J. King Saud Univ., Vol. 6, *Eng. Sci.* (2), pp. 167–184 (A.H. 1414/1994)

COMPUTER ENGINEERING

Imposing an FP Layer on a Risc Machine

M.A. El-Affendi

Department of Computer Science, College of Computer and Information Sciences, King Saud University, P.O. Box 51178, Riyadh 11543, Saudi Arabia

(Received 30/12/1990; accepted for publication 16/12/1992)

Abstract. In this paper a simple abstract machine for the translation of FP programs is suggested. The machine simply consists of an instruction selection register (ISR), a pipeline control register (PCR) and a large collection of general purpose registers. It is illustrated how the FP structures may directly be mapped on the ISR and executed. Although the main purpose of the abstract machine was to simplify the translation of FP programs, it is shown that the suggested machine may as well be used to extend RISC instruction sets in a proper healthy way. This will have the effect of simplifying the translation of procedural programs written in any other language. The dynamic nature of the FP combining forms makes it possible to tailor any sort of intermediate instruction suitable for the particular application. The machine may either be implemented at the macro-assembler level or at the firmware level using efficient microcode.

Introduction

Few years ago Backus [1,2] introduced an ambitious functional programming system different from conventional programming systems in many respects. His motivation then was to provide a system that liberates programming from the long adopted von-Neuman habits and traditions. The FP system is not just a programming system but a full algebraic specification system in which solutions to problems may be specified, validated and proven correct using algebraic rules and theorems.

The introduction of the FP programming style coincided with the appearance of the RISC design approach [3] as an alternative to the CISC approach in the late 70's and early 80's (the term RISC stands for Reduced Instruction Set Computers while the term CISC stands for Complex Instruction Set Computers). The central feature of RISC architectures is that their instruction sets are reduced in size and include only

primitive (single cycle) instructions. This reduced instruction set is implemented directly in hardware to provide for maximum efficiency.

For people working in the area of compiler design the RISC approach implies that they should no longer expect the intermediate level complex instructions previously provided by CISCs to close the gap between high and low level languages. For functional programming designers this may be a relief rather than a curse. The relief comes from the fact that most of the complex instructions provided on CISCs are not very suitable for functional program translation and designers used to rely on some intermediate abstract machines to produce intermediate code which is finally translated into actual machine code. Good examples of these abstract machines are the SECD machine for LISP [4] and the G-machine used for Miranda [5]. As for FP systems, the RISC approach is naturally compatible with its built-in philosophy that neatly separates higher order functions from low level primitive functions. In fact the FP style may be used to compliment RISC machines in the proper way without introducing any redundancies. The FP combining forms may be used to add further modular layers of complex instructions on the original instruction set. The topmost layer may correspond to a high level programming language while the bottom layer may correspond to a macro-assembler or a microcode interpreter.

In this paper, it is illustrated how the bottom FP layer may be superimposed on a RISC machine. An abstract FP machine is described and hints about its use in the enhancement of the host instruction set are given. The original motivation for introducing the machine was to simplify code generation for FP programs, but it turned out that the machine may even be used to generate code from procedural programs very neatly. This is mainly due to the dynamic and flexible nature of the FP combining forms.

Backus FP System

The FP system described by Backus is mainly composed of the following:

i) A set of objects which are the data items of FP systems. Objects may either be atoms (simple data types or constants) or sequences (compound data items).

ii) A set of primitive functions that include the basic operations such as ADD, MULT, DIV, SUB, EQ, AND, OR, NOT ... etc. together with some special list operations such as TAIL, APPEND, DISTRIBUTE, TRANSPOSE, SELECT. For more details on these operations the reader is urged to see [1,2].

iii) A set of combining forms (higher order functions) that may be used to combine existing functions in order to produce useful programs. The combining forms are the main tools for program construction in FP systems.

Based on this FP system Backus defined an FFP system (Formal systems for Functional Programming) that uses the FP combining forms and a metacomposition rule to write functional programs. In what follows the combining forms and metacomposition rule are discussed in some detail since they are the main source of FFP power.

The combining forms

FP combing forms include the following:

(1) Composition: Backus uses the circle symbol to denote algebraic composition, for example f c g:x is the same as f: (g:x) where the colon indicates function application. In this example g is applied to the object x first and then f is applied to the result. Composition may involve any finite number of functions which may primitive or non-primitive.

(2) Construction: construction is denoted by the square brackets and is used to apply several functions to the same data object e.g.:

$$[f_1, \ldots\ldots, f_n] : x \text{ is the same as } \langle f_1 : x, \ldots\ldots, f_n : x \rangle$$

where the angular brackets here denote a sequence structure.

(3) Condition: in FP notation a conditional expression is written as:

$$(p \text{ -- } > f;g] : x \text{ and is interpreted as}$$

if (p:x) evaluates to true then the result is (f:x) else the result is (g:x)

(4) Insert: the insert combining form is denoted by the foward slash / and is used to apply a function to a sequence of objects step by step by step in a semirecursive manner. The order of evaluation is similar to that of composition but the semantics are different and may be explained by the following pseudo-Pascal algorithm:

Given a sequence $\langle x_1, \ldots\ldots, x_n \rangle$ then $/f:\langle x_1, \ldots\ldots, x_n \rangle$ may be interpreted as follows:

```
        g: = x_n;
for k: = n downto 2 do
begin
    y: = g;
    g: = f: (x_{n-1}, y);
end;
f: = g;
```

(5) Apply to all: this combining form is denoted by the Greek letter α and is used to apply a function simultaneously to a sequence of objects. Given a sequence $<x_1, \ldots, x_n>$ and a function f then:

$$\alpha\, f:<x_1, \ldots, x_n> \text{ is the same as } < f:x_1, \ldots, f:x_n>$$

The metacomposition rule

To facilitate efficient implementation of the above abstract combining forms Backus introduced a more practical FFP system, the semantics of which are well described in [1,2]. In the FFP system a new constructor $(e_1:e_2)$ which denotes the application of an expression e_1 to another expression e_2 has been introduced. In this constructor e_1 is in most cases a sequence of functions and combining forms $<h_1, \ldots, h_n>$ and e_2 is a sequence of objects $<x_1, \ldots, x_n>$. In this case the semantics of the constructor are specified by a metacomposition rule according to which:

$(<h_1, \ldots, h_n>:<x_1, \ldots, x_n>)$ is interpreted as:

$(h_1:<<h_1, \ldots, h_n><x_1, \ldots, x_n>>)$

The main motivation behind the metacomposition rule is to be able to represent the combining forms as a composed sequence of functions and/or other combining forms. In order for the metacomposition rule to work an FP representation for h_1 that produces the desired effect should be chosen and stored beforehand, *e.g.* The combining form construction may be represented as the composition:

construct = apply to all o tail o distribute right

In view of this program:

(<construct, tail, id>:<A,B>) may be interpreted as:

(construct: << construct, tail, id><A,B>>) according to the metacomposition rule. The FP representation of the construction combining form may then be applied as follows to produce the required result:

i) Start by applying the rightmost function in the composite representation which is distribute right. This gives:

<<construct, <A,B>>,<tail,<A,B>>,<id,<A,B>>.

ii) Then apply the next function in the composition which is the tail function. The yields:

<<tail,<A,B>>,<id,<A,B>>>

iii) Finally we apply the last function in the composition apply to all which gives:

<<B>,<A,B>> which is the final result of the program.

Assumptions About the Abstract Machine

As stated in the introduction, the main purpose of this work is to show how an FP layer may be imposed on a RISC machine. As will be explained below the FP layer is supposed to serve the dual purpose of enhancing the host instruction set and simplifying the translation of higher level FP programs. By definition the RISC instruction set is a reduced set consisting of a number of hardware implemented primitive instructions. It is assumed that the number of these instructions does not exceed 32.

The suggested FP layer is implemented as an abstract machine consisting of a set of abstract registers together with a set of software or firmware algorithms supporting execution on the machine. The set of registers include:

i) A-64 bit Instruction Selection Register (ISR). This register is supposed to hold the bitmap specification of the currently executing combining form as will be explained below.

ii) A Pipeline Control Register (PCR) to control intermediate level pipeline execution.

iii) A large number of 32-bits general purpose registers to hold data items.

At this stage it may be assumed that the FP list operations are not provided as part of the RISC instruction set and should be implemented as an add-on facility. This is not an easy matter and greatly depends on the implementation of sequences. For simplicity here it is assumed that only homogenous contiguous sequences are used. It is also assumed that all sequences relating to the current instruction may be loaded on the general registers. Assuming that there are about 128 general purpose registers (which is normal in RISC designs), then each sequence may abstracted as a 4-byte integer:

start address	element size	length	type

The start address is the place of the first element in the register pad, the element size is the length in byte of each element in the sequence, the length is the total number

of elements in the sequence and the type field is a flag which is zero if each element in the sequence is a primitive data item and one if each element is again a sequence pointer. This allows for multiple level sequences. Note that we need not use the full byte for the type flag but the byte may have other uses later on.

Using this sequence abstraction, the list operations may easily be implemented by shuffling sequence pointers. *e.g.* The distribute right operation:

$$\text{distribute r} :<<x_1, \ldots, x_n><y_1, \ldots, y_n><$$

may be implemented as a new sequence map in which a pointer to each element x_i is followed by a pointer to the full sequence y. Notes that in order to maintain homogeneity each primitive element x_i is replaced by a sequence pointer pointing to a sequence consisting of just x_i. The creation of the new sequence map requires the use of MOV and bit-shift operations which are supposed to be available on the host machine.

The above scheme may be generalised to nonhomogeneous sequences by adding size and type prefixes to each element in the sequence. In that case the element size field in the sequence pointer is no longer needed and may be discarded.

$$\text{RISC} + \text{FP LAYER} = \text{CISC}$$

One of the most interesting features of FP systems is the neat separation between low level primitive functions and the higher order functions that may be constructed using combining forms. It may be claimed that this sort of segregation is also implicitly inherent in the RISC design philosophy. RISC designers differentiate between simple primitive instructions that may be implemented as part of the instruction set and complex instructions which are excluded at the hardware level. Their main motivation is the claim that efficient implementation of a reduced set of primitive instructions is much better for performance than a fat, heterogenous set of complex instructions. Perhaps one of the main facts justifying this claim is the implicit dependence of complex instructions on primitive instructions in the same set. Consider for example the VAX-11 [6; pp. 172-200, 7; pp. 179-207] instruction set which is one the classic examples of CISC designs. Most of the instructions in this set may be interpreted as functions of the form:

$$f: A \rightarrow B$$

where A is the domain and B is the codomain. These functions belong to one of two categories:

i) Primitive functions that may not be interpreted in terms of other junior functions.

ii) Higher order functions that may be composed of other junior functions using one of three methods:

(a) Generalized insertion: *e.g.* The bit clear operations BICx may be interpreted as follows:

ASSIGN (OP3, AND (NOT (OP1), OP2))

Here the function NOT is applied first to OP1, then the result is ANDed with OP2 and finally the result is assigned to OP3. This type of insertion is quite common in the VAX-11 instruction set and sometimes complements the work of other function construction methods. Note that this type of insertion is a generalisation of the insertion process used in Backus FP system. Here different operations are applied at the various stages of the insertion process.

(b) The conditional functional combining form an example of which is the BNEQ instruction which may be interpreted as follows:

COND: EQ(Z,0) → ASSIGN(PC,ADD(PC,DISP))

Here the second part of the function (after the arrow) is executed only if the condition on the left (Z = 0) is satisfied. Another interesting example of this functional form is the SOBGTR instruction which may be interpreted as follows:

COND: LSS(DEC(INDEX),0) → ASSIGN(PC,ADD(PC,DISP))

(c) The block functional combining form, an example of which is the procedure call instruction CALLG which may be interpreted as follows (for details see [6]):

[*ALIGNLONG*(SP),*PUSH*(*GREGISTERS*,*PC FP*,*AP*),*CLEAR*(*N*,*Z*,*V*,*C*),*PUSH* (*PSW*),*PUSH*(*ZERO*(*LONGW*)),*ASSIGN*(*FP*,*SP*),*ASSIGN*(*FP*,*ARGLIST*),*SET* (*TRAPBITS*),*ASSIGN*(*PC*,*PROCNAME*+2)]

The block combining form may be thought of as a generalisation of the construction combining form.

The implication of the above observations is that FP systems may play a complementary role in the design of RISC machines. In particular the FP combining

forms may be used to add another layer of more complex instructions on the reduced set of primitive instructions. In fact this FP layering approach may be pursued further to produce higher level constructs similar to those found in high level languages.

In the section below we illustrate how a layer of complex instructions may be added on a primitive set using some of the FP combining forms.

The Implementation of Combining Forms

In this section it is shown how FP combining forms may be used to define new compound instructions that are a smooth extension of the existing instruction set. The listed ideas may either be implemented in microcode [8; pp. 86-106], if the machine allows or at the macro assembler level otherwise. It is assumed that the leftmost 3 bits of the host machine instruction opcodes are always 0. This is consistent with the previous assumption that at most 32 primitive instructions are available. In view of this the abstract machine differentiates between six types of instructions:

000	primitive instructions
001	composition instructions
010	construction instructions
011	apply to all instructions
100	insertion instructions
101	conditional instruction

The implementation of these types of instructions is detailed below.

Composition instructions

Instruction format

As implied by the discussion of an earlier, section a composition instruction may be viewed as a compound instruction in which a nested set of primitive instructions are applied one at a time to obtain the desired effect. Execution starts with the innermost instruction and moves towards the outer ones. The outcome of each primitive instruction in the nested set depends on the outcome of its immediate inner predecessor. In view of this a composition instruction may be implemented as follows:

Here the opcode is a 16-bit pointer to a quadword specification (QSPC) stored within control memory or the application program area, depending on the implementation. The leftmost three bits of the opcode and the QSPC are set to 001 indicating composition mode. The next bit may be used to indicate if the composition instruction is available in system area (frozen) or in user program area (dynamic). The remaining 60 bits of the QSPC are meant to hold the codes for the primitive instructions comprising the composition form. Assuming that the codes for the primitive instructions are from 0 to 31, then five bits are needed to hold the code for each primitive instruction. This means that the QSPC may hold at most 12 primitive instructions, the innermost of which is on the rhs and the outermost is on the lhs.

The remaining bits following the QSPC pointer are reserved for operand specification. The number of operands depends on the type of instruction.

The execution algorithm

When the processor encounters an instruction for which the leftmost three bits are 001, it realizes that this is a composition instruction and proceeds as follows:

i) load the QSPC on the ISR.
ii) set the rightmost PCR bit and clear all remaining bits. This indicates that in the beginning only the innermost instruction in the composition can proceed to handle the operands.
iii) Initiate a pipeline consisting of all the instructions comprising the composition. Theoretically each instruction in the pipeline can proceed in parallel with the other instructions until it reaches a point where it requires the value of the operands. At this point the processor checks the corresponding bit in the PCR. If this bit is set then the instruction may proceed to handle the operands in the data registers of the composition, otherwise it sleeps until the bit is set. When a primitive instruction finishes, it immediately enables the next instruction in the composition by setting the corresponding bit in the PCR. It is expected that each primitive insitruction deposits its results in the appropriate work registers.
iv) The processor finally retrieves the instruction outcome from the work registers.

Construction instructions

Instruction format

The instruction format is very similar to that of composition. Again the opcode is a pointer to a QSPC with the leftmost three bits set to 010. The QSPC specifies up to 12 instructions that may simultaneously be applied to the given operands. Like composition instructions, the operands appear after the QSPC.

Execution

This type of instruction is more suitable for machines with multiple functional units [9; pp. 272-284] where a number of instructions may be executed in parallel. However, on a sequential machine the instructions may be executed in any arbitrary order.

Note: The above construction scheme may easily be generalized to become a block combining form. In this case each operation may be applied to a different set of operands.

Apply to all instructions

Instruction format

The opcode is again a pointer to a QSPC whose leftmost three bits are 011. The remaining part of the QSPC is used to specify a primitive operation and an operand which should be a sequence in this case.

Insert instructions

Instruction format

The same as that of apply to all with the only difference that the leftmost three bits of the opcode and the QSPC are 100.

Execution

The operation specified is first applied to the rightmost double elements of the specified sequence. Execution then proceeds as specified by the algorithm given in an earlier section.

Note: The insertion scheme may easily be generalized to allow for different operations to be applied at the various stages of the insertion process. e.g. in the beginning a + operation may be applied followed in the next phase by * operation and so on.

Conditionals

As shown in section 1 above many of the VAX-11 instructions may be encoded as conditionals of the form:

$$COND: \text{EXP1} \rightarrow \text{EXP2};\text{EXP3}$$

where EXP1 is a conditional expression, EXP2 is an expression that will be executed if EXP1 is true and EXP3 is an alternative expression that will be executed if EXP1 is false. In this section we show how such an instruction may be implemented when each of the three expressions above are of the composite type (*i.e.* formed of primitive instructions using the composition combining form described above).

what is suggested here is that available CISC designs may easily be modified to include a hardware implemented primitive layer plus a microcode FP layer. This may improve performance and help to maintain compatibility with previous designs.

A third use of the suggested machine is to provide users with a mechanism to define their own FP structures. e.g. at the macro-assembler level a user may be allowed to right his own compositions by using an instruction:

$$\text{COMP } n, f_1, \ldots, f_n, \text{data}$$

where n is the number of operations to be composed. In this case the instruction internal representation will be built in the user program area. This may even be pursued at higher level languages by allowing statements such as:

$$\text{COMPOSE } f_1, \ldots, f_n \text{ FOR data;}$$

or INSERT operation IN sequence;

to appear as high level constructs. e.g. The sum of n numbers $x_1, \ldots, x_n$ may simply be obtained using the statement:

$$\text{sum IS INSERT + IN} < x_1, \ldots, x_n >;$$

For more on this see [10].

A Simulated Example Machine

To illustrate the practical uses and applicability of the abstract FP layer described above, a simulated intermediate FP machine is presented and discussed in this section. In addition to the earlier assumptions, it is assumed that the underlying hardware machines satisfied the following assumptions:

i) The availability of multiple arithmetic and logic units which make the machine capable of performing more than one operation at a time [9; pp. 272-284].

ii) The built-in instruction set includes the basic arithmetic, logical and shift operations such as ADD, MUL, DIV, SUB, AND, NOT, OR, TEST, SHL, SHR, GR, LT, GE, LE, NE, EQ ... etc.

iii) The availability of a group of stacks with built-in operations PUSH and POP.

iv) The availability of more than one ISR (Instruction Selection Register).

Based on these assumptions, the example machine provides the following set of intermediate instructions:

Arithmetic oplist

where OPLIST is a list of arithmetic operations such as +,*,/,− . The OPLIST is obtained by converting the given expression into the generalized insertion form described earlier. For large expressions, each subexpression may lead to a generalized insertion form that may be dealt with independent of the other subexpressions. In this case square brackets may be used to indicate the priorities and the degree of concurrency, e.g. The arithmetic expression:

K*A+B/(X−Y/D) becomes ADD(MUL,(K,A),DIV(B,SUB(X,DIV(Y,D))))

which consists of two subexpressions MUL(K,A) and DIV(B,SUB(X,DIV(Y,D))). The evaluation of these two subexpressions may proceed concurrently, eventually feeding their results to the ADD operation. On our machine this is indicated using square brackets as follows:

ARITHMETIC +,[*],[/,−,/]

This expression requires to stacks, one for each subexpression. The corresponding operands are loaded on these stacks using the LOADSTACKS operation described below. The work of the instruction is simulated by the OCCAM [11,12] processes given in the appendix. The machine generates two pipelines, one for each subexpression. In each pipeline, execution proceeds left to right. Each operation takes its operands from the top of the corresponding stack and deposits its results on top of the stack. In the end each pipeline informs the parent process which performs its operations using the tops of the stacks.

Loadstacks operandlist

This instruction has a complimentary role to the ARITHMETIC instruction above and the PREDICATE instruction below. It merely prepares the stacks for the execution of these instructions. For example, in case of the above expression the LOADSTACKS operation may be called to load the operands on the corresponding stacks as follows:

LOADSTACKS [K,A] , [B,X,Y,D]

In this case D will be loaded on top of stack 1 followed below by Y,X,B respectively. A will be loaded on top of stack 2 with K immediately below it.

Predicate oplist

This instruction is very similar to the ARITHMETIC instruction above with the only difference that it involves both Boolean and arithmetic operations and returns a Boolean value. This type of expression is very useful in controlling loops and selections. The LOADSTACKS operation may againt be used to load the corresponding operands.

Insert OP, list

This instruction is a direct implementation of the insertion combining form described in section (5.4) above. The pipeline process given in the appendix is a direct simulator for the generalised form of insertion.

Compose OP, list

This is a direct implementation of the composition combining form described in an earlier section.

Apply to all OP, list

This is a direct implementation of the APPLY TO ALL combining form shown above.

Select entry, exit

This is a simple instruction that emulates the GARD and IF statements in high level languages. The entry is this case is a PREDICATE representing the first conditioin to be tested in the IF-THEN-ELSEIF .... structure. If this PREDICATE evaluates to true, the block of code immediately following it is executed. The last byte in the ENTRY predicate and all other predicates in the selection structure is a field indicating the offset for the next predicate to be tested if the current predicate evaluates to false. The last predicate in the sequence contains zeros in this field. The EXIT field indicates the end of the selection structure.

Loop predicate, exit

This is a direct emulation for WHILE loops in high level languages. The block following the LOOP instruction is executed as long as the PREDICATE is true. EXIT indicates the end of the block to be repeated.

Conclusion

It has been illustrated how a simple FP layer may greatly improve a RISC

instruction set and simplify the translation of functional programs. The suggested FP layer is based on Backus FP combining forms and may be used to define control-flow instructions (COMPOSITON, CONDITION, CONSTRUCTION) or data-flow structures (APPLY TO ALL, INSERT). It is suggested that this approach should be further extended to provide an alternative multilayer approach to general program translation, since procedural programs may easily be converted into functional forms.

APPENDIX
A Simple Occam Simulator for the Suggested Machine

Due to the fact that the ARITHMETIC instruction is the most characteristic feature of the machine suggested in section. On Simulated Example Machine, an occam simulator for a two-level version of this instruction is given below. The behavior of other instructions may easily be deduced from the behavior of this instruction.

In the lower level the procedure evaluates two subexpressions in parallel, using two ISR's (Instruction Selection Registers) and two stacks. Each ISR is handled by a pipeline process consisting of a set of parallel pipeunit processes. Each pipeunit may stand for an arithmetic function unit on the real machine. The work of the parallel pipeunits is synchronised by a PCT (Pipeline Control Token). Before proceeding to handle the operands on the stack, each pipeunit has to wait for the PCT to be passed from the preceding unit in the pipeline. At the upper level the main process applies the root op to the results yielded on the stacks by the two parallel pipelines.

For simplicity, we assumed that there are only five binary operators, but the process may be generalised easily for any number of operators and for any number of levels.

It should also be noted that the pipeline process used in this simulator may be used as it is to implement generalised insertion. It is also clear that the ARITHMETIC instruction simulator below may easily be modified to simulate the work of PREDICATE instruction. The only change needed is to substitute Boolean variables and operators for their arithmetic counterparts.

```
PROC arithmetic (INT root.op,tp1,tp2,INT64 isr1,INT64 isr2, [] INT stk1, [] INT
stk2)
  [12] CHAN OF INT pipe:
  [10] INT gstk:
```

```
    INT gtop,top1,top2:
    PROC apply (INT opcode, [] INT stack, INT top)
     INT result:
     SEQ
      CASE opcode
      1 result:=add (stack [top], stack [top-1])
      2 result:=sub (stack [top], stack [top-1])
      3 result:=mult (stack [top], stack [top-1])
      4 result:=div (stack [top], stack [top-1])
      5 result:=power (stack [top], stack [top-1]
      ELSE
        SKIP
      top:=top-1
      stack [top]: = result
      --- end of apply procedure

PROC pipeunit (INT order, opcode, stktop, [] INT stk)
  INT pct:
  SEQ
    pipe [order] ? pct              -- wait for previous pipeunit to finish
    apply (opcode, stktop, stk)
    pipe[order + 1] ! pct           -- trigger next pepeunit
  : -- end of pipeunit procedure
PROC pipeline ([] INT stack, stktop, INT 64 isr)
    INT64 mask:
    [12] opcode:
    SEQ
     mask: =31
     SEQ k = 0  FOR 12      -- extract the opcode from the ISR
      opcode[k]:=isr ∧ mask
      mask: = mask << 5
    PAR
      pipe[0]! pct -- feed the first pipeunit
      PAR i = 0   FOR 12
       pipeunit (i, opcode [i], stktop, stack)
      pipe[12]? pct -- get indication that the work is finished
    :--end of pipeline procedure
    -- body of main procedure ARITHMETIC
```

```
    SEQ
     top1:=tp1
     top2:=tp2
     gtop:=2;
     PAR
      pipeline(stk1, top1, isr1)
      pipeline(stk2, top2, isr2)
      gstk[gtop]:=stk1[top1]
      gstk[gtop-1]:=stk2[top2]
      apply (rootop,gstk,gtop)
:-- end of main procedure
```

References

[1] Backus, J. "Can Programming be Liberated from the von Neuman Style? A Functional Style and its Algebra of Programs." *CACM*, 21, No. 8 (1978), 613-641.

[2] Darlington, J. Et. Al. *Functional Programming and its Applications*. Cambridge: Cambridge University Press, 1982.

[3] Patterson, D.A. "Reduced Instruction Set Computers." *CACM*, 28, No. 1 (1985), 9-21.

[4] Henederson, P *Functional Programming Application and Implementation*. Englewood Cliffs, New Jersey: Prentice Hall, 1980.

[5] Jones, S. *The Implementation of Functional Programming Languages*. Englewood Cliffs, New Jersey: Prentice Hall, 1987.

[6] Baase, S. *VAX-11 Assembly Language Programming*. Englewood Cliffs, New Jersey: Prentice Hall, 1983.

[7] Brink, J. & Spillman, R. *Computer Architecture and VAX Assembly Language Programming*. Menlo Park, California: Benjamin/Cummings Pub. Company, 1987.

[8] Banerji, D.K. and Raymond, J. *Elements of Microprogramming*. Englewood Cliffs, New Jersey: Prentice Hall, 1982.

[9] Mano, M.M. *Computer System Architecture*. Englewood Cliffs, New Jersey: Prentice Hall, 1982.

[10] El-Affendi, M.A. "Sunbla: An Intermediate Step in a Gradual Promotion Model for Natural Arabic Programming." a departmental report, *CSC/AFF/SMB/01*, Dept. of Computer Science, KSU, Riyadh.

[11] Inmos Limited: *Occam 2 Reference Manual*, New York: Prentice Hall International Computer Science Series, 1988.

[12] Inmos Limited: *Occam*, Provo, UT: Computer System Architects, 1989.

إضافة طبقة وظيفية على حاسبات الأوامر المختصرة

محمد أحمد الأفندى

قسم علوم الحاسب، كلية علوم الحاسب والمعلومات، جامعة الملك سعود،

الرياض . . . ، ص.ب ٥١١٧٨، المملكة العربية السعودية

(استُلم في ١٩٩٠/١٢/٣٠م؛ قُبل للنشر في ١٩٩٢/١٢/١٦م)

ملخص البحث. تشرح هذه الورقة كيف أن ماكينة وظيفية وسيطة يمكن أن تُيسر بقدر كبير ترجمة البرامج الوظيفية وغيرها من البرامج. الماكينة المقترحة تتكون من خازنة لحفظ سلاسل الأوامر، خازنة للتحكم في الأوامر المتزامنة، ومجموعة كبيرة من الخازنات متعددة الأغراض.

تبدأ الورقة بشرح أساسيات نظام بُكْس للبرمجة الوظيفية ثم تشرح كيف يمكن أن يستخدم هذا النظام في تحسين أوامر الحاسبات المختصرة وذلك بإضافة طبقة من الأوامر الوسيطة. بعد ذلك توضح الورقة أوجه التطبيقات المختلفة للماكينة المقترحة ثم تعطي مثالاً حيًّا على استخدامها. في الملحق تستخدم لغة أوكام في تمثيل بعض وظائف الماكينة المقترحة.