000]
    store tmp
    
```

- to set a range of bits:

```

AC: [xxxxx | aaaaaa | zzzzz]
    and CMsk
AC: [xxxxx | 000000 | zzzzz]
    or tmp
AC: [xxxxx | yyyyyy | zzzzz]
    
```

- note: we can use `dvd #32` to shift right 5 bits and `mul #32` to shift left 5 bits (care: overflow!)

PeANUt Predefined Traps

#1 Halt (return control to operating system, user program process then terminates)

#2 Get (operating system code will *wait* if needed)

#3 Put

#4 Data error (from Get/Put)

#5 Illegal Instruction

#6 Illegal Mode (e.g. from **store** #5)

#7 Integer Overflow (if EN=1 abort, else OV=1)

#8 Integer Divide by 0

#9 Establish Trap Routine (set new trap or modify existing one)

#10 Trapping Error (from e.g. **trap** #23 (if trap 23 not yet established), or error in handling established trap)

#11 Page Fault (for PeANUt virtual memory mode only; needs predefined handler at a46)

#12, #13 Swap Page In, Out (AC contains page number)

● note: the default action of traps #4, #5, #6, #8, #10 is to abort

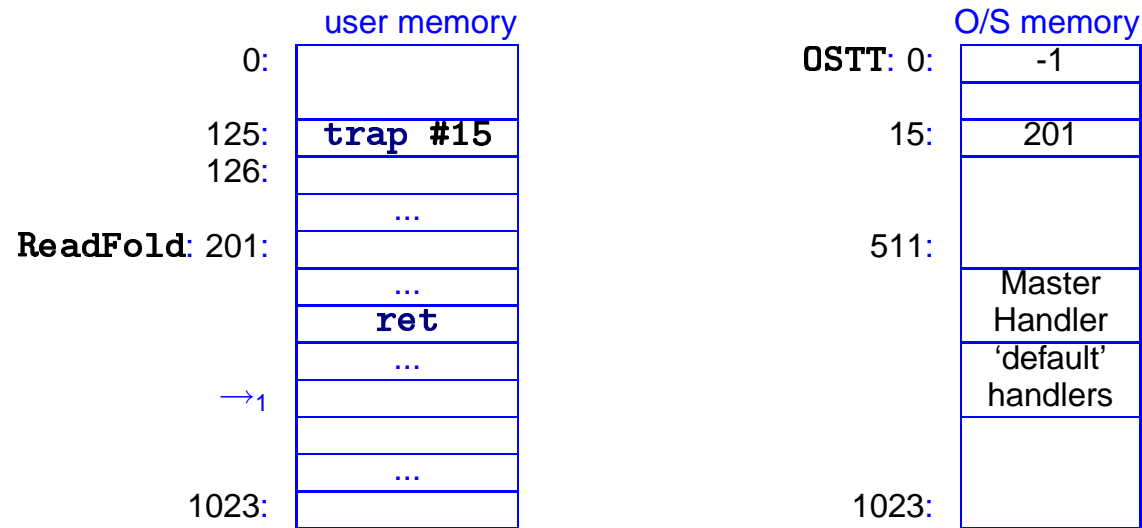
Software Initiated Traps: softwaretrap-example.ass

- can be used to implement simple procedures, e.g. define trap 15 to read in chars. with upper case folded to lower case

```
TT15:    data    15        ;
         data    ReadFold; //ret. next char read, folded to lower
ReadFold:                ; char ReadFold() { /*RV passed via AC*/
                          ; char ch;      /*also impl. via AC*/
         trap    #2      ; scanf("%c", &ch);
         cmp     #'@'    ; if ((ch >= 'A') &&
         ble     RFenf   ;         (ch <= 'Z')) {
         cmp     #'Z'    ;
         bgt     RFenf   ;
         sub     #'A'    ;     ch = ch - 'A'
         add     #'a'    ;         + 'a';
RFenf:   ;
         ;             ;
         ;             ;
         ret      ; } /* ReadFold() */
main:    ; int main() {
         loada   TT15   ;     /* establish trap #15 */
         trap    #9     ;     /* with handler ReadFold */
do1:     ; do {
         trap    #15    ;     char c = ReadFold(); /*impl. via AC*/
         trap    #3     ;     printf("%c", c);
```

Traps and the Operating System

- a simple model for the operating system role of traps:
 - user program sees only *half* of the PeANUt machine;
 - operating system has own memory, special instructions and registers
 - operating system has 512-word trap table (**OSTT**) for current action of each trap and also code (handler routines) for the default actions
 - a trap is like a **procedure call** to an operating system Master Handler, which then uses the **OSTT** to call the corresponding handler routine



Traps – Review

- establish traps via TTI (trap table item) and **trap #9**
 - traps work via a procedure-like interaction of the operating system and user program
 - the handler routine calling convention is different:
 - uses AC to pass parameter and return value (if any)
 - ◆ Q: how does program control return to correct point in user program?
what about upon exceptions?
- require a fair bit of extra hardware
(and then some more to perform raw input/output accesses etc.)
- virtual input/output is an important abstraction, and is usually implemented via traps
(simplicity, security)
- “No [user program] is an island!”

Debugging

- very 'small' errors can lead to very strange results!
- use break points in the PeANUt tool. Set a break point at:
 - each **call** instruction (check parameters)
 - head of each (unproven) loop (check index variables)
 - other critical points
 - are the values what you expected?
 - use **single step** between break points if suspected bug is there
 - map memory / PC locations to `.ass` code
(compare with listing file `.lst` if not obvious)
 - check sequence of instructions, value of accumulator at **load / store**
- end of PeANUt Module; followup with 'real world' perspectives:
 - Memory Systems and Modern Machines (5)
 - Operating System Concepts (4)
 - Interconnection Networks (1)