PeANUUt Assembly Language: Loops and Arrays

- ref: [PeANUUt Spec, sect 4]; additional reading: [O’H&Bryant, sect 3.8]
- loops
- arrays in assembly language
- evaluating complex expressions

other issues:
- Mid-Semester Exam: comparison with 2008 MSE
- Assignment 2 is up!
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.

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

- 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

\[
\begin{align*}
N &= 4 ; \quad \text{\#define N 4;} \\
\text{a:} &\quad \text{block N ; char a[N+1];} \\
\text{block} &\quad 1 ; \\
\text{b:} &\quad \text{block N ; int b[N];} \\
\text{i:} &\quad \text{block 1 ; int i;} \\
... &\quad \text{load i ; printf("\%c", a[i]);} \\
\text{storexr} &\quad ; \quad /\ast XR = AC \ast/ \\
\text{load} &\quad *a ; \quad /\ast AC = \text{mem}[a+XR] \ast/ \\
\text{trap} &\quad \#3 ; \\
\text{load} &\quad i ; \quad b[i] = 5; \\
\text{storexr} &\quad ; \quad /\ast XR = AC \ast/ \\
\text{load} &\quad \#5 ; \quad /\ast AC = 5 \ast/ \\
\text{store} &\quad *b ; \quad /\ast \text{mem}[b+XR] = AC \ast/
\end{align*}
\]
Array Example: Memory Layout

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

```
<table>
<thead>
<tr>
<th>a:</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>'a'</td>
<td>a[0]</td>
</tr>
<tr>
<td></td>
<td>'b'</td>
<td>a[1]</td>
</tr>
<tr>
<td></td>
<td>'c'</td>
<td>a[2]</td>
</tr>
<tr>
<td></td>
<td>'d'</td>
<td>a[3]</td>
</tr>
<tr>
<td></td>
<td>'\0'</td>
<td>a[4]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>b:</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>b[0]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b[1]</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>b[2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b[3]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>i:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>
```

- 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"
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: Further Translation Patterns

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

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

- one can use XR as an index variable:

```assembly
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 ;
```
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) \( \ldots \) later

- for arrays, the machine needs an indexed addressing mode to efficiently access individual elements
Evaluating Complex Expressions

- compound conditions may be joined by `&&` (AND) or `||` (OR - more tricky!)
- example: skip characters until a digit is found

```assembly
    do1:    ; do {
      trap   #2    ; scanf("%c", &ch);
      store  ch    ;
      load   #'0'   ;   } while (('0' > ch) ||
      cmp    ch     ;
      bgt    do1   ;
    }
      load   ch     ;   (ch > '9'));
    while1: load   #'0'   ;   while (('0' <= ch) &&
      cmp    ch     ;
      bgt    endwh1 ;
      load   ch     ;   (ch <= '9')) {
      cmp    #'9'   ;
      bgt    endwh1 ;
      trap   #2    ; scanf("%c", &ch);
      store  ch    ;
      jmp    while1 ;  }
    endwh1: ...
```
Arithmetic Expressions: Direct Evaluation if ‘Right-simple’

- **right-simple** form: if the right-hand side of every arithmetic operator is a variable or a constant.
- Such expressions can be easily evaluated, e.g.:

  ```assembly
  load x ; x = x * 10
  mul #10 ;
  add ch ; + ch
  sub #’0’ ; - ’0’;
  store x ;
  ```

- Similar evaluation if on left operand in a conditional expression:

  ```assembly
  load x ; if ((x+1) <= 0) {
  add #1 ; ...
  cmp #0 ;
  bgt ... ;
  ```
Using The Stack to Evaluate Complex Expressions

- general strategy: break down expression into right-simple sub-expressions, which are stored in temporary variables
- can use stack to store n temporary variables
- incsp #n, then access !0, !−1, !−2, ..., !−(n−1)
- e.g. \( d = v - (v / 10) \times 10 + x; \)

```assembly
incsp  #1          ;   /* allocate stack space for t1 */
t1    = 0          ;   /* define SP offset for t1 */
load  v             ;   t1 = (v / 10)
dvd   #10           ;
mul   #10           ;   * 10;
store  !t1          ;
load  v             ;   d = v - t1
sub   !t1           ;
add   x             ;   + x;
store  d             ;
incsp  #-1          ;   /* deallocate stack space for t1*/
```

- important: don’t forget incsp #−n at the end!