next instruction in the threaded list of the current keyword. The word address register (**WA**) containing the address of the current keyword or the address of the first code body location of the current keyword. The code address register (**CA**) contains the address of the next abstract machine instruction to be executed.

The two stacks are perhaps the most important part of the abstract machine, only by exploiting their unique properties can **TILE**'s reverse Polish notation and thread execution be implemented. The exact sizes of each of the stacks is dependent upon the type of computation required.

4.4 The Instruction Set

The very nature of a **TIL** leads to one of the most unusual design decisions regarding the instruction set of the underlying abstract machine. Because the entire language is based around a small number of mutually exclusive function it is perfectly feasible to implement the entire language as abstract machine instructions. That is for each function within the language there is an equivalent abstract machine instruction.The advantage in both speed and size of this approach is obvious, and indeed there are many commercially available microprocessors that provide just this type of implementation. The structure of a **TIL** also provides yet another strategy when designing the instruction set for the abstract machine. Because of the extendible nature of the language there is a minimum instruction set from which all other function can be defined, thus the instruction set need only contain these essential atomic instruction.

The **TILE** abstract machine can be regarded as a compromise between both of the design philosophies described, providing enough instruction to ensure efficient execution, and few enough to keep the abstract machine to sensible proportions.

4.5 Functional Composition Of The Instruction Set

The instruction set of the **TILE** abstract machine can be divided into ten broad functional categories, each representing one particular aspect of the computational environment.

4.5.1 Arithmetic

All arithmetic operations can be approximated to addition in one form or another. Subtraction, multiplication and division can all be achieved using just the addition operation. For completeness and efficient execution however, it is sensible to implement as many arithmetic functions as possible at a low-level to remove any unnecessary bottle-necks. Arithmetic operations using both signed

and unsigned quantities in single and double precision formats are also a feature which must be taken into consideration when deciding the extent to which arithmetic operations will be incorporated at the abstract machine level.

The **TILE** abstract machine is based around the minimum arithmetic requirements, and provides single and double length addition operations and the corresponding negation operations to convert between integer and 2's complement format, in order to implement subtraction. Unsigned single-length multiplication and division are the only other operations provided by the abstract machine, all other more complex arithmetic functions are defined at a higher level interns of theses lower level primitives.

4.5.2 Logic

In the same way that all arithmetic operations can be approximated to addition and negation. Logic operations can be approximated to a combination of **not** and **and**. However in the interests of efficient execution it is sensible to implement all logic operations separately at the abstract machine level. Conventional languages such as **'C'** provide two types of logical operators, those concerned with Boolean truth values, and those concerned with bitwise or binary values. The overloading of logic operators in this way is entirely the result of a decision to adopt (1) and (0) as true and false truth vales respectively. Such a representation requires a complementary set of Boolean logic operations that perform essentially the same function as their bitwise counterparts. This ambiguity can however be eliminated by adopting (-1) and (0) as true and false truth values respectively, thus removing the need for a second set of logical operators.

The **TILE** abstract machine is based around bitwise logic operators and provides **AND** , **OR** , **XOR** and **NOT** as primitive logic operations, and (=) equal, (<) smaller than and (>) greater than as primitive comparative logic operations. The truth values of (-1) and (0) are conventionally adopted to represent Boolean truth values and prevent the ambiguity described above.

4.5.3 Stack Manipulation

The stack is the major parameter passing mechanism utilised by the abstract machine. It is therefore important to ensure that the stack manipulating instructions are as efficient as possible. For this reason all stack intensive operations are implemented at the abstract machine level.

The **TILE** abstract machine is based around a small number of stack operations and provides **SWAP** , **DUP** , **OVER** and **ROT** to manipulate the top three stack

elements, with a supplementary mechanism **>R** and **R>** to transfer data to and from the return stack to provide additional scratch work space.

4.5.4 I/O

All I/O can be regarded as either a read or write on a specific data stream. For the purposes of clarity, streams are usually divided into input streams (i.e. keyboard and file system) and out put streams (i.e. vdu and file system). All operations upon these streams can then be approximated to either a single or multiple character read or write. The more complex I/O functions can then be defined in terms of these low level primitives.

The **TILE** abstract machine supports two input streams (i.e. the keyboard and file system) and provides a universal line input operation **inline** for both, and a single character input operation **key** for the keyboard only. The file system can be selected as the primary input stream via the **load** operation, however when the input stream is terminated (i.e. an end of file marker is detected) the stream is reset to the keyboard. The only output stream supported is the vdu, with a single character output operation **emit**.

4.5.5 Memory Manipulation

All memory manipulation can be regarded as either a read or write. The exact size of the operation is dependent upon the amount of memory available, and the word length or size of the underlying abstract machine. Conventionally the minimum size of such operations is 8 bits or 1 byte, although there are architectures based around word lengths as large as 64 bits or 8 bytes, and as small as 4 bits or 1 nibble. Having established the word length, all subsequent memory operations can then be approximated to multiples of this fundamental unit.

The **TILE** abstract machine is based around a 16 bit or 2 byte word length, and provides the associated **@** (fetch) and **!** (store) operations to manipulate quantities of the basic length. In addition to the two 16 bit operations, there are also complementary 8 bit operations **C@** (c-fetch) **C!** (c-store) to manipulate byte quantities such as ASCII characters. Although the abstract machine manipulates 16 bit quantities, the memory system supports only 8 bit quantities. Therefore all 16 bit presentations are stored as two consecutive bytes with the most significant byte first.

4.5.6 Dictionary Manipulation

The dictionary constitutes over 90% of a **TIL**. It is therefore inevitable that the abstract machine will at least in part contain operations for manipulating and maintaining this essential component of the language.

The **TILE** abstract machine supports two dictionary oriented operations. The **find** operation to search the dictionary for a specific header, and the **create** operation to generate a new dictionary header. Other more complex dictionary operations such as localised vocabulary searching, constants, variables, and defining words can all implemented at a higher level in-terms of these dictionary primitives.

4.5.7 Numeric Manipulation

The manipulation of numeric information, as both constant and literal representations, is an essential part of the interpreter. Operations to evaluate in-line literals and constant functions must therefore be supported at the abstract machine level.

The **TILE** abstract machine supports two numeric oriented manipulation operations. The **constant** operation to evaluate constant functions and the **literal** operation to evaluate in-line literals.

4.5.8 String Manipulation

The manipulation of string information, as both ASCII representations of numeric values and delimited lexemes, is one of the central mechanisms of a **TIL**. Operations to converts ASCII character string to their numeric equivalent and scan the input stream for the next lexeme must be supported at the abstract machine level.

The **TILE** abstract machine supports two string oriented manipulation operations. The **number** operation to convert an ASCII character string into its numeric representation according the the current working **BASE**, and the **word** operation to transfer the next delimited lexeme from the input stream to the text scratch pad.

4.5.9 Flow Control

Conditional construct such as **if** ... **else** ... **then** , **begin** ... **while** ... **repeat** , and **begin** ... **until** are all variations upon essential the same flow control mechanism. This mechanism must therefore in provided as part of the abstract machine, to enable such higher level constructs to be implemented. All conditional structures can be approximated to two basic flow control mechanisms, a conditional branch

that transfers control depending upon the result of some Boolean expression, and an unconditional branch or jump, that will always transfer control regardless of the context in which it is executed.

The **TILE** abstract machine supports two flow control mechanisms. The conditional branch operation **branch** , and the unconditional branch operation **jump**.

4.5.10 Iteration

Iterative constructs such as **do** ... **loop**, and **do** ... **+loop** are essentially flow control mechanisms, and must therefore be implemented as part of the abstract machine. The flow control for both iterative constructs is depended upon an index incremented by one in the case of the **do** ... **loop**, and by a specified value in the cases of the **do** ... **+loop**.

The **TILE** abstract machine supports three iterative flow control mechanisms. The **do** operation to initiate the loop construct, the **loop** operation to maintain flow control based around an index increment of one, and the **+loop** operation to maintain flow control based around a specified index increment.

4.6 Execution Mechanism

The **TILE** abstract machine is based around a graph representation of function application. Each higher order function or secondaries, can seen as an n-tuple of function applications, which may themselves be n-tuples or abstract machine instructions. Thus the graph for a secondary is constructed from n-tuple function applications at each branch, and abstract machine instructions a each leaf node. Evaluating such a function is simply a case of performing and in-order traversal of the function graph, and executing the abstract machine instructions at each leaf node. The **TILE** abstract machine provides five basic mechanisms for evaluating or executing function graphs. The **colon** operation to mark the start of an n-tuple of function applications, the **semi** operation to denote the end of a n-tuple of function applications, the **next** operation to traverse the graph representing the next function application until a leaf node is encountered, the **run** operation to execute an abstract machine instruction at a leaf node, and the **does>** operations to transfer evaluation to a new point within the graph. Figure 4.6.1 shows the graph for a simple secondary function.

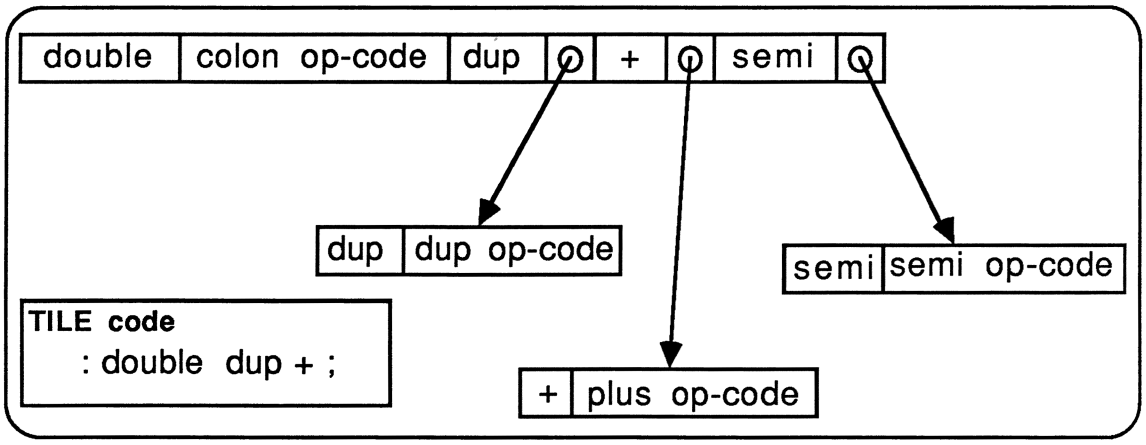

Figure 4.6.1 - Diagram of the graph for a secondary function.

4.6.1 The Colon Mechanism

The word containing the colon operation (**a colon-definition**) must have been called from some other word and initially **I** (**instruction register**) contains the address of a point within the calling word. The first action of the colon operation must therefore be to copy the contents of **I** to the return stack for later retrieval. The contents of **WA** (**word address register**) are then incremented by two to give the address of the start of the new word's parameter field and this address is stored in **I**. A call to **next** then starts the interpretation of the new word.

4.6.2 The Semi Mechanism

The word containing the semi operation must have been called as the last word from a colon-definition. The operation of semi is to transfer the top value from the return stack into **I** and call **next**. This value is the old contents of **I**, placed on the return stack by the **colon** operation and so the effect of the semi is to restore execution back to the point from which the colon-definition was called.

4.6.3 Next Mechanism

The operation of next is to execute the next word in the sequence. The contents of **I** Initially contains the address of the next word to be executed. The contents of the address contained within **I** are then therefore transferred to **WA** and **I** is incremented by two to point to the next word in the sequence. A call to **run** then starts execution of the new word.

4.6.4 Run Mechanism

The operation of run is to execute the abstract machine instruction in the sequence. The contents of the address contained within **WA** are then transferred

to **CA (code address register)** and **WA** in incremented by two to point to the next instruction in the sequence.

4.6.7 Does> Mechanism

The operation of does> is to execute the sequence of words following does> in the creating word. The action of does> is identical to that of **colon** save that before **next** is called the address of the parameter field of the word is copied onto the parameter stack.

4.7 Termination

Terminate either normally as the result of program completion, of abnormally as a result of some erroneous condition, is a facility that must be provided as part of the abstract machine.

The **TILE** abstract machine supports two termination operation. The **trap** operation which causes the abstract machine to trip back to the interpreter, and the **exit** operation which causes the abstract machine and the interpreter to terminate and return control to the operating system.

4.8 Implementation

The **TILE** abstract machine has been implemented as two separate modules, the architecture as described in section 4.3 and the instruction set as described in sections 4.5 though to 4.6. Both of theses modules have been further decomposed into their constituent components, each which is implemented separately as a **#include** library. The hierarchical structure of the abstract machine implementation, is a deliberate attempt to control complexity and improve the overall clarity of the source code.

4.8.1 Memory

The memory has been implemented as a 64K block of contiguous unsigned characters. The block is allocated with a **malloc** commend when **TILE** is first executed, and released with a **free** command upon termination.

```
char *base ;

void  claim_memory()
{
        base = (char *) malloc (BLOCKSIZE) ;
}
```

The memory manipulation is based around the get_byte and put_byte procedures. The **get_byte** procedure takes a base relative address and returns a pointer to the unsigned character that represents the contents of that address. The

put_byte procedure takes a base relative address and unsigned character, and stores the character at the position represented by the address.

```
unsigned short get_byte(addr)
unsigned short addr ;
{
        char *place ;
        place = base + addr ;
        return(*place) ;
}
```

4.8.2 Stacks

The two stacks have been implemented as two vectors of unsigned short's. This implementation was chosen over the dynamic equivalent (which is also implemented) simply because of the enormous performance overhead that would result from a linked list implantation.

The stack manipulation is based around four operations which are identical for both the return and parameter stacks. The **push** procedure push a an unsigned short onto the top the stack, and checks that overflow has not accrued. the **pop** procedure removes the top unsigned short form the stack, and checks that under flow has not accrued. The **reset** procedure resets the stack pointer to top of the stack, thus removing any data that may have been present. The **empty** procedure returns **TRUE** if then stack is empty, or **FALSE** if not.

```
unsigned short psstk[ PSTKSIZE ] ;

void push_ps(n)
        unsigned short n ;
{
        if( ! trapped())
                if( ps == PTSKSIZE )
                {
                        printf("Parameter stack overflow.\n") ;
                        forth_abort() ;
                }
                else
                pstk[ps++] = n ;
}
```

4.8.3 Arithmetic

The arithmetic described in section 4.5.1 is based around the unsigned short arithmetic operators provided as part of the basic **'C'** environment. The procedures **add_op** is a typical example of the structure of the single length arithmetic operations.