

## Home work and unsupervised lab this week

- Home work 2 is now online on COMP2300 Web site
- Part 1 on paper (estimated one hour work)
- Part 2 in labs (normal lab times, but no tutor available) (experiments with the PeANUt simulator)
- Do part 1 **before** you go into the labs!
- Home work and lab is very important for assignment 1 (part 3)!

## Further PeANUt

- Traps
  - Input and output
- Conditional branches
  - Condition codes
  - Branch instructions
  - Repetition
- Index register and index addressing mode
- Reference: Specification of the PeANUt computer
- Additional reading: Section 3.6 in Bryant and O'Hallaron

## Traps

- They are operations that cannot be performed with the normal PeANUt instructions (e.g. print, read from keyboard, or stop a program)
- There is a special instruction called **trap**
- It is used to have PeANUt perform a service that is usually a call to the operating system
- Its opcode is 110101, it's operand is the *trap number*
- Depending on the operand of the trap, some particular operation will be performed

## Trap operands

- **Halt:** Tells the PeANUt where your program ends
- **Get:** Allows you to read a character from keyboard
- **Put:** Allows you to print out a character

Action	Opcode	Operand
HALT	110101	1
GET	110101	2
PUT	110101	3

- Extension of the instruction set

## Traps – Input / output example

```
; Simple example machine language program to read and print a line
; Program: echoline.mli

START a10 ; Start address of the program, initialise PC to this

AT a0
0000 0000 0000 0000 ; a0: Memory cell where the character will be stored

AT a10 ; Store the following data items (instructions) into
; memory from a10 onwards
110101 0000 0000 10 ; a10: trap #2 (Get), read character
001 010 0000 0000 00 ; a11: store a0, store the character
000 111 0000 0010 10 ; a12: compare #EOF (End-Of-Line character)
101001 0000 0011 10 ; a13: branch equal a16
110101 0000 0000 11 ; a14: trap #3 (Put), print character
101000 0000 0010 00 ; a15: jump to a10, jump to start of loop
110101 0000 0000 11 ; a16: trap #3 (Put), print EOL character
110101 0000 0000 01 ; a17: trap #1 (Halt), stop the program
```

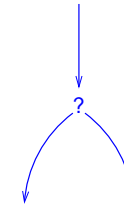
(use this as starting point for assignment 1, part 3)

5

COMP2300, 2006

## Conditional branches

- A conditional branch is a decision point for the instruction sequence
- In C: `if (cond) {...} else {...}`



7

COMP2300, 2006

## More on traps

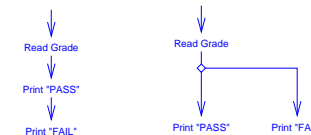
- The term *trap* and the word *exception* are also used to describe the occurrence of a failure condition and PeANUt's response to it (more on traps in lecture P9)
- Important PeANUt initiated traps / exceptions:
  - *Illegal Instruction*: When it is asked to execute an illegal instruction
  - *Illegal Mode*: When it is asked to execute an instruction with an invalid mode
  - *Overflow*: When an arithmetic overflow occurs while the overflow enable (EN) bit in CC (condition codes) is set
  - *Divide by zero*

6

COMP2300, 2006

## Are conditional branches necessary?

- Imagine an instruction sequence that, after reading in a grade, outputs either the word PASS or the word FAIL



- So far, the PeANUt has only been allowed to execute a continuous instruction sequence. Each execution of the program will execute exactly the same instruction sequence
- The machine should be able to behave differently under different situations

8

COMP2300, 2006

## Condition codes

- Condition Codes **GT** (greater than), **EQ** (equal) and **OV** (overflow)
  - Bits within the program status word (PSW) (bits 12, 11 and 10)
  - *Compare* operation sets the CC's
  - **GT**: set to 1 if  $AC > OP$  (operand), otherwise 0
  - **EQ**: set to 1 if  $AC = OP$  (operand), otherwise 0
  - **OV**: set to 1 if last ALU operation resulted in an overflow, else 0
- Branches
  - Instructions allow a new value for the PC to be set
  - Depends on the state of the GT, OV or EQ bits (flags)
- A branch needs to be executed immediately after a compare

## What happens to the PC?

- $PC \leftarrow OP$  (operand), if the branch is taken
- PC remains unchanged otherwise (points to the next instruction)
- Different branches available:

Branch	Taken when
BranchEqual	<b>EQ = 1</b>
BranchNotEqual	<b>EQ = 0</b>
BranchGreater	<b>GT = 1</b>
BranchLessEqual	<b>GT = 0</b>
BranchOverflow	<b>OV = 1</b>
Jump	<b>ALWAYS</b>

## Example

- We want to write a program that performs:  
 $mem[a30] \leftarrow mem[a20] + mem[a25]$ , but only if  $mem[a20] \leq 17$

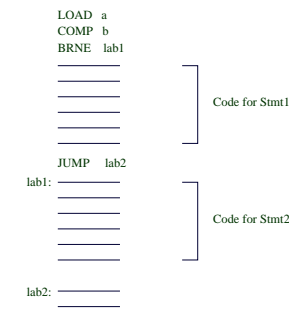
```

AT a5
001 001 0 000 010 000 ; a5: load mem[a20]
000 111 0 000 010 001 ; a6: comp #17 (GT = 1 if AC > 17)
101011 0 000 001 010 ; a7: branch greater (to a12 if GT == 1)
001 011 0 000 010 101 ; a10: add mem[a25]
001 010 0 000 011 000 ; a11: store mem[a30]
... more code ... ; a12: ...

```

## If ... then ... else ...

- We should consider the usual conditional:  
`if (a == b) { <Stmt1> } else { <Stmt2> }`



## Repetition

- Do we need repetition?
- Can programs be written without repetition?
- To do this, we may need programs of (almost) infinite length
- Sometimes we don't know how many times a repetition will be performed (`while` loop)
- So, repetition is *essential* to computation

## Example

- A program that writes out the even letters of the alphabet
  - Reminder: **Trap 3** will result in an ASCII character being output
  - ASCII code for **A** is  $101_8$
- First Attempt (no repetition):

```
START a10
AT a10
000 001 0 001 000 010 ; a10: load #1028
110101 0 000 000 011 ; a11: trap 3 (Print B)
000 001 0 001 000 100 ; a12: load #1048
110101 0 000 000 011 ; a13: trap 3 (Print D)
000 001 0 001 000 110 ; a14: load #1068
110101 0 000 000 011 ; a15: trap 3 (Print F)
000 001 0 001 001 000 ; a16: load #1108
110101 0 000 000 011 ; a17: trap 3 (Print H)
```

(...continued in next slide...)

## Example continued

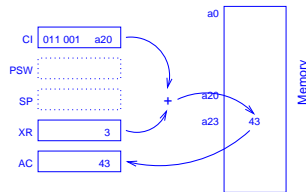
- (...continue from previous slide...)
- ```
000 001 0 001 011 010 ; a40: load #1328
110101 0 000 000 011 ; a41: trap 3 (Print Z)
110101 0 000 000 001 ; a42: trap 1 (halt)
```
- First attempt will need 27 lines
  - Second attempt (using repetition):
- ```
START a10
AT a10
000 001 0 001 000 010 ; a10: load #1028
110101 0 000 000 011 ; a11: trap 3 (Print char)
000 111 0 001 011 010 ; a12: comp #1328 (EQ = 1 if AC == 1328)
101001 0 000 001 110 ; a13: branch equal a16 (to a16 if EQ == 1)
000 011 0 000 000 010 ; a14: add #2
101000 0 000 001 001 ; a15: jump a11
110101 0 000 000 001 ; a16: trap 1 (halt)
```
- Second attempt takes 7 lines, 67 instructions executed

## Index addressing

- Basic data structures
  - Often we want to represent more complex data than single, unrelated values
  - A common data structure is an array (a sequence of related data)
- Why indexed addressing?
  - Access to a sequence of data often involves repetition
  - For example, a program to find the statistical mean of a set of data would have to access a large set of numbers in sequence

## Indexed addressing mode (mode bits 011)

- The address of the operand is given by adding the contents of the Index Register (XR) and the opspec
- The operand is at **mem[XR + opspec]**



## Index register instructions

- **XR** instructions:
    - **SetIndexReg** ( $XR := OP$ )
    - **InclIndexReg** ( $XR := XR + OP$ )
    - **CompareIndexReg** (compare XR with AC, set CC's)
    - **LoadIndexReg** ( $AC := XR$ )
    - **StoreIndexReg** ( $XR := AC$ )
- (OP = operand)

## Index register

- The **index register (XR)** can be used to store a number (an *offset*) to which a base address is automatically added when using the **indexed addressing mode**
- The value in the index register can be set with **SetIndexReg** or copied from the accumulator with **StoreIndexReg**
- The value in the index register can be incremented or decremented by using the **InclIndexReg** instruction

## Index register example

- A program that prints out the characters stored in locations mem[a50] to mem[a76]
- START a10  
AT a10  
110001 0 000 000 000 ; a10: set XR #0  
011 001 0 000 101 000 ; a11: load mem[a50+XR]  
110101 0 000 000 011 ; a12: trap 3  
110010 0 000 000 001 ; a13: increment XR #1  
000 001 0 000 010 111 ; a14: load #27<sub>8</sub>  
1110101 000 000 000 ; a15: compare XR (EQ = 1 if XR == AC)  
101010 0 000 001 001 ; a16: branch not equal a11 (if EQ == 0)  
110101 0 000 000 001 ; a17: trap 1 (halt)