

## PeANUt Assembly Language: Loops and Arrays

- ref: [PeANUt Spec, sect 4]; additional reading: [O'H&Bryant, sect 3.8]
- loops
- arrays in assembly language
- evaluating complex expressions
- other issues:
  - Mid-Semester Exam: some details
  - revise 2006 exam Q1

### Translating Loops into Assembly Language

- iteration is required for most (non-trivial) computations
  - **while loop**: test is at the top of the loop i.e. `while (condition) { ... }`
  - e.g. while-example.ass
- ```
while1:  load   n           ;   while (n != 0) {
        cmp    #0         ;
        beq   endwh1      ;
        load  pn          ;   pn = pn * 2;
        mul  #2           ;
        store pn          ;
        load  n           ;   n = n - 1;
        sub  #1           ;
        store n          ;
        jmp  while1       ; }
endwh1:  trap   #1         ;   return 0;
```
- body of if is executed 0 or 1 time; body of while can be exec. 0, 1 or many times
    - the difference: an unconditional backward branch (`jmp`) at the bottom
    - e.g. if first instr'n is at address 6, `mem[n] = 1`; what happens to the PSW?

```
load   n           ; GT=0  EQ=0  PC=7
cmp    #0          ; GT=1  EQ=0  PC=8
beq   endwh1      ; GT=1  EQ=0  PC=9
```

## Do-While Loop in Assembly Language

- test is at the bottom of the loop (iterate 1 or many times)
- i.e. `do { ... } while (condition)`
- e.g.

```
repeat1:  EOL = 10           ; #define EOL 10 /* new line */
          ;
          ; do {
          ;   trap   #2           ;   scanf("%c", &n);
          ;   store  n           ;
          ;   load   n           ; } while (n != EOL);
          ;   cmp    #EOL        ;
          ;   bne   repeat1      ;
```

- review:
  - `while` → like if but with `jmp` at end
  - `do { ... } while` → like if in reverse order (but uses the *same* branch instr'n to the condition)
  - for loop translates into while loop
  - for loops, the machine needs only a backwards branch capability

### Arrays in Assembly Language

- iterative computations normally require an iterative data structure
  - the array is the most fundamental
- how is defining / access different from normal variables?
- easiest to use lower bound of 0 (zero), like in C

```
          N           = 4           ; #define N 4;
          ;
a:        block   N           ; char a[N+1];
          block   1           ;
b:        block   N           ; int b[N];
i:        block   1           ; int i;
          ...
          load    i           ; printf("%c", a[i]);
          storexr ;           /* XR = AC */
          load   *a          ;           /* AC = mem[a+XR] */
          trap   #3          ;
          load   i           ; b[i] = 5;
          storexr ;           /* XR = AC */
          load   #5          ;           /* AC = 5 */
          store  *b          ;           /* mem[b+XR] = AC */
```

## Array Example: Memory Layout

- the memory layout of the previous (and next) example is

|    |      |      |
|----|------|------|
| a: | 'a'  | a[0] |
|    | 'b'  | a[1] |
|    | 'c'  | a[2] |
|    | 'd'  | a[3] |
|    | '\0' | a[4] |
| b: |      | b[0] |
|    |      | b[1] |
|    | 5    | b[2] |
|    |      | b[3] |
| i: | 2    |      |

- general method: put the value of the index in XR (via AC), then simply use indexed addressing mode
- strings: position of the first NULL ('`\0`') character defines the length
  - can be more conveniently initialized using the `data` directive
  - e.g. `a: data "abcd\0"`

## Arrays: Further Translation Patterns

- array access inside conditions: just set up XR before evaluating

```

load    i        ;   if (b[i] > 0) {
storexr ;
load    *b       ;                               /*AC = mem[b+XR]*/
cmp     #0       ;
ble     endif1  ;
load    x        ;   x = x + b[i]; /*AC = mem[x]*/
add     *b       ;                               /*AC += mem[b+XR]*/
store  x        ;   }
endif1: ;

```

- one can use XR as an index variable: (c.f. array-example.ass)

```

load    #0       ;   i = 0;           /*AC = 0*/
storexr ;                               /*XR = AC*/
repeat1: ;   do {
load    *a+1     ;   b[i] = a[i+1];
store  *b       ;
incxr  #1       ;   i = i+1;
loadxr ;                               /*AC = XR*/
cmp    #N       ;   } while (i != N);
bne   repeat1  ;

```

## Further Array Example: array-example.ass

```

N      = 4      ;   #define N 4;
a:     block   N      ;   char a[N+1];
       block   1      ;
b:     block   N      ;   int b[N];
i:     block   1      ;   int i;
       ;

...    ; // (code to initialize a[] omitted)

load   #0     ;   i = 0;
store  i     ;
repeat1: ;   do {
load   i     ;   b[i] = a[i+1]; /* AC = mem[i] */
storexr ;                               /* XR = AC */
load   *a+1  ;   /* AC = mem[a+1+XR] */
store  *b    ;   /* mem[b+XR] = AC */
load   i     ;   i = i+1;
add    #1    ;
store  i     ;
load   #N    ;   } while (i < N);
cmp    i     ;
bgt   repeat1 ;

```

## Arrays in Assembly Language - Review

- more difficult if two different indices are used:
  - e.g. `b[j] = a[i] + 42;`
  - solution?
- define an array using `block array_length`
- set XR to value of the index (usually start at 0)
- reduce access of multi-dimensional array elements to an effective 1-dimensional access (e.g. through row-major ordering) (*...later*)
- for arrays, the machine needs an indexed addressing mode to efficiently access individual elements

