PeANUt Assembly Language: a Better Way to Initialize the PeANUt

- ref: [PeANUt Spec, sect 4]

- today:
  - motivation
  - addressing modes revisited
  - assembly language format
  - translating C into PeANUt

- over next 4 lectures:
  - a ‘second pass’ of PeANUt (faster!)
  - a somewhat higher-level view, emphasizing translation from C

- other issues:
  - revise lecture P3
  - revise 2006 exam, Q2(a–d)
Motivation for Assembly Language

- assembly language is 'next level' of abstract machine (after conventional machine and operating system)
  (machine levels figure)
- implemented via translation rather than interpretation
- example machine language file:

```assembly
START a20 ; start address
AT a10

D3 ; L
D4 ; M
D2 ; N

AT a20
O1 O1 a10 ; a20: load L into AC
O1 O6 a11 ; a21: multiply by M
O1 O7 a12 ; a22: compare with N
O53 a25 ; a23: branch to a25 if L*M > N
O1 O2 a13 ; a24: store L*M into a13
O65 a1 ; a25: trap 1 (halt)
```
Problems with Machine Language

● what if we have to:
  ■ replace instruction a24 with two new instructions?
  ■ need to insert a large string at the beginning of the program?
  ■ use an addressing mode which is not supported by the machine?

● does this express a good algorithm?

● can we (easily) write much larger programs in this way?
  ■ can we utilize libraries?
Solutions

- symbolic names (‘labels’) for addresses (variables, branch targets, procedure entry points) needed
  - especially need position-independent code!
  - must sacrifice direct control of memory layout in mli
- we also need symbolic names (mnemonics) for opcodes and modes
- symbolic names for user-defined constants also useful
- we need a way of defining more complex operations (macros)
- separate conversion of (mli → img) of program modules will be useful
- we can document assembly language code with high level language code to express a (structured) algorithm
  - we can introduce standard ‘translation patterns’ to illustrate the compilation process
Review: PeANUt Addressing Modes

- corresponding to most instructions is an operand (OP)
- OP is sometimes derived from the corresponding address (AOP)
  
  e.g. OP = Memory[AOP]

- AOP is generally derived from the lowest 10 bits of the instruction (opspec)

- PeANUt has five addressing modes:
  - immediate (#)
  - direct
  - indirect (@)
  - indexed (*)
  - stack (!)
PeANUt Addressing Modes – 1

- **Immediate (#):** \( OP = \text{opspec} \)
  
  \(-512 \leq OP \leq 511\)

  ```
  load #6 ; OP = 6  AOP is undefined
  mul #−2 ; OP = −2  AOP is undefined
  ```

- **Direct:** \( AOP = \text{opspec} \quad OP = \text{Memory}[AOP] \)

  ```
  n=26
  pn=27
  load n ; AOP = 26  OP = 5
  mul pn ; AOP = 27  OP = 1
  add 27 ; AOP = 27  OP = 1
  sub 30 ; AOP = 30  OP = −10240
  add 1 ; AOP = 1  OP = ?
  ```
PeANUt Addressing Modes – 2

- **indirect (⊗):**  
  \[ AOP = \text{Memory}[\text{opspec}] \quad OP = \text{Memory}[AOP] \]

  \[
  \begin{array}{c|c|c}
  \text{load} & \text{a=10} & AOP = 12 \\
  \text{OP} & 42 & \\
  \end{array}
  \]

- **indexed (∗):**  
  \[ AOP = \text{opspec} + \text{XR} \quad OP = \text{Memory}[AOP] \]

  (normally \( \text{opspec} \) is a label; i.e. base address + index)

  \[
  \begin{array}{c|c|c}
  \text{if XR = 1:} & a=14 & 45 \\
  \text{load } *a & 15 & -1 \\
  \text{add } *a+1 & 16 & 7 \\
  \end{array}
  \]
PeANUt Addressing Modes – 3

- stack (!): AOP = opspec + SP  OP = Memory[AOP]

  (normally opspec ≤ 0)

  load ! 0 ; AOP = 262  OP = 40  SP → 262
  mul !−2 ; AOP = 260  OP = 15
  store !−3 ; AOP = 259
  OP = 600

- review: different addressing modes have very different effects
- the modes correspond to some high level language construct
PeANUt Assembly Language Format

<label>: operation operand (e.g. =<mode><opspec>)
↑
↑
optional, defines a symbolic address
example: load *<number>
store <label>
add *<label>
sub <label>+<number>
mul !<number>
dvd @<number>
cmp
jmp beq...

● a <number> is +/− decimal or binary integer (or a symbol representing an integer)
● operations are either instructions or directives
## Directives

- **block** $n$:
  - Allocates $n$ memory cells (words) initialised to 0
  - Example: `i: block 1`

- **data** $n$
  - `data <label>`
  - `data "string"` (one cell per character)
  - Allocates the appropriate number of cells and initialises them
  - Example: `text: data "Hello"`

- **end** $<label>$
  - Ends the program here, and define $<label>$ to be the start address of the program
  - (Each PeANUt assembly program must have exactly one)
  - (The **end** directive is at the end of the program)
Translating C into PeANUt – Assignments

- variable declarations and simple assignments: \( n = 1; \) or \( i = n; \)

```c
n:  block  1 ;  int n;
i:  block  1 ;  int i;
...
load  #1 ;  n = 1;  /* AC = 1 */
store  n ;  /* Memory[n] = AC */
load  n ;  i = n;  /* AC = Memory[n] */
store  i ;  /* Memory[i] = AC */
```

- more complex assignments: \( n = n - 1; \)

```c
;  n = n - 1;
load  n ;  /* AC = Memory[n] */
sub   #1 ;  /* AC = AC−1 */
store  n ;  /* Memory[n] = AC */
```
Translating C into PeANUt – Variable Declarations

- variable declarations and simple calculations:

```c
sal: block 1 ; int sal;
tax: block 1 ; int tax;
...
; tax = sal / 12;
load sal ; /* AC = mem[sal] */
dvd #12 ; /* AC = AC/12 */
store tax ; /* mem[tax] = AC */
```

- what if an assignment is even more complex?

  e.g. `x = val - q*42 + x*x;`
Translating C into PeANUt – Simple if

- simple if (assume first instruction is at address 20)

```
load  #0 ; if (0 > x) { /* AC = 0 */
cmp   x ; /* compare AC, mem[x] */
ble   endif1 ;
load  #0 ; x = 0 - x; /* AC = 0 */
sub   x ; /* AC - AC - mem[x] */
store x ; /* mem[x] = AC */
endif1: ; }
```

- what happens to the PSW (for Memory[x]=3)?

```
load #0    → GT=0, EQ=0, PC=21
cmp x      → GT=0, EQ=0, PC=22
ble endif1 → GT=0, EQ=0, PC=26 (sets PC to endif1 if GT=0)
```
Translating C into PeANUt – if .. else if .. else ...

load    sal    ;   if (sal <= 250) {
    cmp    #250    ;
    bgt    elsif2    ;
    load    #0    ;   tax = 0;
    store   tax    ;
    jmp    endif2    ;   }
elsif2:  load    sal    ;   else if (sal <= 500) {
    cmp    #500    ;
    bgt    else2    ;
    ....    ....    ;   tax = sal*20;
    ....    ....    ;
    store   tax    ;
    jmp    endif2    ;   }
else2:    ;   else {
load    sal    ;   tax = sal*30;
mul    #30    ;
store   tax    ;
endif2:   ...

Q: the omitted instructions are: (a) load sal;mul 20   (b) mul #sal;load 20
    (c) load *sal;mul #20   (d) none of these
Translating C into PeANUUt – Some Remarks

- for simple C, we have standard translation patterns
- expression evaluation similar like assignment, except use `cmp` instead of `add, sub` etc
- use the opposite branch instruction to the condition
- use systematically-named branch targets
- for `if`, we need to be able to do a conditional forward branch,
  e.g. `PC = endif, if GT=0`
- for `else if/else`, we also need an unconditional forward branch (`jmp`)
- the PSW plays an important role in all control structures
Translation of Simple I/O into PeANUt

- scan and print characters

```assembly
ch: block 1 ; char ch; /* 16 bits! */

trap #2 ; ch = getchar();
store ch ;

load ch ; printf("%c", ch);
trap #3 ; // same as putchar(ch)
```

- basic I/O is via traps, i.e. the operating system
- in PeANUt, the value $-1$ (0xffffff) gets stored in AC to signify end-of-file (as in C)
- always make sure you print ASCII values (not integer numbers)
- question: how do we do more complex I/O in PeANUt?