

PeANUt Assembly Language: Loops and Arrays

- ref: [PeANUt Spec, sect 4]; additional reading: [O'H&Bryant, sect 3.8]
- loops
- arrays in assembly
- evaluating complex expressions
- other issues:
 - weighting of assignments in assessment scheme
 - Mid-Semester Exam: some details
 - revise 2006 exam Q1(b),(c),(d)

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 {
          ;       scanf("%c", &n);
          ;   } while (n != EOL);
          ;
          ;   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 */
```


