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:
  - recent tute/lab solutions;
  - 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:

```
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` $\longrightarrow$ `img`) of program modules will be useful
- we can document assembly language code with high level language code to express a (structured) algorithm

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 (!)

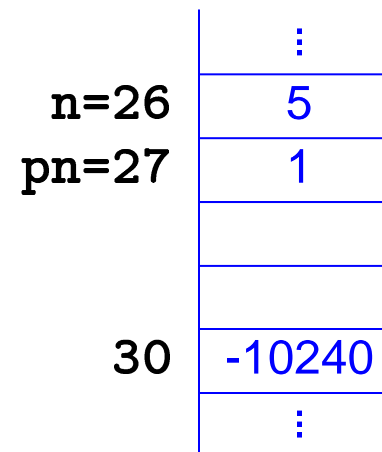
PeANUt Addressing Modes – 1

- immediate (#): OP = opspec

  $-512 \leq \text{OP} \leq 511$)

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

- direct: AOP = opspec OP = Memory[AOP]




```
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 = Memory[opspec]

```
load   @10        ; AOP = 12
OP = 42
```

	⋮
10	12
11	-1
12	42
	⋮

- indexed (∗): AOP = opspec + XR

  (normally opspec is a label; i.e. base address + index)

if XR = 1:

```
load   *a         ; AOP = 15   OP = -1
add    *a+1       ; AOP = 16   OP = 7
```

	⋮
a=14	45
15	-1
16	7
	⋮

PeANUt Addressing Modes – 3

- stack (!): AOP = opspec + SP

  (normally opspec $\leq$ 0)

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

		⋮
	260	15
	261	-1
SP $\longrightarrow$	262	40
		⋮

- 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,      e.g.         e.g.
defines a      load         *<number>
symbolic       store        <label>
address        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

  e.g. `i: block 1`

- **`data`** n

  `data <label>`

  `data "string"` (one cell per character)

  allocates the appropriate number of cells and initialises them

  e.g. `text: data "Hello"`

- **`end`** `<label>`

  end 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;`

```
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;`

```
                     ;    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:

```
sal:  block   1      ;   int sal;
tax:  block   1      ;   int tax;
      ...
                     ;   tax = sal / 12;
      load    sal    ;       /* AC = Memory[sal] */
      dvd     #12    ;       /* AC = AC/12 */
      store   tax    ;       /* Memory[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) {
            cmp     x       ;
            ble     endif1  ;
            load    #0      ;       x = 0 - x;
            sub     x       ;
            store   x       ;
endif1:                     ;    }
```

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

  load #0 $\longrightarrow$ GT=0, EQ=0, PC=21

  cmp x $\longrightarrow$ GT=0, EQ=0, PC=22

  ble endif1 $\longrightarrow$ 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   ;
        load    sal     ;    tax = sal*20;
        mul     #20     ;
        store   tax     ;
        jmp     endif2  ;  }
else2:                  ;  else {
        load    sal     ;    tax = sal*30;
        mul     #30     ;
        store   tax     ;
endif2: ...             ;  }
```

Translating C into PeANUt – 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

```
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 $-\mathbf{1}$ (`0xffff`) 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?