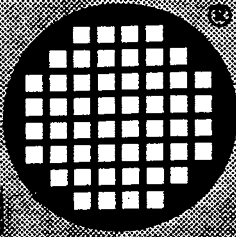


inmos®

Technical Note 8

Prepared by the Microcomputer Applications Group, Colorado Springs

The Implementation of OCCAM on the IMS T414

Introduction

This note provides information on the internal architecture of the processor of the IMS T414 transputer by, firstly, describing the overall organisation of the processor and then demonstrating its use to implement the various constructs of occam. (occam is a trademark of the INMOS Group of Companies). The demonstration is organized thus: various interesting fragments of occam program are given, and for each fragment the instruction sequence generated by the compiler is given, along with the size and execution time of each instruction in the sequence. Other high level languages would give rise to similar or identical instruction sequences as the equivalent occam fragments (at least for the sequential fragments).

The transputer has been designed so that programs can be compiled simply and straightforwardly, and so that the use of high level languages results in efficient use of silicon capability.

As a consequence, its execution mechanisms differ in several respects from those of conventional microprocessors. It is based on the use of very fast single byte instructions, which results in faster and more compact code than is possible for a conventional instruction set. The correct and optimal sequence is easy for a compiler to compile, but impractical for hand coding.

The execution architecture contains an evaluation stack, which removes the need for instructions to specify registers explicitly. Consequently, most of the executed operations (typically 80%) are encoded in a single byte. Many of these, such as 'load constant', or 'add', require just one processor cycle.

In general, a program needs much less store to hold it than an equivalent program in a conventional microprocessor. Since a program requires less store to represent it, less of the memory bandwidth is taken up with fetching instructions. As memory is word accessed, the processor will receive four instructions for every fetch.

Short instructions also improve the effectiveness of the instruction fetch mechanism, which in turn improves processor performance. The processor uses otherwise spare memory cycles to fetch instructions. There are two words of instruction fetch buffer, with the result that the processor rarely has to wait for an instruction fetch. Since the buffer is short, there is little time penalty when a jump instruction causes the buffer contents to be filled.

The overall effect is thus that both compactness and speed have been achieved, together with economical use of silicon.

The lowest level of programming transputers is to use occam (occam is equivalent in effectiveness to a conventional microprocessor's assembler). It is far easier to understand the occam corresponding to a sequences of instructions, than it is to understand the individual instructions themselves. This note therefore describes the instruction set, and the use of occam as its programming language, by describing the main usage of the various registers in the machine, and then by giving typical instruction sequences for simple occam constructs.

Memory Organisation

The memory address space comprises a signed linear address space of2**32bytes. The instruction architecture does not differentiate between on-chip and off-chip memory. This allows the application designer to have complete control over the placement of code and data to take advantage of the performance benefits of on-chip memory.

The internal organization of the processor is based on the word length. All internal registers and data paths on T414 are 32 bits wide.

A byte in memory is identified by a pointer, which is a single word of data divided into two parts

- a word address and a byte selector. On T414 the byte selector occupies the least significant two bits of the word; the word address occupies the most significant bits. Pointer values start from the most negative integer and continue, through zero, to the most positive integer. This enables the standard comparison functions to be used on pointer values in the same way that they are used on numerical values.

For efficiency, the processor accesses memory a word at a time on word boundaries, and, where necessary, performs appropriate byte manipulation operations.

The addressing instructions provide access to items in data structures, using short sequences of single byte instructions, allowing the representation of data structure access to be independent of the word length of the processor.

Registers for sequential programming

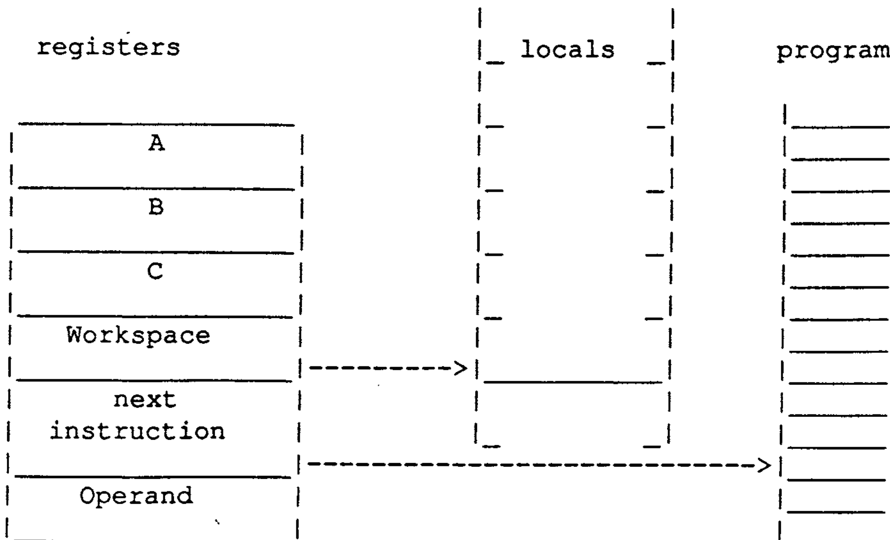
The design of the processor exploits the availability of fast on-chip memory by having only a small number of registers. The small number of registers, together with the simplicity of the instructions, enables the processor to have relatively simple and therefore fast data paths and control logic.

Sequential programs use the following registers (Figure 1):

The **workspace pointer** which points to an area of store where local variables are kept.
The **instruction pointer** which points to the next instruction to be executed.
The **operand register** which is used in the formation of instruction operands.

The A, B and C registers which form an evaluation stack. The evaluation stack is used for expression evaluation, to hold the operands of scheduling and communication instructions, and to hold the first three parameters ofprocedure calls.

Figure 1 Use of registers for sequential programming



Support for concurrency

The processor provides efficient support for the occam model of concurrency and communication. It has a scheduler which enables any number of concurrent processes to be executed together, sharing the processor time.

At any time, a concurrent process may be

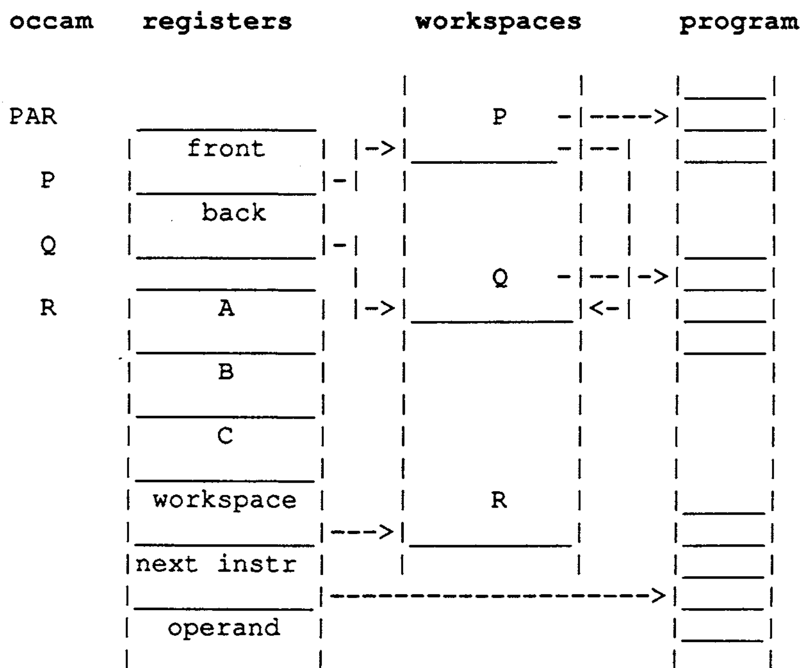
active
- being executed
- on a list awaiting execution

inactive
- ready to input
- ready to output
- waiting until a specified time

The active processes waiting to be executed are held on a list. This is a linked list of workspaces of processes, implemented using two registers, one of which points to the first process on the list, the other to the last.

In Figure 2, **R** is executing, and **P** and **Q** are active, awaiting execution.

Figure 2 Concurrent processes



Whenever a process is unable to proceed, its instruction pointer is saved in its workspace and the next process is taken from the list. Actual process switch times are very small as little state needs to be saved. The contents of the evaluation stack will not be needed when the process is resumed, and so the evaluation stack is not saved. Two active lists are maintained, one for each priority.

Communications

A channel provides a communication path between two processes. Channels between processes executing on the same transputer are implemented by single words in memory (internal channels); channels between processes executing on different transputers are implemented by point-to-point links (external channels).

The processor uses the address of a channel to determine whether the channel is internal or external. This means that the same instruction sequence can be used for both internal and external channels, allowing a process to be written and compiled without knowledge of where its channels are connected. In particular, either an internal or an external channel can be used as the actual parameter for a channel parameter of a named process.

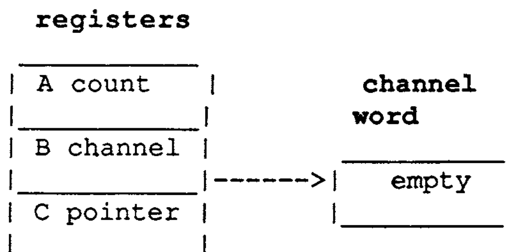
As in the occam model, communication takes place when both the inputting and outputting processes are ready. Consequently, the process which first becomes ready must wait until the second one is also ready.

A process prepares for input or output by loading the evaluation stack with a pointer to a buffer, the identity of the channel, and the count of the number of bytes to be transferred.

An internal channel is allocated a word in memory, and instructions compiled to initialize it to empty. In figure 3, a process P is about to execute an input or an output message instruction.

Figure 3 Input or output on empty channel

P executing



When a message is passed using an internal channel, the identity of the first process to become ready is stored in the channel word, and the pointer stored (with the instruction pointer etc) in the workspace (Figure 4). The processor starts to execute the next process from the scheduling list.

Figure 4 Ready channel and process

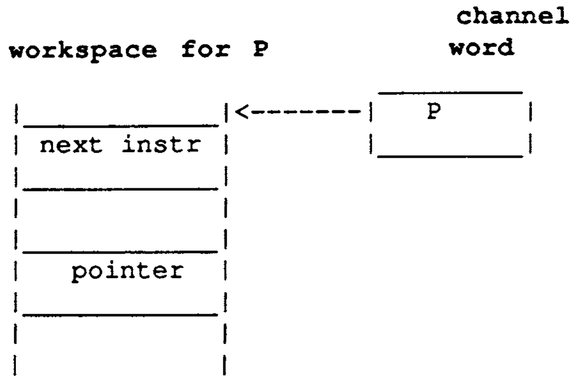
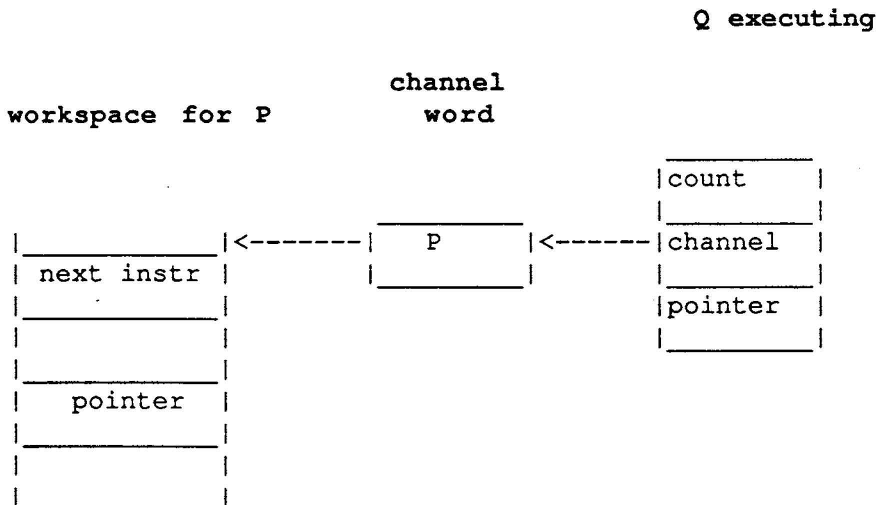


Figure 5 shows the second process to use the channel:

Figure 5 Input or output on ready channel



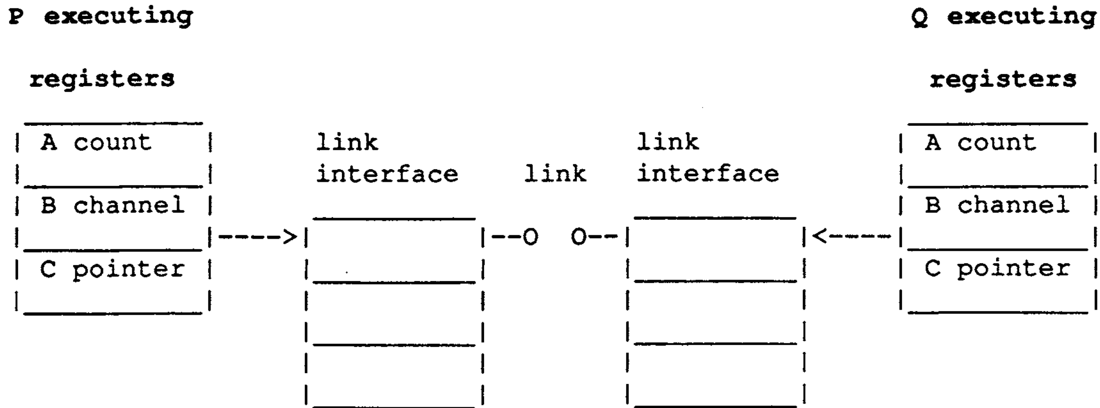
The message is copied, the waiting process is added to the active process list, and the channel reset to the empty state. It does not matter whether the inputting or the outputting process becomes ready first.

When a message is passed via an external channel the processor delegates to an autonomous link interface the job of transferring the message and deschedules the process. The link interface transfers the message using direct memory access. When the message has been transferred the link interface causes the processor to reschedule the waiting process. This allows the processor to continue the execution of other processes whilst the external message transfer is taking place.

The following figures (6, 7) show the sequence of operations when two processes, executing on

separate transputers, communicate using a link connecting the two transputers. Each process prepares for the transfer as already described.

Figure 6 Input and output on link interface

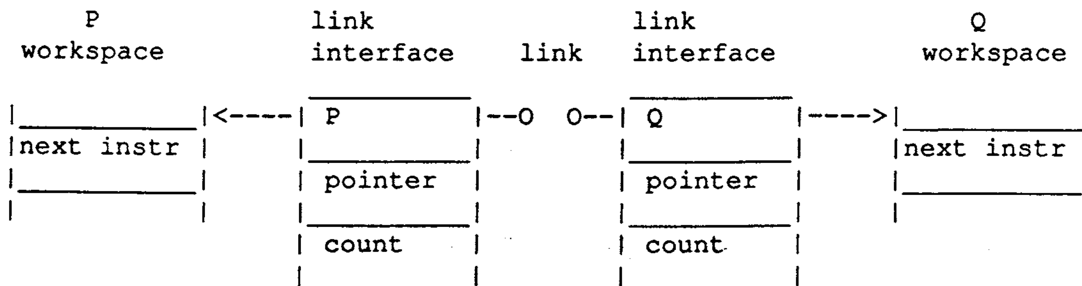


Each link interface uses three registers to hold the following information

a pointer to the workspace of the process
a pointer to the message
a count of bytes to be transferred

When the **input message** or **output message** instruction is executed, these registers are initialized, and the instruction pointer is stored in the process workspace. The processor starts to execute the next process on the scheduling list.

Figure 7 Use of link interface registers



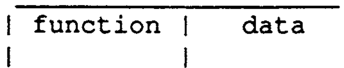
When both processors have initialized both link interfaces, the message is copied, and each link interface adds the respective process to the end of the local active list.

Instruction format

For simplicity in reading the example code sequences, the instruction's names are shown unabbreviated.

All instructions have the same format (Figure 8). Each is one byte long, and is divided into two 4 bit parts. The four most significant bits of the byte are the function code, and the four least significant bits are the data value.

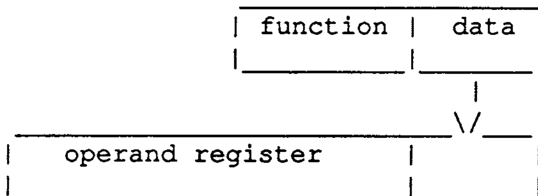
Figure 8 Instruction format



This representation provides for sixteen functions, each with a data value ranging from 0 to 15. Thirteen of these values are used to encode the most frequently occurring functions.

Two more of the function codes, **prefix** and **negative prefix,** are used to allow the operand of any instruction to be extended in length. All instructions start by loading the four data bits into the least significant four bits of the operand register, which is then used as the instruction's operand (Figure 9). All instructions, except **prefix** and **negative prefix**, end by clearing the operand register, ready for the next instruction.

Figure 9 Use of operand register



The **prefix** instruction loads its four data bits into the operand register, and then shifts the operand register up four places. The **negative prefix** instruction is similar, except that it complements the operand register before shifting it up. Consequently operands can be extended to any length up to the length of the operand register by a sequence of prefixing instructions. In particular, operands in the range -256 to 255 can be represented using one prefixing instruction.

The following example shows the instruction sequence for loading the hexadecimal constant #754 into the A register, and gives the contents of the O register and the A register after executing each instruction

```
                        O register    A register
prefix  #7                      #7             ?
prefix  #5                     #75             ?
load constant #4                 0          #754
```

The use of prefixing instructions has certain beneficial consequences. Firstly, they are decoded and executed in the same way as every other instruction, which simplifies and speeds instruction decoding. Secondly, they simplify language compilation, by providing a completely uniform way of allowing any instruction to take an operand of any size. Thirdly, they allow operands to be represented in a form which is independent of the processor wordlength.

The processor does not allow interrupts between a prefixing instruction and the following instruction. This removes the need to save the operand register on an interrupt.

The remaining function code, **operate**, causes its operand to be interpreted as an operation on the values held in the evaluation stack. For example, the **plus** operation adds the values of the A and B registers. The result is left in the A register, and C is copied into the B register.

The **operate** instruction allows up to 16 such operations to be encoded in a single byte instruction. However, the prefixing instructions can be used to extend the operand of an operate instruction just like any other. This allows the number of operations in the machine to be extended indefinitely.

The encoding of the indirect functions is chosen so that the most frequently occurring operations are represented without the use of a prefixing instruction. These include arithmetic, logical and comparison operations, together with the most frequently used control functions and register manipulation functions.

Prefixing instructions are not given explicitly in many of the examples which follow. However, the extra byte(s) and cycle(s) are included in the costs which annotate the instruction sequences. A * is used to indicate where the costs of a prefixing instruction has been included with those of a direct function.

The examples give the execution time for each instruction, in terms of processor cycles. The costs of fetching instructions and accessing data in external memory are not included.

Sequential constructs

Local variables and constants

Description

Operations on locals and small constants are fast and compact.

The most common operations in any program are the loading of small literal values, and the loading or storing of one of a small number of variables. The **load constant** instruction enables values between 0 and 15 to be loaded onto the evaluation stack with a single byte instruction. The **load local** and **store local** instructions transfer values between the evaluation stack and locations in memory relative to the workspace pointer. The first 16 locations can be accessed using a single byte instruction.

Examples

In the following examples, x and y are assumed to be local variables within the first sixteen words of workspace.

occam	instruction sequence	bytes	cycles
x := 0	load constant 0	1	1
	store local x	1	1
x := #24	prefix 2	1	1
	load constant 4	1	1
	store local x	1	1
x := y	load local y	1	2
	store local x	1	1

Non-local variables and data structures

Description

Compact sequences of instructions are used to provide simple implementations of the static links or displays used in the implementation of block structured programming languages, and to provide efficient access to data structures.

This eliminates the need for complicated and difficult-to-use addressing modes.

The **load non local** and **store non local** instructions access locations in memory relative to the A register.

Examples

In this example, **z** is assumed to have been declared externally to the **PROC** which contains this assignment statement. The compiler allocates a local workspace location, here named **staticlink**, to hold the address of the workspace containing the variable z.

occam	instruction sequence	bytes	cycles
z := -1	negative prefix 0	1	1
	load constant #F	1	1
	load local staticlink	1	2
	store non local z	1	2

Expression evaluation

Description

Loading a value onto the evaluation stack pushes B into C, and A into B, before loading Storing a value from A, pops B into A and C into B.

The A, B and C registers are the sources and destinations for arithmetic and logical operations. For example, the **add** instruction adds the A and B registers, places the result in the A register, and copies C into B.

If there is insufficient room to evaluate an expression on the stack, then the compiler introduces the necessary temporary variables in the local workspace. However, expressions of such complexity are, in practice, rarely encountered. Three registers provide a good balance between code compactness and implementation complexity.

Examples

In the following examples, the variables v, w, x, y and z are assumed to be in the first sixteen words of workspace.

occam	instruction sequence	bytes	cycles
y + z	load local y	1	2
	load local z	1	2
	add	1	1
(v + w) * (y + z)			
	load local v	1	2
	load local w	1	2
	add	1	1
	load local y	1	2
	load local z	1	2
	add	1	1
	multiply	2	39

Arithmetic overflow

Description

The processor uses the sticky-bit method of error detection. The processor contains an error flag, which is set on arithmetic overflow. The instruction **test error** sets the A register to **TRUE** (least significant bit set to 1, all other bits zero) if the error flag is set, otherwise it sets the A register to **FALSE** (all bits zero). In either case, the error flag is reset.

The instruction **stop on error** stops the current process if the error flag is set (NB the process does not terminate).

The **Error** pin is connected directly to the error flag. If this pin is connected to the **StopProc** pin, then the effect of any error is to stop the processor.

The error flag is preserved as part of the context of a priority 1 process when it is is interrupted by a priority 0 process.

Example

occam	instruction sequence	bytes	cycles
x := (v + w) * (y + z)			
	test error	2	3
	load local v	1	2
	load local w	1	2
	add	1	1
	load local y	1	2
	load local z	1	2
	add	1	1
	multiply	2	39
	stop on error	2	3
	store local x	1	1

Single Length Arithmetic

Description

Single length signed and single length modulo arithmetic is directly supported. In addition, a quick unchecked multiply is provided, in which the time taken is proportional to the logarithm of the second operand.

Examples

Signed arithmetic (sets error on arithmetic overflow):

occam	instruction sequence	bytes	cycles
x + 2	load local x	1	2
	add constant 2	1	1
x + y	load local x	1	2
	load local y	1	2
	add	1	1
x - y	load local x	1	2
	load local y	1	2
	subtract	1	1
x * y	load local x	1	2
	load local y	1	2
	multiply	2	39
x / y	load local x	1	2
	load local y	1	2
	divide	2	42 (worst case)
x \ y	load local x	1	2
	load local y	1	2
	remainder	2	41 (worst case)

Modulo arithmetic (arithmetic overflow not checked):

occam	instruction sequence	bytes	cycles
x (+) y	load local x	1	2
	load local y	1	2
	sum	2	2
x (-) y	load local x	1	2
	load local y	1	2
	difference	1	1
x (*) y	load local x	1	2
	load local y	1	2
	product	2	3-35

Array access

Description

The subscript is evaluated, and left in the A register. It is then checked to be in range (alternatively, the compiler may be able to perform this check at compile time). The error flag (see arithmetic overflow) is set if the subscript is out of range. The transputer provides instructions for subscript checking both from zero and from one.

The address of a data structure held in the local workspace is loaded using the **load local pointer** instruction. The **load non local pointer** instruction is used to load the address of a data structure relative to the A register.

The **byte subscript** and **word subscript** instructions are used to calculate the pointer to an item in an array of bytes, or an array of words respectively. Both interpret the contents of the A register as the address of the beginning of a data structure. These instructions are used to make the code independent of the word length of the machine.

Example

In these examples, **v** is declared as **[5]INT, w** is declared as **[3][5]BYTE, i** is a local variables.

occam	instruction sequence	bytes	cycles
x := v[i]	test error	2	4
	load local i	1	2
	load constant 5	1	1
	check subscript from 0	2	3
	stop on error	2	2
	load local pointer v	*2	2
	word subscript	1	2
	load non local 0	1	2
	store local x	1	1
x := w[2][i]	test error	2	4
	load local i	1	2
	load constant 5	1	1
	check subscript from 0	2	3
	load constant 3	1	1
	product	1	6
	add constant 2	1	1
	stop on error	2	2
	load local pointer w	*2	2
	byte subscript	1	1
	load non local 0	1	2
	store local x	1	1

* including a prefix instruction

Bit manipulation operations

Description

Bit manipulation operations are provided by secondary instructions operating on the evaluation stack.

Examples

occam	instruction sequence	bytes	cycles
x /\ #F	load local x	1	2
	load constant #F	1	1
	and	2	2
x \/ y	load local x	1	2
	load local y	1	2
	or	2	2
x >< y	load local x	1	2
	load local y	1	2
	xor	2	2
~ x	load local x	1	2
	not	2	2
x << y	load local x	1	2
	load local y	1	2
	shift left	2	3+y
x >> 3	load local x	1	2
	load constant 3	1	1
	shift right	2	6

Boolean expressions

Description

Boolean expressions (Boolean values combined with AND, OR and NOT) employ short circuit evaluation techniques. If the contents of the A register is **FALSE**, the **conditional jump** instruction adds its operand to the instruction pointer, without altering the evaluation stack. Otherwise it pops the evaluation stack (ready for the next instruction(s) to replace the value of the first operand by the value of the second operand of the AND or OR).

TRUE and **FALSE** are represented by the values 1 and 0 respectively. Thus they can both be loaded in a single instruction, and a Boolean value negated by comparison with zero, using the **equals constant** instruction with zero as operand. This loads the A register with **TRUE** if A is initially 0 **(FALSE)**, **FALSE** otherwise.

Examples

occam	instruction sequence	bytes	cycles	(jump taken)
NOT x	load local x	1	2	
	equals constant 0	1	2	
x AND y	load local x	1	2	2
	conditional jump M	1	2	4
	load local y	1	2	-
	M:			
x OR y	load local x	1	2	2
	equals constant 0	1	2	2
	conditional jump M	1	2	4
	load local y	1	2	-
	equals constant 0	1	2	-
	M:equals constant 0	1	2	2
NOT (x OR y)	load local x	1	2	2
	equals constant 0	1	2	2
	conditional jump M	1	2	4
	load local y	1	2	-
	equals constant 0	1	2	2
	M:			

Comparisons

Description

A comparison normally uses the modulo arithmetic **difference** instruction, followed by a comparison to zero. The **equals constant** instruction provides comparison with a constant value. It loads the A register with **TRUE** if A is initially the value in the operand register, **FALSE** otherwise. Similarly, the **greater than** instruction loads the A register with **TRUE** if B > A, **FALSE** otherwise.

Examples

occam	instruction sequence	bytes	cycles
x = 5	load local x	1	2
	equals constant 5	1	2
x = y	load local x	1	2
	load local y	1	2
	difference	1	1
	equals constant 0	1	2
x <> 5	load local x	1	2
	equals constant 5	1	2
	equals constant 0	1	2
x <> y	load local x	1	2
	load local y	1	2
	difference	1	1
	equals constant 0	1	2
	equals constant 0	1	2
x > y	load local x	1	2
	load local y	1	2
	greater than	1	2
x < y	load local y	1	2
	load local x	1	2
	greater than	1	2
x >= y	load local y	1	2
	load local x	1	2
	greater than	1	2
	equals constant 0	1	2

Conditional behavior

Description

Conditional behavior is provided by using both **conditional jump** and the unconditional **jump** instructions. Both transfer control relative to the instruction pointer, providing position independence and compact encoding. A timeslice may occur on an unconditional **jump** instruction (this allows a process to be saved when a timeslice occurs without having to save the evaluation stack).

Example

occam	instruction sequence	bytes	cycles		
			x <	=	> 0
IF	load local x	1	1	1	1
x = 0	equals constant 0	1	2	2	2
y := 1	conditional jump M1	1	4	2	4
x > 0	load constant 1	1	-	1	-
y := 2	store local y	1	-	2	-
	jump M0	1	-	3	-
	M1:load local x	1	1	-	1
	load constant 0	1	1	-	1
	greater than	1	2	-	2
	conditional jump M2	1	4	-	2
	load constant 2	1	-	-	1
	store local y	1	-	-	2
	jump M0	1	-	-	3
	M2:stop process	2	12	-	-
	M0:				

Named processes (procedure calling)

Description

The use of PROCs saves on code space requirements. Each PROC also introduces a new workspace, so local workspaces remain small, which, in turn, maximizes the efficiency of local variable access.

The first two parameters and the static link are passed to a named process on the evaluation stack, the remaining parameters are passed in workspace locations, allocated, if necessary, using the **adjust workspace** instruction.

To prepare for a procedure call, instructions are compiled to evaluate the third and subsequent parameters, storing them in workspace locations starting from 0, and then to load onto the evaluation stack the static linkand the first two parameters.

The **call** instruction saves the contents of the evaluation stack and the instruction register in the locations immediately below the current workspace,copies the instruction pointer to the A register, decrements the workspace pointer by four words, and jumps. The called procedure allocates space for local variables by using the **adjust workspace** instruction, and deallocates the space before exit. VAL parameters are evaluated, and the value passed to the called procedure. Other parameters are passed by pointer.

The **return** instruction assumes that the workspace pointer has the same value as immediately after the corresponding **call**. It restores the instruction pointer from the stored value, and adjusts the workspace pointer by four words.

Example

```
     occam     instruction sequence      bytes    cycles

PROC g(INT g1, VAL INT g2, g3, g4) =
  . :
PROC f(VAL INT f1, f2, INT f3) =
  VAR x:
  . . .
  g(x, f1, f2, f3)
  . . . :
               f:adjust workspace -1      *2        3
                  . . .
                 adjust workspace -2      *2        3
                 load local f3             1        2
                 load non local 0          1        2
                 store local 1             1        1
                 load local f2             1        2
                 store local 0             1        1
                 load local f1             1        2
                 load local pointer x      1        2
                 load local static link    1        2
                 call g                   *2        8
                 adjust workspace 2        1        2
                  . . .
                 adjust workspace 1        1        2
                 return                    2        7
```

* including a **prefix** instruction

Replicated SEQ

Description

The **loop end** instruction enables efficient implementation of all the occam replicated constructs.

The index and count are evaluated, and stored in consecutive locations of local workspace. The **loop end** instruction uses the B register as the address of the index, which is incremented. The count, assumed in the next location, is decremented. If the result is greater than zero, the value in the A register is subtracted from the instruction pointer. A timeslice may occur on a **loop end** instruction.

Example

In these examples, **index** is the first of two locations in local workspace, allocated by the compiler for loop control.

```
 occam     instruction sequence     bytes    cycles

SEQ i = 0 FOR 10
  P
                load constant 0          1       1
                store local index        1       1
                load constant 10         1       1
                store local index+1      1       1
              L:P
                load local pointer index 1       1
                load constant M-L        1       1
                loop end                 2      12   6 (final time)
              M:

SEQ i = 0 FOR n
  P
                load constant 0          1       1
                store local index        1       1
                load local n             1       2
                store local index+1      1       1
                load local index+1       1       2
                load constant 0          1       1
                greater than             1       2
                conditional jump M       1       2   4 (on jump)
              L:P
                load local pointer index 1       1
                load constant M-L        1       1
                loop end                 2      12   6 (final time)
              M:
```

WHILE

Description

The loop commences with the evaluation of the Boolean expression, and a conditional jump. The end of the loop contains an unconditional jump to the top of the loop to re-evaluate the expression. A timeslice may occur on the **jump** instruction.

Example

In this example, the body of the loop is assumed to occupy between 8 and 247 bytes of code.

```
occam          instruction sequence      bytes    cycles

 WHILE x > 0
   P
              L:load local x               1        2
                load constant 0            1        1
                greater than               1        2
                conditional jump M        *2        3    5 (final time)
                P
                jump L                    *2        4
              M:
```

* including a **prefix** instruction

Concurrency and communication
PAR

Description

The **start process** and **end process** instructions make process creation and termination fast and compact.

When a parallel construct is executed, extra workspaces are created for the extra processes. A termination block is created in the current workspace containing a count of the number of components of the parallel construct and the instruction pointer of the instruction which follows the parallel construct.

The first of the concurrent processes uses the existing workspace. Each of the remaining concurrent processes is created by loading the address of its first instruction and the address of its workspace, and executing the **start process** instruction. This initializes the new workspace and adds it to the end of the scheduling list, enabling the new concurrent process to be executed together with the ones already being executed.

The correct termination of a parallel construct is assured by use of the **end process instruction**. This examines the counter of the components of the parallel construct which are still executing. If the counter is greater than one, then it is decremented, and the next process is taken off the process queue. If the counter is equal to one, then this is the last component process to terminate, and, after adjusting the workspace pointer, control is transferred to the next instruction following the parallel construct.

Example

In this example, **P0** and **P1** are unspecified, but have workspace requirements of **W0** and **W1**. In the timings, it is assumed that **P0** terminates first (if **P1** terminates first, then the timings for the two end process instructions are exchanged).

occam	instruction sequence	bytes	cycles
PAR	adjust workspace -2	*2	3
P0	load constant M-L	*2	2
P1	load pointer to instruction	2	3
	L:store local 0	1	1
	load constant 2	1	1
	store local 1	1	1
	load constant L0-L1	*2	2
	load local pointer -(W0+W1)+2	*2	2
	start process	1	12
	L0:adjust workspace -W0	*2	3
	P0		
	load local pointer W0	1	1
	end process	1	13
	L1:P1		
	load local pointer W0+W1-2	*2	2
	end process	1	8
	M:		

* including a **prefix** instruction

Replicated PAR

Description

Space is allocated for the workspaces of the component processes in a manner similar to an unreplicated PAR. In addition, each workspace contains a location for its value of the replicator index, and a pointer to the termination block. These permit one copy of the code to be used for all components of the replicated parallel construct.

The workspaces for the component processes are established using a loop. A dummy local variable is used in the loop as a pointer to the workspace of the process being established.

Example

In this example, the component (replicated) process **P** is unspecified, but has a workspace requirement of **W**. The offsets for locations in workspace for the loop index, count, temporary pointer to the workspace being initialized, and in each component processes workspace for the replicator index value and termination block pointer, are indicated by corresponding names.

```
 occam        instruction sequence       bytes    cycles
                                                (4 processes)
PAR i = 0 FOR 4
  P
             adjust workspace -5          *2        3
             load constant 0               1        1
             store local index             1        1
             load constant 4               1        1
             store local count             1        1
             load local pointer -(4*W)+2  *2        2
             store local wtemp             1        1
             load constant M-L1           *2        2
             load pointer to instruction   2        3
          L1:store local 0                 1        1
             load local count              1        2
             store local 1                 1        1
          L2:load local pointer 0          1        4
             load local wtemp              1        8
             store non local tblockptr     1        8
             load local index              1        8
             load local wtemp              1        8
             store non local replindex     1        8
             load local count              1        8
             add constant -1              *2        8
             conditional jump L3           1       10
             load constant L5-L4           1        3
             load local wtemp              1        6
             start process                 1       36
          L4:jump L6                      *2       12
          L3:adjust workspace -W+2         2        3
```

```
L5:P                                               4*P
   load local tblockptr              1          8
   load non local pointer 0          1          4
   end process                       1         47
L6:load local wtemp                  1          8
   load non local pointer W          1          4
   store local wtemp                 1          4
   load local pointer index          1          4
   load constant L2-M               *2          8
   loop end                          2         36
 M:
```

* including a **prefix** instruction

Input and output

Description

Input or output is performed by loading the pointer to the buffer, the address of the channel, and the count of bytes to be transferred, and calling the **input** or **output** instruction. Optimised communication (using less instructions to prepare for output) is provided by the **output word** and **output byte** instructions.

Examples

In the following examples, v is assumed to be an array of bytes. The times given include the cycles used for copying the message if the channel is ready, otherwise the number of cycles required are those annotated for when the channel is not ready.

occam	instruction sequence	bytes	cycles
c ? v[i FOR 12]			
	load local i	1	2
	load local pointer v	*2	2
	byte subscript	1	1
	load local pointer c	1	1
	load constant 12	1	1
	input message	1	25 17 (not ready)
c ! 0	load constant 0	1	1
	load local pointer c	1	1
	output word	1	27 20 (not ready)
c ! BYTE(0)	load constant 0	1	1
	load local pointer c	1	1
	output byte	1	27 20 (not ready)
c ? x	load local pointer x	1	1
	load local pointer c	1	1
	load constant 1	1	1
	byte count	*2	3
	input message	1	22 17 (not ready)

* including a **prefix** instruction

Timer

Description

Input from the timer is performed using the **load timer** and **timer input** instructions. **timer input** delays completion until the timer is (>) the value in the A register. If this is already the case when the instruction is executed, then no delay occurs. Waiting on the timer is supported by hardware scheduling mechanisms, and so is not busy.

Examples

occam	instruction sequence	bytes	cycles
TIME ? x	load timer	2	2
	store local x	1	1
TIME ? (>) t	load local t	1	2
	timer input	2	36 4 (no delay)

ALTernative

Description

Alternative input is supported by hardware scheduling mechanisms, and so is not busy. The alternative is implemented by enabling the channels specified in each of its components. The **alternative wait** instruction is then used to deschedule the process until one of the channel inputs becomes ready, whereupon the process is scheduled again. The channel inputs are then disabled. The **disable** instruction is also designed to select the component of the alternative to be executed.

Example

In this example, the guarded processes are not specified. **b0** and **b1** are boolean variables, **c** is a channel, and **x** is a local variable.

```
     occam      instruction sequence       bytes    cycles

ALT
  b0 & c ? x
    P0
  b1 & c ? x
    P1
             L2:alt start                    2        3
                load local pointer c         1        1
                load local b0                1        2
                enable channel               2        7
                load local pointer c         1        1
                load local b1                1        2
                enable channel               2        7
                alt wait                     2        7 18 (not ready)
                load local pointer c         1        1
                load local b0                1        2
                load constant L0-L3          1        1
                disable channel              2        9
             L3:load local pointer c         1        1
                load local b1                1        1
                load constant L1-L4         *2        2
                disable channel              2        9
             L4:alt end                      2        5
             L0:load local pointer x         1        1
                load local pointer c         1        1
                load constant 1              1        1
                byte count                   2        3
                input message                1       22
                P0
                jump M
             L1:load local pointer x         1        1
                load local pointer c         1        1
                load constant 1              1        1
                byte count                   2        3
                input message                1       22
                P1
             M:
```

* including a prefix instruction

Replicated alternative

Description

This is similar to **alternative.** Loops are used to enable and disable the vector of channels. A location in workspace is used to hold the replicator index, permitting one copy of the code to be used for all the components.

Example

In this example, c is a vector of channels and x is a local variable.

```
    occam     instruction sequence        bytes    cycles
ALT i = [0 FOR 10]                              (10 channels)
  c[i] ? x
    P
           L2:alt start                    2        3
              load constant 0              1        1
              store local index            1        1
              load constant 10             1        1
              store local count            1        1
           L1:load local index             1       20
              load local pointer c         1       10
              word subscript               1       20
              load constant 1              1       20
              enable channel               2       70
              load local pointer index     1       10
              load constant L3-L1          1       10
              loop end                     2       95
           L3:alt wait                     2        7 18 (not ready)
              load constant 0              1        1
              store local index            1        1
              load constant 10             1        1
              store local count            1        1
           L4:load local index             1       20
              load local pointer c         1       10
              word subscript               1       20
              load constant 1              1       20
              load constant L0-L5          1       10
              disable channel              2       90
           L5:conditional jump L6          1       38
              load local index             1        2
              store local temp             1        1
           L6:load local pointer index     1       10
              load constant L7-L4          1       10
              loop end                     2       95
           L7:alt end                      2        5
           L0:load local temp              1        2
              store local replindex        1        1
              load local pointer x         1        1
              load local pointer c         1        1
              load local replindex         1        2
              word subscript               1        2
              load constant 1              1        1
              byte count                   2        3
              input message                1       22
              P
```

Timer and SKIP guards

Description

If one or more guards in an alternative takes the form of a timer input, then **timer alt start** and **timer alt wait** instructions are used. **SKIP** guards are directly supported by corresponding **enable skip** and **disable skip** instructions.

Example

In this example, the guarded processes are not specified. **b** is a boolean variable, **t** is a local variable.

```
 occam      instruction sequence       bytes    cycles

ALT
  TIME ? (>) t
    P0
  b1 & SKIP
    P1
            L2:timer alt start            2       5
               load local t               1       2
               load constant 1            1       1
               enable timer               2       9
               load local b1              1       2
               enable skip                2       4
               timer alt wait             2       13 49 (not ready)
               load constant 1            1       1
               load constant L0-L3        1       1
               disable timer              2       31
            L3:load local b1
               load constant L1-L4       *2       2
               disable skip               2       5
            L4:alt end                    2       5
            L0:P0
               jump M
            L1:P1
             M:
```

* including a **prefix** instruction

Other instructions

The T414 transputer includes the following instructions. Where not directly relevant to an occam program, these instructions are provided for use by the development system or to support languages other than occam.

miscellaneous facilities

`minimum integer`	loads the minimum integer
`word count`	array subscription
`check subscript from 1`	
`load byte`	byte arrays
`store byte`	byte arrays
`move message`	slice assignment
`set error`	
`general call`	simple jump and link
`general adjust workspace`	save and set W register

partword arithmetic

`extend to word`	(signed)
`check word`	

long arithmetic

`extend to double`	(signed)
`check single`	
`long add`	
`long subtract`	
`long sum`	
`long difference`	
`long multiply`	
`long divide`	
`normalize`	
`long shift left`	
`long shift right`	

scheduling

```
run process
load current priority
```

booting and analyzing

```
start
test processor analyzing
save high priority registers
save low priority registers
store high priority front pointer
store high priority back pointer
store low priority front pointer
store low priority back pointer
store timer
```