

More PeANUt assembly language

- Loops in assembly code
- Arrays in assembly
- Evaluating complex expressions
- Reference:
 - Specification of the PeANUt computer (Section 4)
- Additional reading: Section 3.8 in Bryant and O'Hallaron

Translating loops into assembly code

- Iteration is required for most (non-trivial) computations
- While loop: Test is at the top of the loop
- Example: `while (condition) {...}`

```
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: ...
```

While loop in assembly

- Body of **if** is executed zero or one time, but body of **while** is executed zero, one or many times
- The difference: A unconditional backward branch (**jmp**) at the bottom
- Example: Assume first instruction is at address **a6**
- What happens to the **PSW** ($n = 1$)?

load n	GT=0	EQ=0	PC=a7
cmp #0	GT=1	EQ=0	PC=a10
beq endwh1	GT=1	EQ=0	PC=a11

Do ... while loop in assembly

- Test is at the bottom of the loop
- Example: `do {...} while (condition)`

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

- Review:
 - **while** → Like **if** with **jmp** at end
 - **do ... while** → Like **if** in reverse order
 - **for** loop translates into **while** loop
 - For loops the machine needs a backwards branch capability

Arrays in assembly

- Iterative computations normally require an iterative data structure
→ **Array** is the most fundamental
- How is defining / access different from normal variables?
- Easiest to use lower bound of **0** (zero), like in C

Array example in assembly

```
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 = Memory[a+XR] */
trap #3            ;
load i              ; b[i] = 5;
storexr            ; /* XR = AC */
load #5            ; /* AC = 5 */
store *b           ; /* Memory[b+XR] = AC */
```

Array example memory layout

- The memory layout of the previous 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 accumulator), then simply use indexed addressing mode
- Strings: Position of the first NULL ('\0') character defines the length of a string

Another array example

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

Conditions and arrays

- Just set up **XR** before evaluating

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

XR as index variable

- One can use **XR** as index variable

```
        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 - Review

- More difficult if two different indices are used
Example: $b[j] = a[i] + 42;$
- Define an array using **block** *array_length*
- Set **XR** to value of the index (usually start at 0)
- For arrays, the machine needs an indexing address mode to efficiently access individual elements

Evaluating complex expressions

- Compound conditions may be joined by **AND** or **OR**
- Example: Skip characters until a digit is found

```
while1:                ; do {
    trap #2            ;     scanf("%c", &ch);
    store ch           ;
    load #'0'          ; } while (('0' > ch) || (ch > '9'));
    cmp ch             ;
    bgt while1         ;
    load ch            ;     /* (ch > '9') */
    cmp #'9'          ;
    bgt while1         ;
    ...
```

Evaluating complex expressions (2)

- Example: Skip characters until a non-digit is found

```
                trap #2      ; scanf("%c", &ch);
                store ch     ;
while1:         load #'0'    ; while (('0' <= ch) && (ch <= '9')) {
                cmp ch       ;
                bgt endwhile1 ;
                load ch      ; /* (ch <= '9') */
                cmp #'9'    ;
                bgt endwhile1 ;
                trap #2      ; scanf("%c", &ch);
                store ch     ;
                jmp while1   ; }
endwhile1:     ...
```

Direct evaluation if right-simple

- If the right-hand side of every arithmetic operator is a variable or a constant

```
load #10      ; x = x*10
mul x         ;
add ch        ;      + ch
sub #'0'      ;      - ord('0');
store x       ;
```

- Evaluation is similar for **if**

```
load x        ; if ((x+1) <= 0) {
add #1        ;
cmp #0        ;
bgt ...       ;
```

Use stack for complex expressions

- Use stack to store temporary variables
- `incsp #n`, then access `!0, !-1, !-2, ... !(n-1)`
- Example: $d = v - (v / 10) * 10 + x$;

```
incsp #1      ; /* Allocate stack space for t1 */
t1 = 0       ; /* Define relative place 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!