

## Further PeANUt

- Ref: [PeANUt Spec, ]; also [O'H&Bryant, sect 3.6]
- traps:
  - input and output
  - relationship with exceptions
- conditional branches:
  - condition codes (part of PSW register)
  - branch instructions
  - repetition
- index register and indexed addressing mode
- other issues:
  - Assignment 1 deadline OK?
  - calculation of the 10% tutorial/laboratory mark
  - revise 2006 exam Q2(a)–(c) (C Programming)
  - revise PeANUt instruction cycle ('micro-operations') (lectures P1 & P2)

# Traps

- traps 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**, which 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 trap number, some particular action will be performed
  - 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

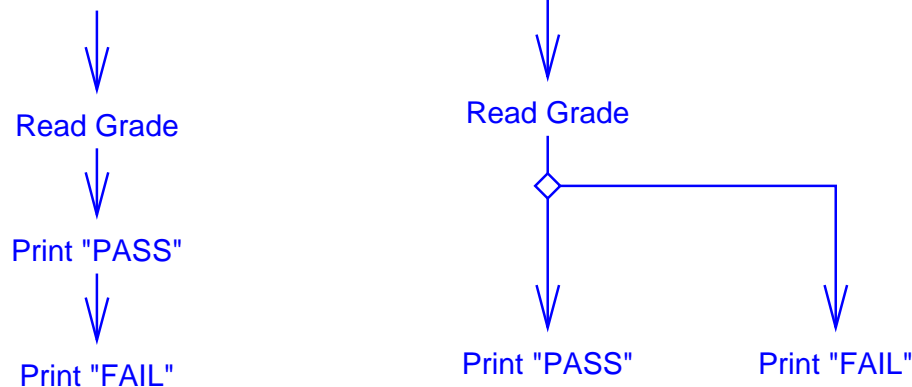


# Traps and Exceptions

- the terms trap and 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 exceptions:
  - 5: Illegal Instruction: when it is asked to execute an illegal instruction
  - 6: Illegal Mode: when it is asked to execute an instruction with an invalid mode
  - 7: Overflow: when an arithmetic overflow occurs while the overflow enable bit (EN) bit in CC (condition codes) is set
  - 8: Divide by Zero
- the same effect can be achieved by executing a trap instruction with the corresponding trap number
- Q: why is this an important mechanism to have in a machine?

## Conditional Branches

- a conditional branch is a decision point for an instruction sequence
- in C: `if (cond) {...} else {...}`
- 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, unchanging over each execution of the program
- the machine must be able to behave differently under different situations

## Condition Codes and Branches

- the condition codes: GT (greater than), EQ (equal) and OV (overflow)
  - bits within the program status word (PSW) (bits 12, 11 and 10)
  - a compare instruction sets the CCs:
    - ◆ GT: set to 1 if  $AC > OP$  (operand), otherwise
    - ◆ EQ: set to 1 if  $AC = OP$  (operand), otherwise
    - ◆ OV: set to 1 if last ALU operation resulted in an overflow, else 0
- conditional branch instructions:
  - allow a new value for the PC to be set, depending on the GT, OV or EQ bits
  - are normally executed immediately after a compare instruction
  - what happens to the PC?
    - ◆  $PC \leftarrow OP$  (operand), if the branch is taken
    - ◆ PC remains unchanged otherwise (points to the next instruction)
  - branches available:

branch	taken when:	branch	taken when:
BranchEqual	EQ = 1	BranchLessEqual	GT = 0
BranchNotEqual	EQ = 0	BranchOverflow	OV = 1
BranchGreater	GT = 1	Jump	ALWAYS

## Conditional Branches Example

- we want to write a program that performs:

$\text{mem}[\text{a30}] \leftarrow \text{mem}[\text{a20}] + \text{mem}[\text{a25}]$ , but only if  $\text{mem}[\text{a20}] \leq 17$

```
001 001 0 000 010 000 ; a5: load mem[a20]
000 111 0 000 010 001 ; a6: compare AC with 17 (GT=1 if AC > 17)
101011 0 000 001 010 ; a7: branch if 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: ...
```

- assume  $\text{mem}[\text{a25}] = 10_{10}$ ; consider execution for:

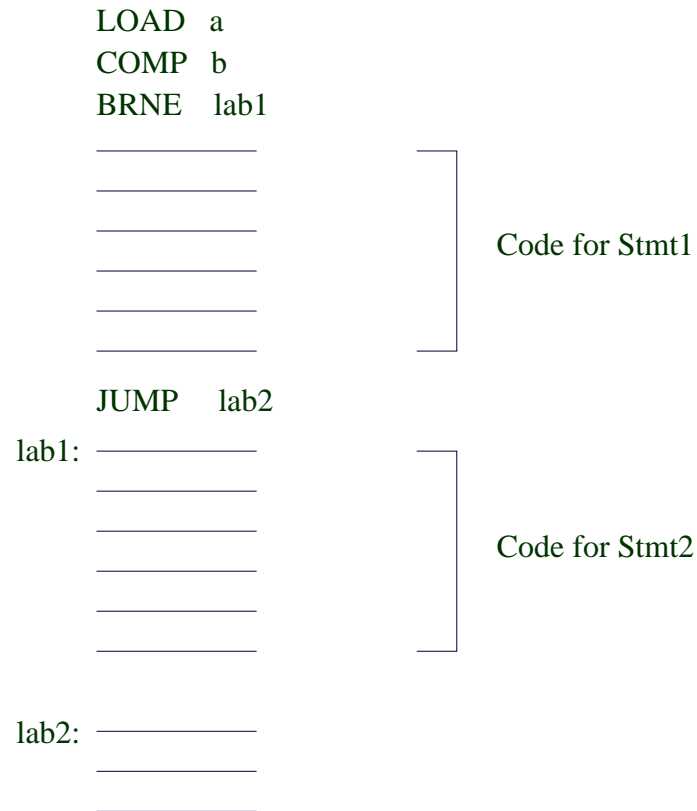
(a)  $\text{mem}[\text{a20}] = 0$

(b)  $\text{mem}[\text{a20}] = 20_{10}$

## Extension to If – Then – Else

- we should consider the usual conditional:

```
if (a == b) { <Stmt1> } else { <Stmt2> }
```



- i.e. at the end of each **<Stmt>**, we insert a jump instruction to go to the end of the conditional

## Repetition

- do we need repetition (loops)?, i.e. can programs be written without it?
  - to do this, we may need programs of (almost) infinite length!
  - we might not know in advance how many iterations to perform (**while** loop)
- 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<sub>8</sub>
- first attempt (no loop, start at a10; will need 27 lines!):

```
000 001 0 001 000 010 ; a10: load 102_8, imm. mode
110101 0 000 000 011 ; a11: trap 3 (print 'B')
000 001 0 001 000 100 ; a12: load 104_8, imm. mode
110101 0 000 000 011 ; a13: trap 3 (print 'D')
000 001 0 001 000 110 ; a14: load 106_8, imm. mode
110101 0 000 000 011 ; a15: trap 3 (print 'F')
000 001 0 001 001 000 ; a16: load 110_8, imm. mode
110101 0 000 000 011 ; a17: trap 3 (print 'H')
...
000 001 0 001 011 010 ; a40: load #132_8, imm. mode
110101 0 000 000 011 ; a41: trap 3 (print 'Z')
110101 0 000 000 001 ; a42: trap 1 (halt)
```

## Loops: Example: printletter-rep.mli

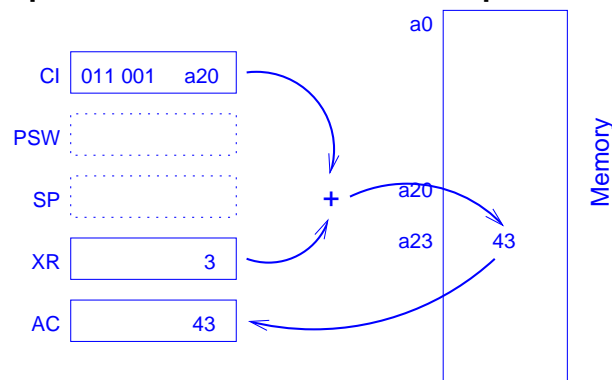
- second attempt takes 7 lines, with 67 instructions executed:

```
START a10      ; start address of the program, init. PC to this

AT a10        ; store these instrns. from a10 onwards
000 001 0 001 000 010 ; a10: load 'B' (=102_8), immediate mode
110101 0 000 000 011 ; a11: trap 3 (put)
000 111 0 001 011 010 ; a12: comp 'Z' (=132_8), immediate mode
101001 0 000 001 110 ; a13: branch equal (to a16, if EQ=1)
000 011 0 000 000 010 ; a14: add 2, immediate mode
101000 0 000 001 001 ; a15: jump a11
110101 0 000 000 001 ; a16: trap 1 (halt)
```

# Indexed 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
  - e.g. 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  
i.e. the operand is at  $\text{mem}[\text{XR} + \text{opspec}]$



## The 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 contents of the index register can be set with SetIndexReg, or copied from the accumulator with StoreIndexReg
- the index register can be incremented or decremented by using the IncIndexReg instruction
- XR instructions:
  - SetIndexReg:  $\text{XR} \leftarrow \text{OP}$
  - IncIndexReg:  $\text{XR} \leftarrow \text{XR} + \text{OP}$
  - CompareIndexReg: compare XR with AC, set CCs)
  - LoadIndexReg:  $\text{AC} \leftarrow \text{XR}$
  - StoreIndexReg:  $\text{XR} \leftarrow \text{AC}$

## Index Register Example: printword-index.mli

- a program that prints out the characters stored in locations mem[a50] to mem[a76]:

```
START a10      ; start address of program, initialise PC to this

AT a10         ; store the following instrns. from a10
110001 0 000 000 000 ; a10: set XR to 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: inc XR by 1
000 001 0 000 010 111 ; a14: load 27_8, immediate mode
1110101 000 000 000 ; a15: comp XR
101010 0 000 001 001 ; a16: bne a11
110101 0 000 000 001 ; a17: trap 1 (halt)

AT a50         ; store chars here (22 chars, last in a76)
000 000 0 001 000 011 ; a50: 'C'
000 000 0 001 001 111 ; a51: '0'
000 000 0 001 001 101 ; a52: 'M'
```

...