

# Department of Computer Science

## The Australian National University

### Specification of the PeANUt computer

## 1 Introduction

This document describes the PeANUt computer, a simple microprocessor created for teaching purposes by the Department of Computer Science at the Australian National University<sup>1</sup>

This document is in three parts. The first part (section 2 of this manual) describes the machine language of the PeANUt machine and corresponds to chapter 5 of Tanenbaum [1] and chapter 2 of Goodman and Miller [2]; the next part (section 3) addresses virtual memory which is the subject of chapter 6.1 of [1]; finally, section 4 of this document (on assembly language) closely parallels chapter 7 of [1] and chapter 10 of [2].

### 1.1 Descriptive Conventions

Throughout this document, the name of machine language instructions, and architectural quantities of the machine (such as registers and memory locations) will be given in *courier* font.

If  $X$  is any unit of storage, such as a memory cell or a register, then the individual bits of that storage unit are regarded as being numbered from zero in ascending order starting at the least significant end of the storage unit (denoted in diagrams as the rightmost bit position in a storage unit). The contents of  $X$ , or part of  $X$ , will be either regarded as an unsigned binary number or as a two's-complement signed binary number, depending on context. In this document the notation  $X[j..i]$  means *bits  $i \dots j$  of storage unit  $X$* . Unless otherwise stated  $X[j..i]$  is treated as an unsigned quantity, with  $X[i]$  being of lesser significance than  $X[j]$ . Thus `memory[3][5..2]` is a 4 bit field commencing at bit position 2 of the memory cell at address 3.

## 2 Machine language

### 2.1 The architecture

The basic structure of the PeANUt microprocessor is given in Figure 1. The processor consists of a memory, a 16 bit arithmetic and logical unit, an addressing unit, an execution unit, a primitive operating system, a set of 16 bit registers and connections to input and output devices. There are 1024 memory cells, with addresses 0 ... 1023. Cells contain 16 bits, and are called `memory[0] ... memory[1023]`. There is a stack, which is based just after the loaded program and data, and grows in the direction of increasing addresses.

### 2.2 Registers

The PeANUt machine has five special registers. Each of these registers is 16 bits in length.

---

<sup>1</sup>Original design and implementation was by Brendan McKay. The addition of virtual memory features is due to Steve Blackburn. This manual is largely the work of Markus Zellner. Other changes to the design, implementation and documentation have been made over the years by Drew Corrigan, Steve Edwards, Peter Farmer, Malcolm Newey, Robin Stanton, Peter Strazdins, Trevor Vickers, Dave Walsh and Peter Christen. The Version 2 implementation is principally the work of Mark Dixon and Brendan Humphries.

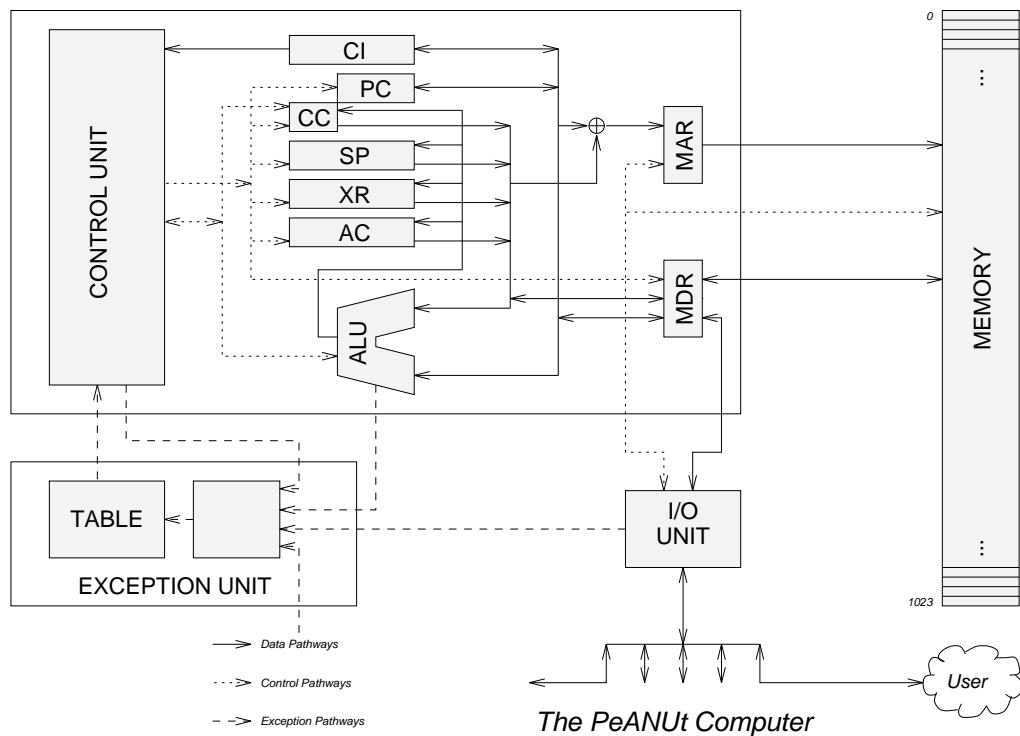


Figure 1: The architecture of the PeANUt microprocessor

AC The accumulator holds data during execution of the program. It is always one of the sources, as well as the destination, of values computed by arithmetic and logical instructions.

CI The current instruction register holds the instruction currently being executed. It cannot be directly referred to by programs.

SP The stack pointer holds the address of the top of the stack.

XR The index register holds the index for indexed instructions.

PSW The program status word contains 16 bits of status information about the program. Many of the parts of the PSW have names of their own; their usage will be described in detail in the next section.

15	14	13	12	11	10	9	0
UN	UN	EN	GT	EQ	OV	PC	

PSW fields	name	meaning
0-9	PC	program counter
10	OV	integer overflow condition code
11	EQ	equals condition code
12	GT	greater than condition code
13	EN	integer overflow enable bit
14-15	UN	unused; permanently zero

The program counter usually contains the address of the next instruction to execute. The bits PSW[12..10] are called the *condition codes* or CCs. The EQ and GT condition codes are set by the comparison instructions, and are tested by the conditional jump

instructions. The *OV* condition code is set if any arithmetic instruction results in integer overflow. The *EN* bit determines whether an arithmetic overflow results in a trap.

Note that the Memory Address Register *MAR* holds the address of the current word (which may be an instruction or data) being fetched inside an instruction cycle, and the Memory Data register holds the contents of corresponding address. Neither of these registers are directly accessible by the instruction set.

### 2.3 The execution algorithm

The execution of the PeANUt computer can be described by the following algorithm.

```
do {
    PC = PC + 1;
    CI = memory[PC-1];
    Evaluate_Operands();
    Execute_Instruction();    /* May cause trap or change PC */
} while (1);    /* Endless loop, condition is always true */
```

The reader might well have expected the first two steps would have been:

```
CI = memory[PC];
PC = PC + 1;
```

which is clearly equivalent in effect. There are technical reasons, to do with the implementation of virtual memory, why the incrementation of the program counter is the first of the four phases of the instruction execution cycle.

The PeANUt computer contains an operating system, which is not directly visible to a program. A program knows of the operating system only as a black box which services traps (this is called the ‘Exception Unit’ in Figure 1). A trap is an exceptional condition which arises either from the deliberate execution of a trap instruction or from an error condition such as attempting to divide by zero or attempting to execute an illegal instruction. The effect of a trap is to return control to the operating system. Depending on what the trap is, the operating system may or may not return control to the program. Full details on traps can be found in Section 2.8.3.

### 2.4 Instruction formats

There are three machine instruction formats, called *Format One*, *Format Two*, and *Format Three*. Each machine instruction (regardless of format) occupies exactly one 16 bit cell of memory.

<i>format</i>	<i>fields</i>			
One	15	13	12	10 9 0
	mode		opcode	
Two	15	10	9	0
	opcode			opspec
Three	15	9	8	0
	opcode			unused

Each of these fields (**opspec** the operand specifier, **opcode** the instruction specifier and **mode** the addressing mode) is considered as an unsigned quantity unless otherwise specified.

The format of an instruction can be determined from the contents of the three most significant bits. **Format One** instructions have the three most significant bits (the **mode** field) equal to 000, 001, 010, 011 or 100, the valid addressing modes, since **Format Two** and **Format Three** instructions cannot begin with these bit patterns. **Format Two** instructions have the three most significant bits equal to 101 and 110. **Format Three** instructions have the three most significant bits equal to 111.

## 2.5 Addressing

The **opspec** and **mode** fields of each machine instruction determine two quantities according to the rules below. One is the **operand OP**; the other is the address of **operand** called **AOP**.

<i>mode</i>	<i>no.</i>	<i>name</i>	<i>meaning</i>
000	0	immediate	AOP is undefined OP is the 16-bit twos-complement value obtained by sign-extending <b>opspec</b> (i.e., by appending 6 copies of bit 9 to the high-order end of OP)
001	1	direct	AOP is <b>opspec</b> OP is <b>memory[AOP]</b>
010	2	indirect	AOP is <b>memory[opspec][9..0]</b> OP is <b>memory[AOP]</b>
011	3	indexed	AOP is <b>opspec + XR[9..0]</b> (ignore overflow) OP is <b>memory[AOP]</b>
100	4	stack	AOP is <b>opspec + SP[9..0]</b> (ignore overflow) OP is <b>memory[AOP]</b>

**Format One** instructions may use any addressing mode. For **Format Two** there is only one possible addressing mode for each instruction (either immediate or direct) and therefore no necessity for a mode field. **Format Three** has no mode field and no **opspec** field and no need for any addressing mode.

## 2.6 Instructions

The instructions of the PeANUt machine can be found listed in Table 1. The instruction formats are described in Section 2.4 and the addressing modes in Section 2.5. Complete details of every instruction is given in appendix A.

The following table defines the modes specified in Table 1.

code	modes	comments
$v$	any mode (0-4)	must be explicitly specified (3 bits)
$v'$	any mode except 0	must be explicitly specified (3 bits)
$f_0$	mode 0 (fixed)	implicitly specified by <b>opcode</b>
$f_1$	mode 1 (fixed)	implicitly specified by <b>opcode</b>
-	no mode is applicable	

## 2.7 Instruction types

### 2.7.1 Data movement

The PeANUt machine is a one address architecture. This means that unless otherwise noted, the destination of most instructions is the accumulator (**AC**) and instructions that require two

<i>format</i>	<i>mode</i>	<i>opcode</i>	<i>name</i>	<i>meaning</i>
One	$v$	000	—	illegal instruction
One	$v$	001	Load	$AC := OP$
One	$v'$	010	Store	$memory[AOP] := AC$
One	$v$	011	Addition	$AC := AC + OP$
One	$v$	100	Subtraction	$AC := AC - OP$
One	$v$	101	Division	$AC := AC / OP$
One	$v$	110	Multiplication	$AC := AC * OP$
One	$v$	111	Compare	compare AC to OP, set CCs
Two	$f_1$	101000	Jump	jump to address AOP
Two	$f_1$	101001	BranchEqual	branch to AOP if EQ=1
Two	$f_1$	101010	BranchNotEqual	branch to AOP if EQ=0
Two	$f_1$	101011	BranchGreater	branch to AOP if GT=1
Two	$f_1$	101100	BranchLessEqual	branch to AOP if GT=0
Two	$f_1$	101101	BranchOverflow	branch to AOP if OV=0
Two	$f_1$	101110	LogicalAnd	$AC := AC \wedge OP$ , bitwise
Two	$f_1$	101111	LogicalOr	$AC := AC \vee OP$ , bitwise
Two	$f_1$	110000	LogicalXor	$AC := AC \oplus OP$ , bitwise
Two	$f_0$	110001	SetIndexReg	$XR := OP$
Two	$f_0$	110010	IncIndexReg	$XR := XR + OP$
Two	$f_0$	110011	IncStackPoint	$SP := SP + OP$
Two	$f_1$	110100	CallProcedure	call procedure at OP
Two	$f_0$	110101	Trap	perform trap number OP
Two	$f_1$	110110	LoadAddress	$AC := AOP$
Three	—	1110000	Return	procedure or interrupt return
Three	—	1110001	ClearOver	$OV := 0$
Three	—	1110010	LoadPSW	$AC := PSW$
Three	—	1110011	StorePSW	$PSW[13..10] := AC[13..10]$
Three	—	1110100	LogicalNot	$AC := \neg AC$ , bitwise
Three	—	1110101	CompareIndexReg	compare XR to AC, set CCs
Three	—	1110110	LoadIndexReg	$AC := XR$
Three	—	1110111	StoreIndexReg	$XR := AC$
Three	—	1111000	LoadStackPoint	$AC := SP$
Three	—	1111001	StoreStackPoint	$SP := AC$

Table 1: PeANUt machine language instructions

operands (the arithmetic and logical instructions) obtain one operand from OP, and implicitly use the accumulator AC as the other source.

Values can be explicitly loaded into the accumulator using the `Load` instruction, and stored into memory from the accumulator using the `Store` instruction. The `LoadAddress` instruction calculates the AOP of the instruction and stores it into the accumulator.

Other data movement is from and to the special registers. The `LoadPSW` and `StorePSW` instructions transfer the value of the PSW (after masking) to and from the accumulator. Similarly the `LoadIndexReg` and `LoadStackPoint` instructions transfer the value of the index register XR or stack pointer SP respectively to the accumulator, while the `StoreIndexReg` and `StoreStackPoint` instructions store the value in the accumulator into the given register. Note that by convention, instructions with the prefix `Load` load into the accumulator from the source, while instructions with the prefix `Store` store from the accumulator into the destination. The `SetIndexReg` instruction allows an immediate value to be loaded directly into the index register XR.

### 2.7.2 Dyadic instructions

The PeANUt dyadic (two operand) instructions implicitly use value of the accumulator as the first operand of an instruction, and use `OP` as given in Section 2.5 as the second operand. The result is always stored into the accumulator. The dyadic instructions are either arithmetic or logical.

All arithmetic instructions (`Addition`, `Subtraction`, `Division`, `Multiplication`) operate on 16 bit twos complement integers. There are no instructions for operating on unsigned integers, or floating point numbers in the PeANUt instruction set. If any of the arithmetic operations overflow the `OV` bit in the `PSW` will be set, and if the `EN` bit in the `PSW` is set, an `OverFlow Trap` will be generated. Division may also cause a `Divide by Zero Trap` if `OP` is equal to zero.

The logical instructions (`LogicalAnd`, `LogicalOr`, `LogicalXor`) also operate on 16 bit quantities and perform the logical operation bitwise between the value of the accumulator `AC` and `OP`, and leave the result in `AC`.

### 2.7.3 Monadic instructions

The PeANUt machine's monadic (one operand) instructions are `LogicalNot`, `IncIndexReg`, `IncStackPoint` and `ClearOverflow`. The `LogicalNot` instruction performs a bitwise negation of the accumulator and returns the result into the accumulator. The `IncIndexReg` and `IncStackPoint` instructions allow an immediate value to be added to the Index register or Stack Pointer respectively. The `ClearOverflow` instruction sets the `OV` bit of the `PSW` to zero.

### 2.7.4 Comparisons, conditional jumps and jumps

The PeANUt comparison and conditional jump model relies on the condition codes in the `PSW`. The comparison instructions `Compare` and `CompareIndexReg` compare `AC` and `OP` or `XR` and `AC` respectively, and set the `EQ` and `GT` bits in the `PSW` based on the result of the comparison. The conditional branch instructions `BranchEqual`, `BranchNotEqual`, `BranchGreater`, `BranchLessEqual`<sup>2</sup> and `BranchOverflow` then jump to the address `AOP` if the `EQ` and `GT` condition codes are set in the appropriate way (or in the case of `BranchOverflow`, if the `OV` bit is set). The `Jump` instruction is an unconditional jump to the address `AOP`.

### 2.7.5 Loop control

There are no explicit loop control instructions in the PeANUt instruction set. Loops must be explicitly coded using the comparison and conditional branch instructions.

### 2.7.6 Procedure calls

The PeANUt machine has two instructions that support a procedure call mechanism, namely the `Call` and `Return` instructions. The `Call` instruction pushes the current `PC` onto the stack and increments the stack pointer, then sets the `PC` to the value of `OP`. The `Return` instruction loads the value at the current top of stack into the `PC`, and decrements the stack pointer by one. The procedure call and return mechanism, and the default parameter passing mechanisms, are discussed fully in Section 2.8.2.

### 2.7.7 Input/Output

The PeANUt machine has no specific programmed I/O instructions. All input and output is done by trapping to the operating system, which is described in Section 2.8.3.

---

<sup>2</sup>There is no *branch if less than* instruction in the PeANUt instruction set; the `BranchGreater` and `BranchEqual` instructions used together make it unnecessary.

## 2.8 Flow of control

### 2.8.1 Sequential flow of control and jumps

As can be seen from the execution algorithm of the PeANUt machine, the normal flow of control proceeds through sequential memory addresses, unless one of the branch instructions, or a `Jump`, `Call`, `Return` or `Trap` instruction is executed, or an exception occurs.

### 2.8.2 Procedures

The nature of the `Call` and `Return` instructions determine to some extent the procedure calling conventions. Beyond that the PeANUt programmer is free to adopt whatever procedure linkage conventions are desired. It is recommended, however, that the following conventions are used<sup>3</sup>. A procedure has the following appearance in PeANUt assembly code (see Section 4).

```
name:           ; called procedure prologue
                ;   (allocate stack space for local variables)
                ; body of procedure
                ; called procedure epilogue
                ;   (fill in return value, if any)
                ;   (deallocate local variable stack space)
ret             ;
```

Here `name` represents the name of the procedure, and is also a label marking the first instruction of the procedure. The `ret` (`Return`) instruction is the end of the procedure, and results in execution returning to the caller of the procedure. A procedure call consists of the caller's part of the procedure prologue, followed by a call to the procedure, and the caller's part of the procedure epilogue.

```
                ; caller prologue
                ;   (allocate space for return value, if needed)
                ;   (push parameters onto the stack)
call name      ; call to procedure
                ; caller epilogue
                ;   (remove parameter stack space)
                ;   (remove return value, if any)
```

Arguments are passed to a procedure (as part of the caller's procedure prologue) by pushing them onto the stack prior to calling the procedure. Data is pushed onto the stack by incrementing the stack pointer and storing the value to memory using stack addressing mode. It is the responsibility of the caller as part of the caller's epilogue to remove arguments from the stack after the call, leaving the stack in its initial (pre-prologue) state.

As part of the called procedure's prologue, storage space on the stack for local variables can be allocated by incrementing the stack pointer. Local variables may then be referenced using stack addressing mode. As part of the called procedure's epilogue, it is the responsibility of the called procedure to remove local variables from the stack before returning, leaving the stack in its initial state at the beginning of the procedure, before the prologue. Also as part of the epilogue, the called procedure must return any values it has calculated to the caller. A procedure may do this by leaving the value in a register (such as `AC`), passing a return value on the stack, or by using variable parameters.

When a procedure is called using the `Call` instruction, the value of the PC (known as the return address, indicating the instruction following the call) is pushed onto the stack,

---

<sup>3</sup>The supplied libraries use these procedure linkage conventions.

by incrementing the stack pointer `SP` and storing the `PC` at that location. The `PC` is set to the address of the first instruction of the procedure, and execution continues. The `Return` instruction pops the value at the top of the stack, by reading the value in `memory[SP]` and decrementing `SP` and setting the `PC` to that popped value. This has the effect of continuing execution at the instruction after the original `Call` instruction. It should be apparent that the procedure prologue and epilogue for both the caller and the called procedure should take care to leave the stack in the appropriate condition.

### 2.8.3 Traps

The PeANUt computer has no operating system visible to the programmer; all interaction with input and output devices, and all handling of erroneous conditions is done by accessing the ‘black box’ operating system via the trap mechanism. We will refer to it as ‘the operating system’ because it supports typical operating system functionality but it is really part of the control unit of PeANUt.

There are two types of trap; those caused by exceptional conditions (such as attempting to divide by zero), and ones caused deliberately with the `Trap` instruction. The first type of trap will also be called an *exception* if the distinction needs to be made. The second type will be termed a *service request* trap. Requests for service via a trap instruction can be handled by built-in functionality defined for PeANUt or by trap service routines supplied by the programmer.

All traps have associated with them an integer in the range 0..511; not surprisingly it is called the trap number and it is this number that forms the operand for trap instructions when they are used.

Full details of the algorithm used to handle traps by the operating system are given in Appendix B.

**2.8.3.1 Standard Service Traps** There are several features of PeANUt (including its I/O and virtual memory) that are accessed via service request traps. For example, the trap instruction with the argument 2 is PeANUt’s way of reading a character. The various services that are predefined in PeANUt are listed in the following table which gives the trap numbers of those service traps as well as a short description. In each case, (except halt) control returns to the program at the location following the trap instruction, provided it is successful. Under some circumstances, however, a failure may occur that results in the program being aborted.

**2.8.3.2 PeANUt exceptions** The following table lists all the exceptions that can be generated by the PeANUt hardware in response to an instruction mis-behaving. In each case, the exception name, trap number and the instructions that may cause the exception are given.

In all cases except that of the page fault, the normal action for the program is to abort the program with the appropriate error message.

<i>exception name</i>	<i>trap number</i>	<i>possible instructions</i>
Data Error	4	Get or Put
Illegal Instruction	5	Any illegal instruction
Illegal Mode	6	Store
Overflow	7	Add, Sub, Div, Mul
Divide by Zero	8	Division
Trapping Error	10	Trap instruction or a bad trap service routine
Page Fault	11	Page fault

<i>trap name</i>	<i>trap number</i>	<i>normal effect</i>
Halt	1	The operating system will terminate the program normally. This is the normal way for programs to exit.
Get	2	The operating system will attempt to read one character from standard input. If the read succeeds, the operating system will place the ASCII code of that character in AC[6..0], zero into AC[15..7]. If the read fails because of end-of-file, the operating system will place the number -1 in AC and return control to the program. If the read fails for any other reason, the operating system will ignore the Get trap and proceed as if a Data Error trap had been generated instead.
Put	3	The operating system will attempt to write bits AC[6..0] as an ASCII character to the output device. If the write fails, the operating system will ignore the Put trap and proceed as if a Data Error trap had been generated instead.
Establish Trap Routine	9	The operating system takes AC[9..0] to be the address of a Trap Table Entry. See the full description in Section 2.8.3.3.
Swap Page In / Out	12 / 13	See Section 3 for details if Virtual Memory mode is in use; otherwise, these traps are predefined to have no effect.

Table 2: PeANUt service traps

The **Page Fault** exception occurs only if Virtual Memory mode of the PeANUt Machine (Section 3) is in use, and can occur in the fetch of the instruction or the evaluation of its operand. If Virtual Memory is not in use, this trap is pre-defined to have no effect.

**2.8.3.3 Programmer-supplied Trap Service Routines** The **Establish Trap Routine** trap allows the user to vary the way in which the PeANUt system responds to a trap. For example, it is possible to tell the operating system to ignore **Overflow** exceptions, to pass control to a specified user-supplied procedure when an **Overflow** exception occurs, or to create an entirely new trap routine. To use the **Establish Trap Routine** service call, it is necessary to create a **Trap Table Item (TTI)** consisting of two contiguous memory cells as given in the following diagram. If the **Establish Trap Routine** is successful then the information in the TTI will become an entry in the PeANUt trap table.

cell 0	trap number
cell 1	trap routine address

- 1 Place the trap number in cell 0 of the TTI
- 2 Place the required value in cell 1 of the TTI (see below)
- 3 Load the address of the TTI into AC.
- 4 Execute the **Establish Trap Routine** trap.

A trap number not in the range 0..511 or one of 1, 9, 10 and 11 in the first cell will cause step 3 to be converted into a **Trapping Error** trap. The second cell (cell 1) of the TTI must be filled with one of the following three possibilities.

<i>value</i>	<i>action</i>
-2	The operating system will ignore the trap silently.
-1	The operating system will take the default action for this trap (as if the trap table for PeANUt had never been changed).
an address	The operating system will call this procedure (called the <i>trap service routine</i> ) when the trap occurs. The procedure should conform to the conventions given in Section 2.8.2.

To modify the behaviour of a trap previously established via the `Establish Trap Routine` trap, the `Establish Trap Routine` trap must be re-executed with a new TTI.

### 3 Virtual Memory

This is an optional mode of execution in the PeANUt machine. Note that a PeANUt image file will have to be specially initialized to be used in this mode.

First a reminder of what virtual memory is.

#### 3.1 Virtual Memory Overview

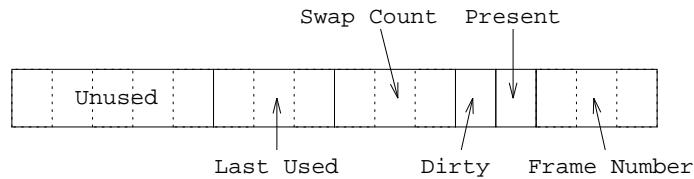
Virtual memory is a technique used in computer architecture to increase the memory available to the user while optimising cost and speed. The underlying problem is that there exists a speed  $\leftrightarrow$  space trade off; for a given amount of money you can usually *either* get a small amount of fast memory or a large amount of slow memory. Being human, we normally want the best of both: a *large* amount of *fast* memory.

The basic approach is to place all of our data in the slow memory and to have a working copy of the most frequently required data (what ever that may be) in the fast memory. The operating system, together with the machine's hardware must be able to decide which subset of memory should be in fast memory (main memory) and is responsible for automatically copying data from the slow memory (secondary memory) to the fast memory (main memory) whenever the CPU needs it.

#### 3.2 PeANUt Virtual Memory

Some important points about the PeANUt's Virtual Memory:

- The PeANUt's address space is divided into 32 pages. Each page has 32 cells. A 10 bit address can be thought of as being composed of two 5 bit parts. The first 5 bits denote which page the address is in, the second five bits denote which cell within that page the address refers to. Note that  $2^5 = 32$  and  $2^5 \times 2^5 = 1024$ , which is the number of cells in the PeANUt address space.
- The PeANUt (when operating in Virtual Memory mode) has a main memory of 256 cells. The main memory can hold exactly 8 pages ( $8 \times 32 = 256$ ). Main memory is composed of 8 'slots' called *page frames*. Each page frame can hold a page. The PeANUt uses a page table to determine whether a particular address is in main memory or not at any given time.
- The PeANUt has a page table which is permanently located in page 0. The page table holds all the information regarding the state of a page. The page table has 32 entries, each entry relating to one of the 32 pages. The format of each page table entry is as follows:



The meaning of the fields is as follows:

**Present** It states whether the page is present (in main memory) or not. A value of 1 indicates that the page is present, a value of 0 indicates that the page is not in main memory. *The remaining fields are only meaningful for pages that are present.*

**Frame Number** This states which page frame (slot in main memory) a page is in *if* the page is present.

**Dirty** A value of 1 indicates that data has been written to that page since it was swapped into main memory.

**Swap Count** This field indicates how many page faults have occurred since the page was swapped into memory. This tells us how long a page has been in main memory for. The page with this field set at 0 is the newest. This will be useful later on when we need to choose pages to remove to make space in main memory.

**Last Used** This field is continuously updated and reflects how recently a page was accessed (for a read or a write). A value of 0 indicates that this page is the most recently accessed. An interesting note: whenever you check this field for page 0, the value will *always* be 0! Why is this? Hint: to check this field, you must read the page table entry. Check your ideas with your tutor.

### 3.3 Traps required for Virtual Memory

The **Page Fault** exception (trap 11, see Section 2.8.3) is not user-initiated or user-definable. Two new user-initiated traps have been predefined to allow the user to ask the operating system to move an entire page to and from main memory.

<i>trap name</i>	<i>trap number</i>	<i>normal effect</i>
Swap Page In	12	The operating system will move page number AC from secondary memory to main memory. The operating system will use page table entry AC (at <code>mem[AC]</code> ) to determine the destination page frame.
Swap Page Out	13	The operating system will move page number AC from main memory to secondary memory.

Table 3: PeANUt predefined (non-exception) traps for Virtual Memory

### 3.4 Initialisation

The user is responsible for some rather important aspects of the virtual memory operation. **When using virtual memory on the PeANUt, the user must take special steps in initialisation:**

- The user must ensure that page table information is placed in page 0 (addresses `a0` to `a37`). Note that only entries for the pages that are initially in main memory need to be defined; if a page is not in main memory, its present bit is set to 0 and all other bits are ignored, so a blank entry is fine. You will soon see an example of how to initialise the page table.

- The user must provide a ‘page fault service routine’. This routine will tell the PeANUt what to do when it tries to access data that is not in main memory. If the routine is not sensible, the PeANUt won’t work! This routine *always* starts at address **a46**. **a46** is the seventh cell in the second page. You might think this is a strange place to start. The ‘page fault service routine’ starts at the seventh cell of the second page because the second page is as good a place as any *and* the first six cells of the second page are reserved for something else (to be discussed later).
- The user must take care to ensure that any programs they write do not interfere with the data in pages 0 and 1. As described above, the data in these pages is essential to the page swapping operation. If this data is accidentally modified, the PeANUt will probably come to a grinding halt. Thus all code the user writes (apart from the ‘page fault service routine’ described above) should never write to addresses **a0** to **a77**.

Of course, if you want the PeANUt to do anything useful, you will also have to provide a normal PeANUt program somewhere in the `.mli` file. As with the standard PeANUt, the `START` directive sets the initial PC value (the place where execution begins). Because the first *and* second pages are reserved (addresses **a0** to **a77**), your program must be written somewhere in the remaining 30 pages.

### 3.5 Normal operation

It is important to remember that virtual memory is supposed to be largely invisible to the user. Apart from the problems of writing the ‘page fault service routine’, initialising the page table and remembering to avoid writing data to the reserved pages (0 and 1), using the PeANUt does not change much.

The operation of virtual memory is only really noticed when the CPU tries to get data from a page that is not in main memory.

When using virtual memory, the steps of a PeANUt memory access are as follows:

1. Split the 10 bit address into a 5 bit page number and a 5 bit page offset.
2. Look at the appropriate page table entry.
3. Check the **Present** bit for that page table entry. If the page is present (in main memory) go to step 6, otherwise continue.
4. Call the page fault service routine.
  - Because the page fault service routine may upset contents of the registers, the contents of each of the five registers are stored to addresses **a40–a44**.
  - The page fault service routine must know which page is wanted, so the page number is written to **a45**.
  - The page fault service routine is called.
  - Upon completion of the page fault service routine the values of the registers are restored from addresses **a40–a44**.
5. The desired page should now be in main memory. Try the access again (go to step 1), this time it should work.
6. Find the page frame number for the page. This is in the page table entry.
7. Combine the page frame number with the page offset to produce an 8 bit address.
8. Read (or write) the data, from (or to) main memory using the 8 bit address.

You don't need to remember all of this, because this is done automatically by the PeANUt machine. You only need to supply the page fault service routine mentioned in step 4.

In contrast, to access memory when the PeANUt is *not* using virtual memory, the following is done:

1. Use the 10 bit address to read (or write) data to (or from) main memory.

Clearly there is a lot more work to be done when virtual memory is used. Generally this is considered acceptable because of the great advantage of being able to use cheap secondary storage to make main memory appear to be much bigger than it really is.

## 4 Assembly language

This section describes the assembly language of the PeANUt machine, and the tools that allow assembling, linking and execution of PeANUt programs. Note that the PeANUt machine has no hardware implementation, and so the execution of the machine is simulated by a software program.

### 4.1 Introduction and notation

The PeANUt machine is programmed in PeANUt assembly language. A file containing PeANUt assembly language is assembled by `assemble` to produce a relocatable file. The linker `join` takes a number of relocatable files and links them into an executable file. The executable file can then be executed using the program `execute`. The program development cycle is shown diagrammatically in Figure 2. Throughout the next sections, the names of assembly language operations and other assembler objects will be given in `courier` font.

### 4.2 The assembly process

The PeANUt assembler is a standard two pass assembler as described in [1]. The first pass of the assembler does lexical and syntax analysis of the assembly language program, and builds the symbol table. If no errors have occurred, the second pass uses the symbol table information to satisfy references to symbols, and to produce the relocatable file.

#### 4.2.1 Pass one

The first pass of the assembler analyses the syntax of the assembly language file (after macro expansion) that is specified in the following sections.

##### 4.2.1.1 Assembler syntax

- Blank lines are ignored.
- Upper and lower case are equivalent in instruction names and labels.

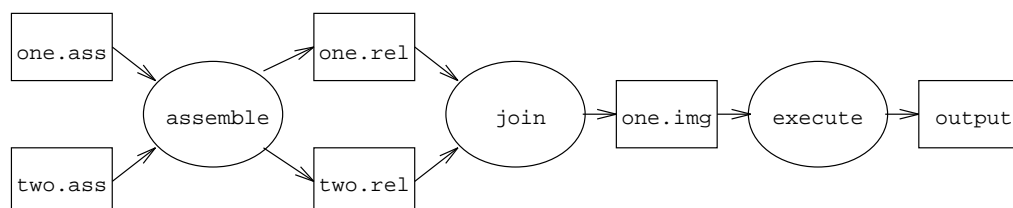


Figure 2: The PeANUt program development environment

- Identifiers consist of a letter followed by zero or more letters or digits. There is no limit on the length of an identifier, but only the first 6 characters are significant.
- Integers can be written as a decimal value, like `-374`, or as a character like `'x'` (including the quotes). The latter form represents the unsigned integer ASCII code for the character. Thus `'x'` is the same as 120. There are also some special forms as shown in the following table.

<i>form</i>	<i>ASCII value</i>	<i>name</i>
<code>'\\'</code>	92	backslash
<code>'\"'</code>	34	double quote
<code>'\''</code>	39	single quote
<code>'\n'</code>	10	newline
<code>'\r'</code>	13	linefeed
<code>'\t'</code>	9	tab

Integers may also be specified in binary, in which the binary number must be preceded by `'%'`. Eg. `'%01110'` is equivalent to decimal 14. Up to 16 binary digits may be specified, with spaces in between binary digits allowed. If 16 binary digits are specified, and the leftmost digit is a `'1'`, a negative number is denoted, under the convention of 16-bit 2's complement arithmetic.

Binary integers may also be signed: a preceding `'+'` sign has no effect, whereas a preceding `'-'` sign *negates* the value denoted by the binary digits<sup>4</sup>.

- A string constant consists of zero or more characters between double quotes. If a double quotes character is to appear in the string, it must appear as `\"`. The other ligatures beginning with `\` shown above are also permitted, as is `\0`, the null character (which can be used to terminate strings). Any other appearance of `\` is illegal.  
Example: `"She said, \"Come here\".\n"`

- Comments consist of everything from a semicolon to the end of the line.

The general form of an assembler statement is

```
label:      operation      operand
```

The `label` field is optional. If it is absent, the `:` (colon) must also be absent. The `operation` field contains the name of a machine instruction or of an assembler directive. The `operand` field specifies the operand, if any. Those operations which require an operand must have an `operand` field. If no operand is required, none is allowed.

**4.2.1.2 Instruction operations** The following table gives the assembler operations that correspond to the machine language instructions of the PeANUt machine.

---

<sup>4</sup>Note that the binary digits alone can denote a negative number if the leftmost of 16 binary digits is a `'1'`; in this case, the preceding `'-'` sign will signify a positive value. Eg. `-%1000 0000 0000 0001` denotes  $-(-32767) = 32767$ .

<i>machine instruction</i>	<i>operation</i>	<i>machine instruction</i>	<i>operation</i>
Load	load	SetIndexReg	setxr
Store	store	IncIndexReg	incxr
Addition	add	IncStackPoint	incsp
Subtraction	sub	CallProcedure	call
Division	dvd	Trap	trap
Multiplication	mul	LoadAddress	loada
Compare	cmp	Return	ret
Jump	jmp	ClearOver	clov
BranchEqual	beq	LoadPSW	ldpsw
BranchNotEqual	bne	StorePSW	stpsw
BranchGreater	bgt	LogicalNot	not
BranchLessEqual	ble	CompareIndexReg	cmpxr
BranchOverflow	bov	LoadIndexReg	loadxr
LogicalAnd	and	StoreIndexReg	storexr
LogicalOr	or	LoadStackPoint	loadsp
LogicalXor	xor	StoreStackPoint	storesp

**4.2.1.3 Addressing mode syntax** In the following table the symbol `number` stands for a decimal integer with optional sign, and `label` stands for the name of a label defined in the current source file, or a name declared in an `external` statement, or the character `.` (period). The legal forms for instruction operands are shown in the following table.

<i>mode</i>	<i>syntax</i>
immediate	<code>#number</code>
direct	<code>label</code> <code>number</code> <code>label+number</code> <code>label-number</code>
indirect	<code>@label</code> <code>@number</code> <code>@label+number</code> <code>@label-number</code>
indexed	<code>*label</code> <code>*number</code> <code>*label+number</code> <code>*label-number</code>
stack	<code>!label</code> <code>!number</code> <code>!label+number</code> <code>!label-number</code>

The operand must have a mode appropriate to the instruction (see Section 2.6). The value of the symbol `.` (period) is the address of the current instruction.

**4.2.1.4 Assembler directives** In addition to machine instructions the PeANUt assembler has the following assembler directives <sup>5</sup>.

**data** This allocates one cell of memory and initialises its contents according to the **operand**. Legal operands are a decimal/binary number in the range `-32768 ... 32767` or a label

---

<sup>5</sup>also called pseudo operations.

name. In the latter case, the least significant 10 bits are set to the address of the label and the most significant 6 bits are set to zero.

The **operand** may also be a string, in which case one cell of memory is allocated per character in the least significant seven bits and zero in the most significant nine bits. If a null character is desired to terminate the string, it must be explicitly included in the string.

**block** There must be an **operand** consisting of a decimal integer in the range 0 ... 1023. The specified number of cells are allocated and initialised to zero. If there is a label field, this refers to the first cell (if any) or to the next cell (if the **operand** is 0).

**end** There is an optional **operand** consisting of a label name. If present, the label gives the starting address for the program. The label must be defined in this file; external names are not allowed. Any statement after an end statement is ignored. No label field is allowed.

**macro** This directive marks the start of a macro definition. See Section 4.2.1.6 for details. No label field is allowed.

**endmacro** This directive marks the end of a macro definition. See Section 4.2.1.6 for details. No label field is allowed.

**global** There must be an **operand** consisting of a label name. It must be a label which is defined in this source file. The effect is to notify the linker that this label is global, i.e. visible from other source files. If there is more than one global declaration for the same label, all but the first are silently ignored. No label field is allowed.

**external** There must be an **operand** consisting of a label name. If a label of this name is defined in this source file, the external declaration is ignored. Otherwise, the label is marked as an external symbol. All external symbols must be explicitly declared as such. If there is more than one external declaration for the same label, all but the first are silently ignored. No label field is allowed.

**4.2.1.5 Arbitrary restrictions** The PeANUt assembler imposes the following arbitrary limits on quantities within an assembly language file.

maximum line length	255 characters
maximum number of tokens on one line (includes lines resulting from macro expansion)	50 tokens
maximum number of different labels (including external)	1024 labels
maximum number of macro names	100 macros
maximum depth of macro expansion	20 nestings

**4.2.1.6 Macros** Macros definitions are used to define sequences of one or more instructions and their operands, and can be parameterised. Macro calls textually insert the body of the macro definition, after parameter substitution. A macro definition has the following form.

```
macro name(parameter-list)
    operation operand
    .
    .
    operation operand
endmacro
```

The `parameter-list` consists of from zero up to eight formal parameters, separated by commas. If there are no parameters, the parentheses are optional. A parameter name can be any legal identifier. The body of the definition can contain any PeANUt assembler statements except `macro` or `endmacro`. A macro call has the following form.

```
name(parameter-list)
```

where `name` is the same as on the macro definition and the `parameter-list` consists of actual parameters separated by commas. The number of actual parameters in the macro call must match the number of formal parameters in the definition. If there are no parameters, the parentheses are optional. Actual parameters can consist of any sequence of identifiers, numbers (including integers given as characters), the characters `@`, `*`, `.` (period), `#`, `!`, or string constants. The following concise form of macro may be used to allow a define constants by assigning a number to a symbolic name.

```
name = number
```

Macros must be defined before they are used. A macro definition overrides any previous definition with the same name or any assembler operation or assembler directive with the same name. Macro calls may not occur to a depth of more than 20 (a macro call at the top level counts as “depth 1”; any macro call in the expansion is on “depth 2”, and so on).

#### 4.2.2 Pass two

The responsibility of the first pass of the assembler is to analyse the syntax of the assembly language program, and build a symbol table containing all the symbols found in the program. In the symbol table is also stored the class of the symbol (whether the symbol is *relocatable*, `global` or `external`) and if `relocatable` or `global`, the offset of the symbol from the beginning of the file. This symbol table information is used in the second pass of the assembler to check whether symbols used in operands are valid, and if valid, to supply the offset from the beginning of the file to allow formation of a relocatable record for the current instruction. The second pass of the assembler also writes records into the relocatable file for each operation and for some assembler directives in the assembly language file. The format of relocatable files produced by the PeANUt assembler is given in Appendix D.

### 4.3 Linking and loading

The PeANUt linker (called `join`) reads one or more relocatable files each containing one *module* (the result of assembling one assembly language source file). If there are no errors, it writes one image (executable) file. The linking process is described briefly as follows.

- The modules are allocated base addresses according to their sizes. The first module has base address 0. The second module has base address equal to the size of first module. The third module has base address equal to the sum of the sizes of the first two modules and so on.
- The values of all `global` symbols are computed.
- The values of all `external` symbols are determined by matching their names against those of the global symbols.
- The initial contents of memory are constructed using relocation or external symbols as required.
- The image file is written.

Typical error conditions include

- illegal relocatable file contents
- more than 1024 cells of memory allocated altogether
- duplicate global symbol
- unsatisfied external symbol (i.e., no matching global symbol)
- no starting address, or more than one starting address

## References

- [1] Andrew S. Tanenbaum, *Structured Computer Organization (4rd edition)*. Prentice-Hall, 1999, ISBN: 0-13-095990-1
- [2] James Goodman and Karen Miller, *A Programmer's View of Computer Architecture*. Saunders College Publishing, 1993, ISBN: 0-03-097219-1

## A Detailed instruction descriptions

<illegal>

opcode: 000 (Format One)  
modes: any  
CCs: none  
traps: illegal operation  
notes: This instruction is illegal.

**Load** (load)

opcode: 001 (Format One)  
modes: any  
CCs: none  
traps: none  
notes: Load OP into AC.

**Store** (store)

opcode: 010 (Format One)  
modes: direct, indirect, indexed, stack  
CCs: none  
traps: illegal mode (if mode=immediate)  
notes: Store AC into memory [AOP].

**Addition** (add)

opcode: 011 (Format One)  
modes: any  
CCs: OV := 1 if overflow occurs  
traps: **Overflow** (if overflow occurs and EN=1)  
notes: Add OP to the value in AC, leaving the result in AC. Both quantities are treated as signed twos-complement numbers. If overflow occurs, store the least significant 16 bits of the 17 bit result in AC, set OV to 1, then if EN=1 generate an **Overflow** exception.

**Subtraction** (subtract)

opcode: 100 (Format One)  
modes: any  
CCs: OV := 1 if overflow occurs  
traps: **Overflow** (if overflow occurs and EN=1)  
notes: Subtract the OP from the value in AC, leaving the result in AC. Both quantities are treated as signed twos-complement numbers. If overflow occurs, store the least significant 16 bits of the 17 bit result in AC, set OV to 1, then if EN=1 generate an **Overflow** exception.

**Division** (divide)

opcode: 101 (Format One)  
modes: any  
CCs: OV := 1 if overflow occurs  
traps: **overflow** (if overflow occurs and EN=1)  
**Divide by Zero** (if OP=0)  
notes: Divide the value of AC by OP, leaving the integer part of the result in AC and discarding the remainder. Both quantities are treated as signed twos-complement numbers. If OP=0, leave AC unchanged and generate a **Divide by Zero** exception. If overflow occurs (only possible for  $-32768/-1$ ), store the least significant 16 bits of the 17 bit result in AC, set OV to 1, then if EN=1 generate an **Overflow** exception.

**Multiplication** (multiply)

opcode: 110 (Format One)  
modes: any  
CCs: OV := 1 if overflow occurs  
traps: **Overflow** (if overflow occurs and EN=1)  
notes: Multiply OP by the value in AC, leaving the result in AC. Both quantities are treated as signed twos-complement numbers. If overflow occurs, store the least significant 16 bits of the 32 bit result in AC, set OV to 1, then if EN=1 generate an **Overflow** exception.

**Compare** (compare)  
opcode: 111 (**Format One**)  
modes: any  
CCs: EQ and GT will be affected  
traps: none  
notes: Compare the contents of AC to OP, treating both as signed numbers. Indicate the result of the comparison by setting the condition codes EQ and GT as follows.  
AC < OP: EQ := 0, GT := 0  
AC = OP: EQ := 1, GT := 0  
AC > OP: EQ := 0, GT := 1

**Jump** (jump)  
opcode: 101000 (**Format Two**)  
modes: direct  
CCs: none  
traps: none  
notes: Replace the contents of PC by AOP.

**BranchEqual** (branch if equal)  
opcode: 101001 (**Format Two**)  
modes: direct  
CCs: EQ is tested  
traps: none  
notes: If EQ=1, replace the contents of PC by AOP.

**BranchNotEqual** (branch if not equal)  
opcode: 101010 (**Format Two**)  
modes: direct  
CCs: EQ is tested  
traps: none  
notes: If EQ=0, replace the contents of PC by AOP.

**BranchGreater** (branch if greater than)  
opcode: 101011 (**Format Two**)  
modes: direct  
CCs: GT is tested  
traps: none  
notes: If GT=1, replace the contents of PC by AOP.

**BranchLessEqual** (branch if less than or equal)  
opcode: 101100 (**Format Two**)  
modes: direct  
CCs: GT is tested  
traps: none  
notes: If GT=0, replace the contents of PC by AOP.

**BranchOverflow** (branch if overflow)  
opcode: 101101 (**Format Two**)  
modes: direct  
CCs: OV is tested  
traps: none  
notes: If OV=1, replace the contents of PC by AOP.

**LogicalAnd** (logical and)  
opcode: 101110 (**Format Two**)  
modes: direct  
CCs: none  
traps: none  
notes: Perform a bitwise logical 'and' of AC and OP, leaving the result in AC.

**LogicalOr** (logical or)  
opcode: 101111 (**Format Two**)  
modes: direct  
CCs: none  
traps: none  
notes: Perform a bitwise logical 'or' of AC and OP, leaving the result in AC.

**LogicalXor** (logical exclusive or)  
opcode: 110000 (Format Two)  
modes: direct  
CCs: none  
traps: none  
notes: Perform a bitwise logical ‘xor’ of AC and OP, leaving the result in AC.

**SetIndexReg** (store value into XR)  
opcode: 110001 (Format Two)  
modes: immediate  
CCs: none  
traps: none  
notes: Load OP into XR.

**IncIndexReg** (increment XR)  
opcode: 110010 (Format Two)  
modes: immediate  
CCs: none  
traps: none  
notes: Add OP to the contents of XR treating both as twos-complement 16-bit integers, and leave the result in XR. Any overflow is ignored.

**IncStackPoint** (increment SP)  
opcode: 110011 (Format Two)  
modes: immediate  
CCs: none  
traps: none  
notes: Add OP to the contents of SP treating both as twos-complement 16-bit integers, and leave the result in SP. Any overflow is ignored.

**Call** (procedure call)  
opcode: 110100 (Format Two)  
modes: direct  
CCs: none  
traps: none  
notes: Push the contents of PSW onto the stack. Set PC to the value of AOP. See Section 2.8.2 for more discussion of procedures.

**Trap** (trap)  
opcode: 110101 (Format Two)  
modes: immediate  
CCs: none  
traps: see Section 2.8.3  
notes: Suspend normal execution of the program, and pass control to the operating system. The value OP is taken as the trap type. Some trap types are predefined by the operating system. See Section 2.8.3 for a full description of trap processing.

**LoadAddress** (load address)  
opcode: 110110 (Format Two)  
modes: direct  
CCs: none  
traps: none  
notes: Load the value of AOP into AC[9..0] and clear AC[15..10]

**Return** (return)  
opcode: 1110000 (Format Three)  
modes: none  
CCs: none  
traps: none  
notes: Return from procedure or from exception service routine. First set PC to memory [SP] [9..0], and then decrement SP. See Section 2.8.2 for more discussion of procedures, and Section 2.8.3 for more discussion of traps.

**ClearOverflow** (clear overflow bit)  
opcode: 1110001 (Format Three)  
modes: none  
CCs: OV is cleared  
traps: none  
notes: Set OV := 0

**LoadPSW** (load PSW)  
opcode: 1110010 (Format Three)  
modes: none  
CCs: none  
traps: none  
notes: Load the contents of PSW into AC.

**StorePSW** (store PSW)  
opcode: 1110011 (Format Three)  
modes: none  
CCs: all are affected  
traps: none  
notes: Store AC[13..10] into PSW[13..10]. This affects the PSW bits EN, GT, EQ, and OV.  
The other parts of the PSW are unchanged.

**LogicalNot** (logical not)  
opcode: 1110100 (Format Three)  
modes: none  
CCs: none  
traps: none  
notes: Perform a bitwise logical 'not' of AC, leaving the result in AC.

**CompareIndexReg** (compare XR to AC)  
opcode: 1110101 (Format Three)  
modes: none  
CCs: EQ and GT are affected  
traps: none  
notes: Compare the contents of XR and AC, treating both as signed numbers. Indicate the result of the comparison by setting condition codes as follows:  
XR < AC: EQ := 0, GT := 0  
XR = AC: EQ := 1, GT := 0  
XR > AC: EQ := 0, GT := 1

**LoadIndexReg** (copy XR to AC)  
opcode: 1110110 (Format Three)  
modes: none  
CCs: none  
traps: none  
notes: Copy the contents of XR into AC.

**StoreIndexReg** (Copy AC to XR)  
opcode: 1110111 (Format Three)  
modes: none  
CCs: none  
traps: none  
notes: Copy the contents of AC into XR.

**LoadStackPoint** (copy SP to AC)  
opcode: 1111000 (Format Three)  
modes: none  
CCs: none  
traps: none  
notes: Copy the contents of SP into AC.

**StoreStackPoint** (copy AC to SP)  
opcode: 1111001 (Format Three)  
modes: none  
CCs: none  
traps: see notes  
notes: Copy the contents of AC into SP.

## B Detailed description of trap handling

The following description gives full details of how the operating system deals with traps.

Normal program execution is suspended, and control is passed to the operating system. (Note that the PC is now pointing at the instruction immediately following the instruction which caused the trap).

A trap number would have been generated by the hardware in the case of an exception; *otherwise* one is extracted from the trap instruction (the case of a service request). The entry in the PeANUt trap table corresponding to this trap number is then consulted.

- If the trap is a predefined one (see Section 2.8.3) and the *predefined action* is currently associated with this trap number then that action is taken.
- If the trap is not a predefined one and no **Establish Trap Routine** trap for this trap number has been executed, then the current trap is converted into a **Trapping Error** trap.
- If the most recent **Establish Trap Routine** trap for this trap number passed the value  $-2$  in place of a trap address then control is returned to the location indicated by the PC.
- If the most recent **Establish Trap Routine** trap for this trap number passed a trap routine address then PeANUt proceeds as if the instruction that caused the trap was a **Call** instruction to the trap service routine.
- If the most recent **Establish Trap Routine** trap for this trap number passed the value  $-1$  in place of a trap address and the trap number is not one of the predefined ones, then the current trap is converted into a **Trapping Error** trap.

## C Format of machine language input files

The PeANUt machine can be programmed directly at the machine language level using an input file format described in this section. Rather than writing an assembly language program that is first assembled into relocatable format then linked into the executable format, machine language input files can be translated directly into the executable file format from the machine language input format.

The machine language input file format consists of lines of word specifiers, each word specifier consisting of one or more bit specifiers. Each word specifier is translated into a 16 bit word (padding the least significant bits with zero if not enough bits are specified, or truncating if too many bits are specified) and written to the executable output file as one PeANUt memory cell. The 16 bit words are written into sequential memory cells, beginning with cell address zero and moving in the direction of larger addresses. Throughout translation, the translator maintains the address of the current cell being filled. The bit specifiers that make up a word specifier can specify bits in either hexadecimal, octal, binary, decimal or address (special octal). Hexadecimal bit specifiers produce 4 bits per digit, octal bit specifiers produce 3 bits per digit and binary specifiers produce one bit per digit. The address specifier is like octal, except that it produces exactly 10 bits (the size of the `opspec` field). When multiple hexadecimal, octal, binary or address specifiers occur in a word specifier, the bits they produce are concatenated in the order of most significant bit to least significant bit, and when 16 bits are accumulated, the word is written to the executable file into the cell of memory at the current address. A decimal bit specifier is translated to a 16 bit representation and appended to any preceding bits specified as part of the word. Note, that if the number of bits specified exceeds 16, the least significant bits will be lost from the decimal specifier as the word is truncated. The type (hexadecimal, octal, binary, decimal or address) of a bit

specifier is specified by preceding the digits of the number with a single letter, as given in the following table.

<i>type</i>	<i>base</i>	<i>bits produced</i>	<i>prefix</i>
hexadecimal	16	4 × number of digits	<b>h</b>
octal	8	3 × number of digits	<b>o</b>
binary	2	1 × number of digits	<b>b</b>
decimal	10	16	<b>d</b>
address	8	10	<b>a</b>

If no specifier type is given, the default is binary.

Three special directives are available in the machine language input file format. The **START** directive (followed by a bit specifier) specifies which cell of memory contains the first instruction of the PeANUt program. This directive *must* appear once and only once in a machine language input file, and should appear as the first thing in a machine language input file. The **AT** directive (followed by a bit specifier) writes out cells containing zero until the cell with address equal to the **AT** parameter is the current cell. The next word specifier after the **AT** will occupy the cell with address equal to the **AT** parameter. If the current address is not less than the parameter to the **AT** directive, then the **AT** directive has no effect. The **FILL** directive (followed by a bit specifier) writes a number of cells to the executable file (filled with zero bits) equal to the parameter of the directive.

The following simple example shows the equivalence between an assembly language program and a machine language input file. The following assembly language program is annotated with the C code that it represents.

```

                                ; void main(void) {
n:      block 1                  ;   int n;
pn:     block 1                  ;   int pn; // stores 2^n
Powers:                                ;
        load  #3                  ;   n = 3;
        store n                   ;
        load  #1                  ;   pn = 1;
        store pn                  ;
Pwhile: load  n                   ;   while (n != 0) {
        cmp   #0                  ;
        beq  Pendwhile           ;
        load  pn                  ;       pn = pn * 2;
        mul  #2                  ;
        store pn                 ;
        load  n                   ;       n = n - 1;
        sub  #1                  ;
        store n                  ;
        jmp  Pwhile              ; } // while
Pendwhile:                          ;
        load  pn                  ;   printf("%d", pn);
        add  #'0'                ;       // convert into ASCII
        trap #3                  ;
        load #'\\n'              ;   printf("\\n");
        trap #3                  ;
        trap #1                  ;
        end  Powers              ; } // main()

```

The following machine language program shows the use of the `START` and `FILL` directives, and also the use of the `a` bit specifier type to fill in the `opspec` field of `Format One` and `Two` instructions. Comments can be included in the machine language file, and consist of everything from a semicolon to the end of the line.

```

START a2                ;
FILL a2                 ; n:    block 1
                        ; pn:   block 1
                        ;
b000 b001 a3           ; Powers: load #3
b001 b010 a0           ;       store n
b000 b001 a1           ;       load #1
b001 b010 a1           ;       store pn
b001 b001 a0           ; Pwhile: load n
b000 b111 a0           ;       cmp #0
b101001 a20           ;       beq Pendwhile
b001 b001 a1           ;       load pn
b000 b110 a2           ;       mul #2
b001 b010 a1           ;       store pn
b001 b001 a0           ;       load n
b000 b100 a1           ;       sub #1
b001 b010 a0           ;       store n
b101000 a6            ;       jmp Pwhile
                        ; Pendwhile:
b001 b001 a1           ;       load pn
b000 b011 a60          ;       add #48
b110101 a3            ;       trap #3
b000 b001 a12         ;       load #10
b110101 a3            ;       trap #3
b110101 a1            ;       trap #1

```

## D Format of relocatable files

A relocatable file is the result of assembling one PeANUt assembly language file. Relocatable files contain one or more 16-bit (2 byte) records. The first record contains the value 1 in the most significant 6 bits and the value 0 in the least significant 10 bits. (This distinguishes the file from an image file; see Appendix E). The remaining records comprise zero or more “items”, each of which occupies one or more records. There are seven types of item, distinguished by the most significant 6 bits of the first record. In describing the items, we will use the following notation.

V = contents of bits 0-9  
T = contents of bits 10-15  
C1 = contents of bits 0-7  
C2 = contents of bits 8-15

- Zero block item (size = 1 record)

This specifies a block of from 0 to 1023 cells of memory initialised to zero.

first record: T = 0  
V = the number of cells

In response to this item, the linker allocates V cells of memory and initialises their contents to zero.

- Relocatable data item (size = 2 records)

This specifies one cell of memory whose least significant 10 bits are to contain an address specified relative to the start of this module.

first record: T = 1  
V = the offset from the start of this module (call this K)  
second record: T = data (call this D)  
V = ignored (but should be zero)

In response to this item, the linker allocates one cell of memory. In the most significant 6 bits it places D. In the least significant 10 bits it places K plus the base address of this module (ignoring overflow).

- External data item (size = 2 records)  
This specifies one cell of memory whose least significant 10 bits are to contain the value of an external symbol plus an offset.

first record:    T = 2  
                  V = the index of an external symbol (call this K)  
second record:  T = data (call this D)  
                  V = data (call this M)

In response to this item, the linker allocates one cell of memory. In the most significant 6 bits it places D. In the least significant 10 bits it places the value of external symbol K plus the number M, ignoring overflow. External symbols are numbered 0,1,2,... in the order they appear in external symbol records in this module.

- Constant data item (size = 2 records)  
This specifies one cell of memory containing a constant.

first record:    T = 3  
                  V = ignored (but should be zero)  
second record:  data

In response to this item, the linker allocates one cell of memory and initialises it to the contents of the second record of this item.

- External symbol item (size = 2 or more records)  
This gives the name of an external symbol.

first record:    T = 4  
                  V = the number of characters in the name  
second record:  C1 = first character of symbol name  
                  C2 = second character of symbol name  
third record:   C1 = third character of symbol name  
                  ... and so on

Only non-blank characters are stored. If there are an odd number of characters, the C2 field of the last record is ignored (but should be zero).

- Global relocatable symbol item (size = 3 or more records)  
This gives the name and value of a relocatable global symbol.

first record:    T = 5  
                  V = the number of characters in the name  
second record:  T = ignored (but should be zero)  
                  V = the offset from the start of this module (call this K)  
third record:   C1 = first character of symbol name  
                  C2 = second character of symbol name  
fourth record:  C1 = third character of symbol name  
                  ... and so on

Only non-blank characters are stored. If there are an odd number of characters, the C2 field of the last record is ignored (but should be zero). This item tells the linker about a symbol defined and declared global in this module. The value of the symbol is K plus the base address allocated by the linker to this module (ignoring overflow).

- Starting address item (size = 1 record)

This specifies the program starting address relative to the start of this module.

first record:    T = 6  
                  V = the offset from the start of this module (call this K)

This item tells the linker than the first instruction to execute in this program is located at address K plus the base address of this module.

## E Format of image files

Image (executable) files are the result of linking (joining) one or more relocatable files. An image file contains a complete program and the information needed to execute it. An image file consists of from 1 to 1025 records of 16 bits each. The first record contains the program

transfer (starting) address in its least significant 10 bits and zero in its most significant 6 bits. The remaining records contain the initial memory contents of cells 0 ... `program_size-1`. If there are less than 1024 of these records, the remaining cells of memory are assumed to be initially zero.

## F Utility programs

A graphical development environment for PeANUt programs running under X windows exists as part of the computing environment available at the ANU.

A set of utility programs are provided that allow you to assemble, link and execute PeANUt programs on Sun Unix computers from the UNIX shell command line. A description of these utilities follow.

### `assemble sourcefile`

This is the PeANUt assembler. It reads an PeANUt program and writes a relocatable file. The default filename extension of the input file name is `.ass`. The relocatable file has the same name as the input file except that it has filename extension `.rel`. A listing is also produced, in a file of the same name but with extension `.lst`.

### `join relfile1 ... relfilen`

This is the PeANUt linker. It reads one or more relocatable files and writes one image file. See Section 4.3 for a description of the linking process. The default type of the input file names is `.rel`. The image file has the same name as the first relocatable file except that it has type `.img`.

### `execute -trace -vm imagefile`

Execute the PeANUt program in the given image file. The default extension for the file name is `.img`. Use the `-trace` option to get an instruction by instruction trace of the execution. Use the `-vm` option *only* to execute the PeANUt machine in Virtual Memory mode (see Section 3).

### `decode filename`

List the contents of a relocatable file or image file in a format readable by humans.

Suppose you have an PeANUt assembler program in the files `part1.ass` and `part2.ass`. To assemble, link and run it, execute the commands on the left of the following table.

<code>assemble part1</code>	makes <code>part1.rel</code> from <code>part1.ass</code>
<code>assemble part2</code>	makes <code>part2.rel</code> from <code>part2.ass</code>
<code>join part1 part2</code>	makes <code>part1.img</code> from <code>part1.rel</code> and <code>part2.rel</code>
<code>execute part1</code>	executes the program in <code>part1.img</code>

## G Example program

The following is a simple program that reads its input line by line into a string, and writes the same string as output. This program contains examples of how most of the major data declarations and control structures of C are translated into PeANUt assembly language, including how to call assembly language procedures.

```
; echoline
;
; This program reads from the input line by line, and stores each line in the
; string variable Line (including the '\n' character, if any). Line is echoed
; to standard output by calling the WriteString procedure. The output of the
; program is thus identical to the input.
;
; The program assumes that no line in the input has more than LineMax chars.
;
; Peter Strazdins 19/8/1991
; Peter Christen 16/12/2002 (converted comments into C)

external Write          ; #include <stdio.h>
external WriteString   ;
    EOF = -1           ; #define EOF -1 /* EOF value */
    LineMax = 120      ; #define LineMax 120 /* maximum line size */
; void main(void) {
line:  block  LineMax   ; char line[LineMax+1];
      block  1         ; // has LineMax+1 chars
nch:   block  1         ; int nch; // value of char read
i:     block  1         ; int i;
echoline:
repeat:
    trap #2           ; do {
      store nch       ; nch = getchar();
    load #0           ; i = 0;
    store i           ;
while2: load nch      ; while ((nch != '\n')
    cmp #' \n'       ;
    beq endwh2       ;
    load nch          ; && (nch != EOF) {
    cmp #EOF          ;
    beq endwh2       ;
    load i            ; line[i] = nch;
    storexr          ;
    load nch          ;
    store *Line       ;
    trap #2          ; nch = getchar();
    store nch        ;
    load i            ; i = i + 1;
    add #1           ;
    store i          ;
    jmp while2       ; } // while
endwh2:
    load i            ; line[i] = '\0';
    storexr          ;
    load #0           ;
    store *Line       ;
    loada Line        ; printf(line);
    incsp #1         ;
    store !0         ;
    call WriteString ;
    incsp #-1        ;
    load nch          ; if (nch != EOF) {
    cmp #EOF          ;
    beq endif        ;
    load #' \n'      ; putchar('\n');
    trap #3          ;
endif:
    load nch          ; } // if
    cmp #EOF          ; } while (nch != EOF);
    bne repeat       ;
    trap #1          ; } // main()
end echoline        ;
```