

PeANUt assembly language

- Motivation
- Addressing modes revisited
- Assembly language format
- Translating C into PeANUt
- Reference:
 - Specification of the PeANUt computer (Section 4)

1

COMP2300, 2006

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?

3

COMP2300, 2006

Motivation for assembly language

- Assembly language is 'next level' of abstract machine (after conventional machine and operating system) [Tanenbaum, Figure 1-2]
- 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 a25 if L*M > N
o1 o2 a13          ; a24: store L*M into a13
o65 a1            ; a25: trap 1 (halt)

```

2

COMP2300, 2006

Solutions

- Symbol names for addresses (variables, branch targets) needed
- We also need symbol names for opcodes and modes
- We need a way of defining more complex operations (macros)
- Separate converting (mli → img) of program modules will be useful
- We can document assembly language code with high level language code to express a (structured) algorithm

4

COMP2300, 2006

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 – 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) (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 – 1

- **Immediate: (#)** $OP = <opspec>$

```
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 = ?
```

n=26	5
pn=27	1
30	-10240
	⋮

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 = 257 OP = 600
```

260	15
261	-1
SP → 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, defines a symbolic address	e.g. load	e.g. *<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
- Operations are either instructions or directives

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 assignment **n = n - 1;**

```
; n = n - 1;
load n ; /* AC = Memory[n] */
sub #1 ; /* AC = AC-1 */
store n ; /* Memory[n] = AC */
```

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>**
End program here, <label> is start address of the program
(each PeANUt assembly program must have exactly one **end** directive at the end of the program)

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?

Example: `x = val - q*42 + x*x;`

Translating C into PeANUt – Simple if

- Simple IF (assume first instruction is at address a20)

```

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 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
(ble sets PC to endif1 if GT=0)

```

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
- for **else if/else** we also need an **unconditional** forward branch (**jmp**)
- The **PSW** plays an important role in all control structures

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: ...   ; }

```

Translation of simple I/O into PeANUt

- Scan and print characters

```

ch: block 1   ; char ch; /* 16 bits! */
              ;
trap #2       ; scanf("%c", &ch);
store ch      ;
              ;
load ch       ; printf("%c", ch);
trap #3       ;

```

- Basic I/O is via traps, i.e. the operating system
- Always make sure you print ASCII values (not integer numbers)
- In PeANUt, the value -1 (hFFFF) gets stored to signify EOF
- Question: How do we do more complex I/O in PeANUt?