Interpreting Programs in Static Single Assignment Form

Jeffery von Ronne *Ning Wang* *Michael Franz*

Technical Report 03-12
School of Information and Computer Science
University of California, Irvine, CA 92697-3425

April 2, 2003

Abstract

Static Single Assignment Form is a common intermediate representation for optimizing compilers but several of its features are difficult to implement in a traditional imperative interpreter. An interpreter for Static Single Assignment would enable mixed-mode (interpreting and compiling) virtual machines to be developed around program representations in Static Single Assignment Form. We propose an extension to Static Single Assignment Form, Interpretable Static Single Assignment (ISSA). Each of Interpretable Static Single Assignment's instructions, including the phi instruction, has self-contained operational semantics. In addition, we describe a prototype interpreter of Interpretable Static Single Assignment and report on its performance.

Contents

List of Figures

List of Tables

1 Introduction

For more than a decade, intermediate representations based on Static Single Assignment (SSA) Form [Alpern et al., 1988, Rosen et al., 1988] have been utilized inside a large number research and industrial compilers. More recently, Amme et al. [2001] proposed an external program representation based on SSA Form, which is well suited for Just-In-Time (JIT) compilation. JIT compilation, however, imposes a startup delay, which may not be justified for short-lived applications. Java Virtual Machine implementations typically use mixed-mode interpretation and compilation in an attempt to combine the benefits of interpretation's shorter startup times and JIT compiled code's better throughput. Due to several problematic features, conventional imperative interpreters have not been written for SSA representations. (There have, however, been some extensions to SSA which support data driven interpretation, e.g., Ballance *et al.*'s 1990 Program Dependence Web and Ananian's 1999 Static Single Information Form.) Consequently, Krintz [2002] recently proposed storing and transporting programs in both JVM class files (which use a stack-based virtual machine) for interpretation and also in SafeTSA classes (which are SSA-based) for JIT compilation, allowing mixed-mode operations at the cost of supporting two program representations.

Several features of SSA Form are particularly challenging for direct imperative interpretation: the large number of variable names, the selection of ϕ-function operands, the simultaneous execution of mutually dependent ϕ-functions, and the handling of non-scalar variables. These difficulties, however, are not insurmountable.

This paper presents Interpretable SSA (ISSA) Form, an SSA variant in which each instruction has directly interpretable operational semantics, discusses a prototype implementation, its safety, and its performance, and proposes several possible improvements.

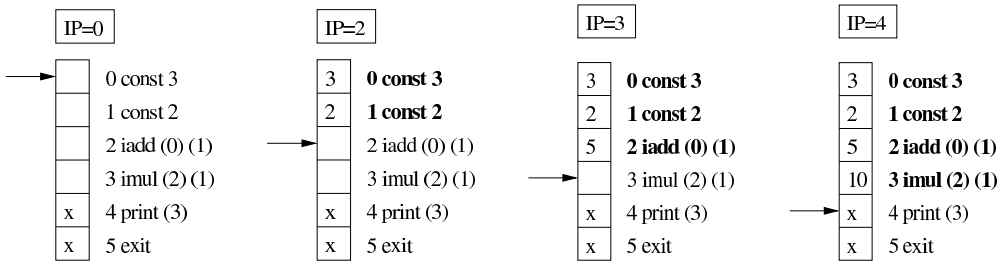
2 Interpretable SSA Form

2.1 Unique Naming

The most important characteristic of Static Single Assignment is that the left hand side of each and every variable assignment must have a unique name. As a result, each original program variable has several corresponding SSA variables. Mapping these SSA variables to abstract machine registers would seem to require an

(a) source code	(b) stack-based code	(c) SSA Form	(d) ISSA code
x = 3	iload 3		
y = 2	iload 2	x0 := mov 3	0 const 3
z = x + y	iadd	y0 := mov 4	1 const 2
v = z * y	iload 2	z0 := iadd x0 y0	2 iadd (0) (1)
print v	imul	v0 := imul z0 y0	3 imul (2) (1)
exit	print	print v0	4 print (3)
	exit	exit	5 exit

Figure 1: a simple program



(a) (b) (c) (d)

Figure 2: execution of a simple program

abstract machine with an unbounded register file. In actuality, each SSA variable is defined by exactly one program instruction (the right hand side of the assignment), so it is sufficient for a program's abstract machine to have one output register per instruction.

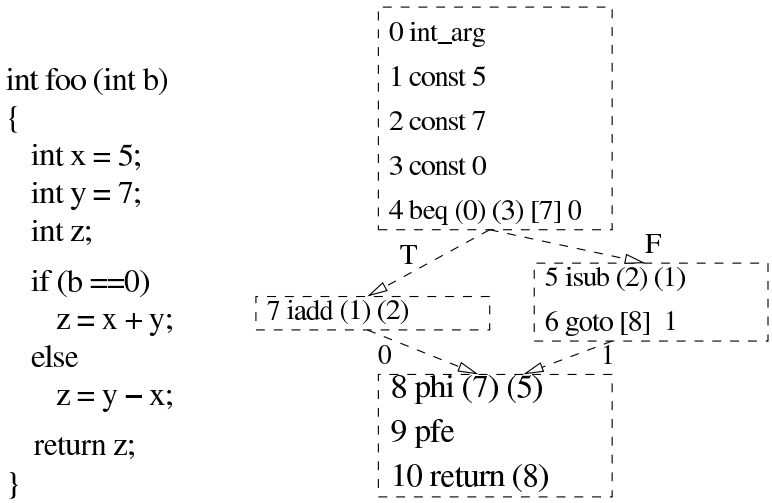
Figure 1(d) shows the program in Figure 1(a) translated into our particular SSA based intermediate representation, Interpretable SSA (ISSA). Each instruction in ISSA is labeled (on the left) with a consecutive instruction line number. A few instruction types, such as the `const` instructions, take immediate integer values, but most have indirect operands which refer to the output of previously executed instructions. Each of these operands selects the defining instruction's output register by the instruction's line number (which is syntactically distinguished by parentheses).

Figure 2 shows the dynamic execution of this program's abstract machine. The instructions' output registers are shown by the boxes to the left of the instructions; an auxiliary *instruction pointer* (IP) register is used to indicate the instruction to be executed next. As each instruction executes, it retrieves its inputs from from the indicated registers, performs its computation, and writes to the appropriate output register (OR) on its left. For example, before instruction 3 executes, the machine state is as show in Figure 2(c); as it executes it reads the values of OR2 (i.e. 5) and OR1 (i.e. 2), multiplies the two, and writes the result (i.e. 10) to OR3.

2.2 Choosing Φ-function Operands

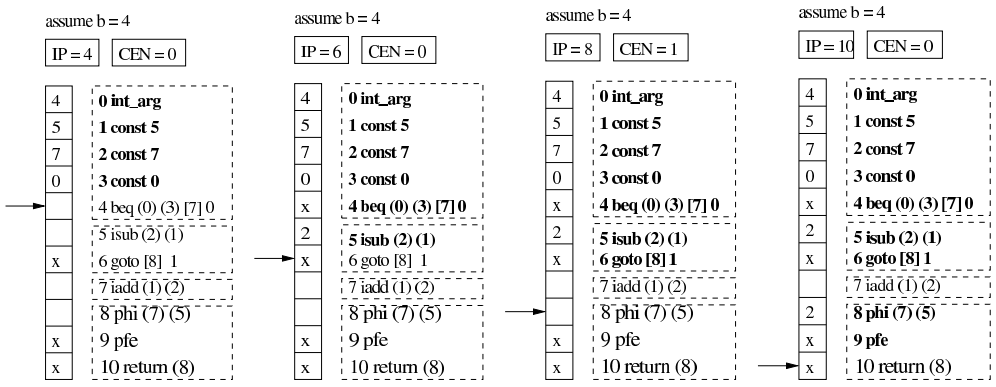
Φ-functions pose a greater obstacle to direct imperative interpretation of programs in SSA Form. In standard SSA Form, each ϕ-function resides in a basic block (where more than one control flow edge converges) and selects one of its input operands (using that operand as its result value) on the basis of the control flow edges on which the dynamic execution entered the basic block.

Figure 3 shows a simple program with converging control flow translated into Interpretable SSA. The ϕ-functions (that would exist in standard SSA Form) are replaced by `phi` instructions. Since the basic-block control flow graph (CFG) (which is shown as the dashed boxes and arrows in Figure 3(b)) is not explicitly represented in ISSA, it is not clear how an interpreter should decide whether the `phi` instruction is to copy from OR7 or OR5. For this reason, ISSA provides an auxiliary CFG-Edge Number (CEN) register, which is set on branching instructions and is used by `phi` instructions to select among their operands. (This CEN register serves the same role as the predicate of Gated Single Assignment Form's γ-functions [Ballance et al., 1990].)



(a) source code (b) ISSA code

Figure 3: a program using the if-then-else construct



(a) (b) (c) (d)

Figure 4: execution of if-then-else construct

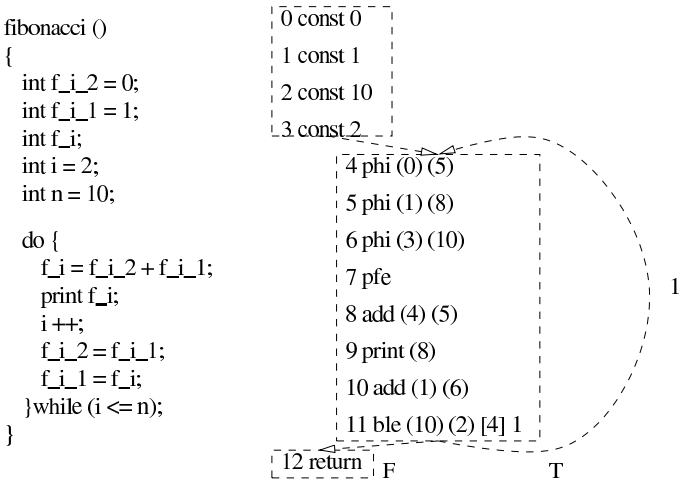


Figure 5: a program for computing the Fibonacci sequence

Figure 4 shows several snapshots of this program's execution. Consider the execution of instruction 6 (transforming the state of Figure 4(b) into that of Figure 4(c)); this corresponds to the traversal of the CFG edge labeled "1" in Figure 3(b). ISSA's `goto` instruction takes two immediate operands, the first is the target instruction number (in this case, 8) and the edge number (in this case, 1). When instruction 6 is executed the CEN register is set to 1 and the control is transfered to instruction 8 (the `phi` instruction). Because the CEN register is 1, the second operand of the `phi` instruction is selected, and 2 is read from OR5 and placed in OR8. After this the CEN register is reset to 0; the resulting state can be seen in Figure 4(d).

2.3 Simultaneous Execution of Φ-functions

The observant reader will have noticed that the discussion in the previous section did not mention the `pfe` (Phi-Function End) instruction, which marks the end of the `phi` instructions within a basic block. The `pfe` instruction is needed, because standard SSA Form ϕ-function semantics require that the ϕ-functions be "executed" at the beginning of the basic block in which they reside [Cytron et al., 1991]. Consequently, when one or more ϕ-function (in a loop) reference the result values of ϕ-functions within the same basic block, they must be implemented, so that the VM behaves as if they were all executed simultaneously using the previous iteration's result values [Morgan, 1998]. For this reason, an ISSA interpreter

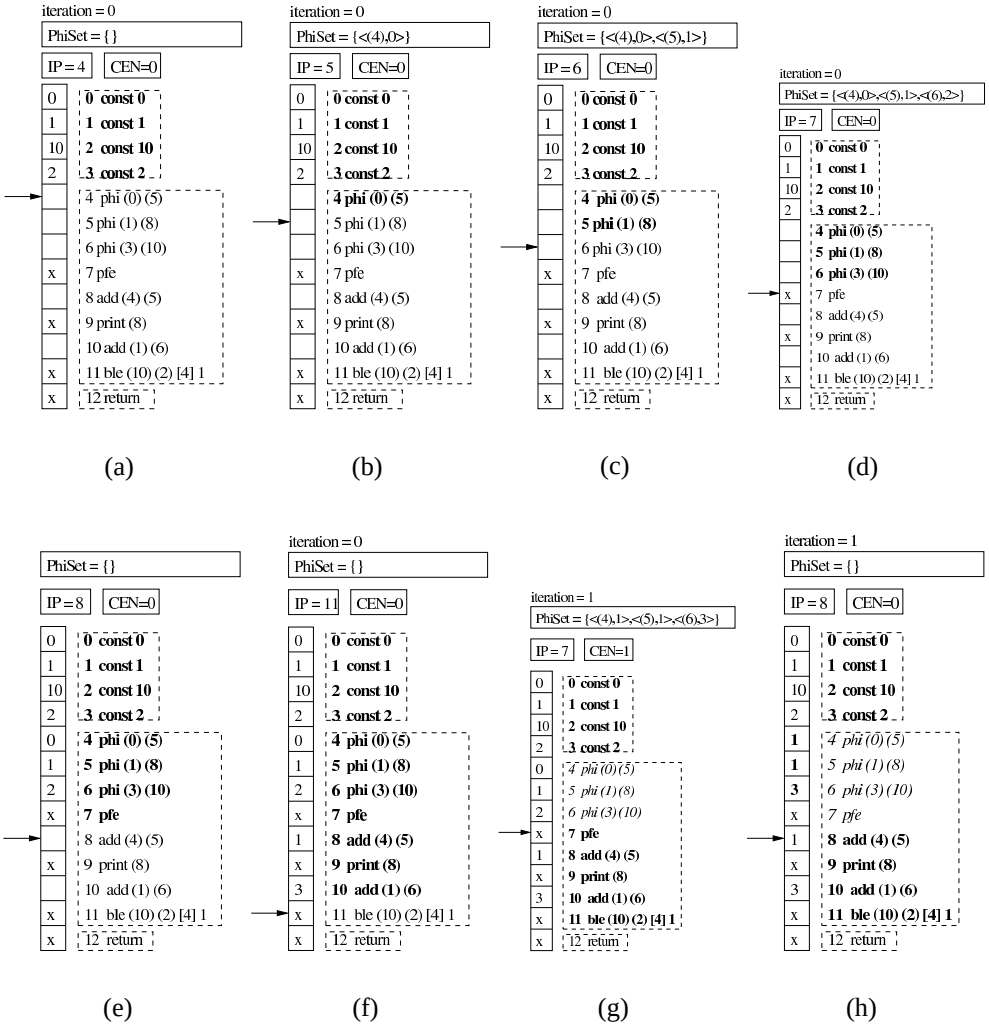


Figure 6: computing the Fibonacci sequence

will buffer `phi` instruction transfers until it executes the `pfe` instruction, which commits the transfers stored in the *PhiSet* buffer and resets the CEN register.

Figure 6 shows the execution of the program shown in Figure 5 which computes the first 10 numbers of the Fibonacci sequence; this program has a ϕ-function (instruction 4) that references the result of a ϕ-function (instruction 5) from the previous iteration. This happens, because the previous iteration's n-1 becomes the new iteration's n-2; in more complicated examples, there could be multiple mutually dependent ϕ-functions. ISSA is able to handle these cases because it performs all of the reads (`phi` instructions) before performing any of the writes (at the `pfe` instruction).

Figure 6(a) show the state of the program on initially entering the loop body; the CEN register is 0, indicating that the first operand of the `phi` instructions should be used. As each `phi` instruction is executed, a pair, containing the `phi`'s instruction number and the selected operands current value, is added to the *PhiSet* buffer. When instruction 4 executes, it selects the first operand, reads in the contents of OR0 (i.e. 0), and places (4, 0) into the *PhiSet* buffer (Figure 6(b)); for 5 it reads in OR1 and places (5,1) into the buffer (Figure 6(c)); for 6 it reads in OR3 and places (6,2) into the buffer (Figure 6(d)). After this, the *pfe* instruction executes; it removes each of the pairs out of the PhiSet (the order does not matter) and writes to the appropriate output registers (0 to OR4, 1 to OR5, and 2 to OR6) and resets the CEN register to 0.

The second iteration is entered from the conditional branch of instruction 11 (Figure 6(f)). Because the value of OR10 (i.e. 3) is less than the value of OR2 (i.e. 10), the test succeeds, control transfers control back to instruction 4, and the CEN register is set to 1. Thus, in this iteration, the second operand of each `phi` instruction is selected, and the VM reads in the current values of OR5, OR8, and OR10 and placing (4,1), (5,1), and (6,3) into the *PhiSet* buffer (Figure 6(g)). When the `pfe` instruction executes, the appropriate OR's are written to and the CEN register is reset; the result is shown in Figure 6(h).

2.4 Arrays

Support for non-scalars has long been problematic in SSA, and many extensions have been proposed for supporting arrays and other non-scalars (e.g., Knobe and Sarkar's 1998 Array SSA Form). ISSA supports array manipulation through `newarray`, `access`, and `update` instructions modeled after the *Update* and *Access* functions of Cytron *et al.* 1991, which treat the entire array as a single SSA variable. Each `newarray` instruction takes as an operand the number of

(a) Source Code	(b) SSA Form	(c) After Code Motion	(d) ISSA Code

a = new int [3];
a[0] = 5;
a[1] = 4;
x = a[1]
a[1] = 3;
y = a[1]
	a1 = newarray 3
	a2 = update (a1, 0, 5)
	a3 = update (a2, 1, 4)
	x0 = access (a3, 1)
	a4 = update (a3, 1, 3)
	y0 = access (a4, 1)
		a1 = newarray 3
		a2 = update (a1, 0, 5)
		a3 = update (a2, 1, 4)
		a4 = update (a3, 1, 3)
		x0 = access (a3, 1)
		y0 = access (a4, 1)
			0 const 5
			1 const 4
			2 const 3
			3 const 2
			4 const 1
			5 const 0
			6 newarray (3)
			7 update (7) (5) (0)
			8 update (8) (4) (1)
			9 update (6) (4) (2)
			10 access (8) (4)
			11 access (9) (4)

Figure 7: a program exhibiting array manipulation

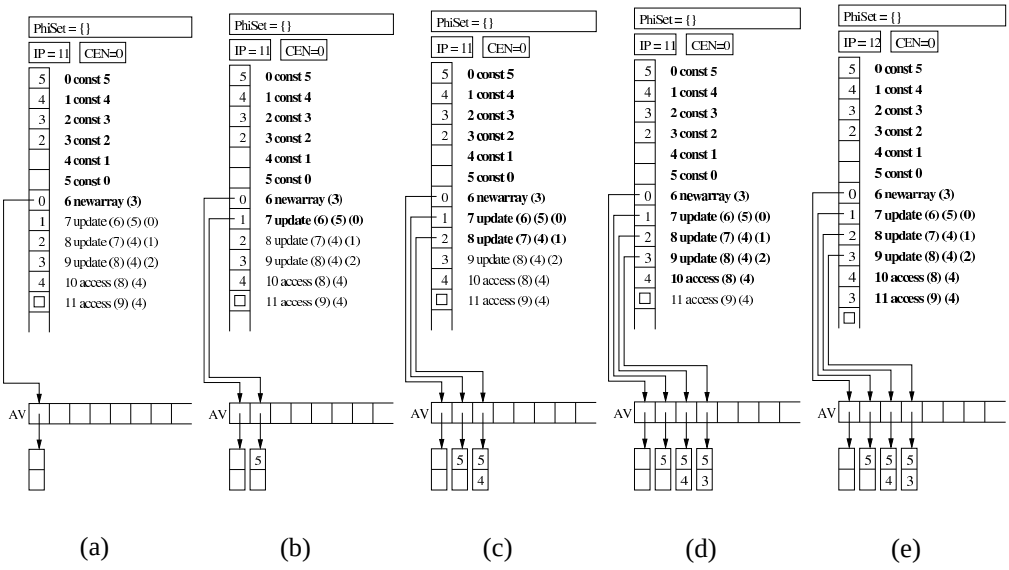


Figure 8: dynamic execution of array manipulation

elements, creates a new array of that size, and places a reference to it in the instruction's OR. Every `access`, takes as operands a reference to an array and the index into the array, fetches the appropriate element from that array, and places a copy of its value in the instruction's OR. Each `update` instruction takes as operands a reference to an array, an index into that array, and a value, copies the array, writes the value to the element in the new array identified by the index, and places a reference to the new array in the `update`'s OR.

Copying the whole array on each update is inefficient but it is the most straightforward way of maintaining proper SSA semantics and avoiding output dependencies. The output dependencies are not a problem if the SSA code was produced by a straight forward translation of a source program. If, however, code motion is performed as part of the program's optimization in SSA Form (for example, if partial redundancy elimination is performed on the external SSA representation for the benefit of JIT compiled code in a mixed-mode compilation environment), the interpreter must be prepared to deal with the possibility of an array access being moved above updates. (Alternatively, the optimizer could be made aware of output dependencies for non-scalars, or output dependencies could be fixed-up by another code motion phase just prior to interpretation.)

An example program requiring non-destructive *update* semantics is shown in Figure 7. Figure 7(b) shows the direct SSA translation of the source program in Figure 7(a). Figure 7(c) which is semantically equivalent to the program in Figure 7(b) but has been altered by legal code motion and as a result accesses an old version of an array (i.e. a3) even after it has been updated (becoming a4). Thus, a3 and a4 have overlapping live ranges and, for this reason, cannot share the same storage space. Figure 7(d) shows the same program in ISSA.

Figure 8 shows this program's step-by-step execution. Array references are implemented as indexes into a dynamic data structure called the *array vector* (AV), which contains pointers to all of the arrays instantiated during program execution. The execution of each `newarray` or `update` adds a new array to the *array vector* for each version of the original program array (Figure 8(a), Figure 8(b), and Figure 8(c)). The `access` instructions select array versions by referencing the OR of the instruction which produced the array version; the `access` of instruction 10 uses the array produced by instruction 8, which was unaffected by the `update` at instruction 9 (Figure 8(d)), so retrieves the "old" value of element 1 (i.e., 4). Instruction 11, however, uses the "current" version of the array produced by the `update` at instruction 9 (Figure 8(e)) and retrieves the the "current" value of element 1 (i.e., 3).

Core Interpreter

File Name	Lines of Code
ssa_vm.h	19
ssa_vm.c	364
Total	383

ISSA Parser

File Name	Lines of Code
inst.h	75
inst.c	140
ssa_lex.l	90
ssa_parser.y	167
Total	472

Misc

File Name	Lines of Code
vm_main.c	49
ssa_array.h	27
ssa_array.c	99
Total	175

Table 1: lines of code for prototype implementation

3 Discussion

3.1 Implementation

We have implemented a simple prototype ISSA Virtual Machine (VM) in C. During execution, it reads and parses an ASCII representation of ISSA code and then executes it using a simple interpreter consisting of a switch statement (with 30 case statements, one for each instruction opcode) embedded in a loop. The virtual machine is untyped; all immediate and register values are 32-bit words but may be used as integers, single-precision floats, or indexes into the *array vector*. Instruction numbers used as operands are also 32-bits. The PhiSet buffer is implemented with an array. Dynamic bounds checking is performed to ensure that neither invalid instruction numbers nor illegal array manipulation violate the virtual machine's integrity.

3.2 Safety

Our prototype virtual machine guarantees VM integrity by the extensive use of dynamic bounds checks (which shut down the VM on violations) but does not check other safety properties, whose violation can only effect the programs correctness. In particular, CEN values on branches, `phi` instructions, and `pfe` instructions, must be used correctly in order to implement standard SSA semantics, misuse may produce programs that are not in SSA form. Neither does the VM distinguish float, integer, and array reference types; instead all data exists as 32-bit words and each instruction uses those words as the types that are appropriate for that instruction; checking type safety is left to the generator of ISSA as a matter of program correctness rather than VM integrity.

Computation	Elapsed Time		Relative	Description
Factorial (C)	10.0	msec		$8!$, 50000 times
Factorial (SSA)	972.25	msec	97x	$8!$, 50000 times
Fibbonacci Sequence Scalar (C)	67.5	msec		first 100, 50000 times
Fibbonacci Sequence Scalar (SSA)	3800.0	msec	$56\times$	first 100, 50000 times
Fibbonacci Sequence Array (C)	4.25	msec		first 100, 1000 times
Fibbonacci Sequence Array (SSA/destructive)	137.5	msec	$32\times$	first 100, 1000 times
Fibbonacci Sequence Array (SSA/non-dest.)	408.75	msec	$96\times$	first 100, 1000 times

Table 2: execution times

3.3 Performance

Although our interpreter was designed for simplicity over performance, we measured its performance on a few simple programs against that of optimized C code; these results are shown in Table 2. There were three benchmarks: one computes the first 100 numbers of the Fibonacci sequence building placing each element into an array sequence; another computes the Fibonacci sequence in scalar variables; the last calculates $8!$ using scalar variables. These benchmarks were executed on an IBM Thinkpad X23 866MHz PIII-M running Windows 2000; all I/O was suppressed from both SSA and C programs, and the interpreter's file parsing time was excluded. Relative performance of the interpreter was one to two orders of magnitude slower than optimized C code (compiled with `g++ -O3`).

3.4 Possible Improvements

Our prototype implementation favors readability and simplicity over performance. A reimplementation prioritizing performance would probably result in a significant improvement based only on low-level optimizations. These, however, are less interesting than several algorithmic level optimizations that are worthy of investigation.

As noted above, each array `update` results in a copy. Most of the time, these are unnecessary. A live range analysis could be used to identify cases where the `update`'s input array is never used again. (For most programs, this would be all of them.) In those cases, the `update` can safely avoid the copy and instead destructively modify the array in place and output the a reference to that same array. In order to approximate the performance gain of this optimization, we unconditionally disabled array copying and ran our Fibonacci array benchmark (which has 100 element arrays); this resulted in a speedup of approximately $3\times$ (the Destructive vs. Non-Destructive variants in Figure 2).

It would be possible to create a "safer" interpreter by adding a pre-pass phase that starts with a type safe and referentially secure SSA representation [Amme et al., 2001] and then adds the correct CEN operands to branches and `pfe` instructions at the correct locations. As a side effect, enforced type safety allows the *array vector* to be safely dispensed with, because the array references could be statically verified and implemented with direct pointers.

Another optimization we have considered is performing a "pseudo register allocation", which would add explicit output locations to instruction and reuse output registers when their live ranges do not overlap. Conceivable, this might reduce the cache footprint resulting in better performance. Unfortunately, this is unlikely occur without other changes being made to the our prototype VM, which assumes everything is a 32-bit word. If this is not amended, adding a (32-bit) output register identifier to every outputting instruction is likely to make the cache footprint worse rather than better.

4 Conclusion

It is indeed feasible to interpret Static Single Assignment Form. Programs in standard SSA Form may be translated into Interpretable SSA (ISSA) Form by renaming operands to implicit registers, annotating edge numbers at branches, and marking the last ϕ-function in each converging basic block. An ISSA interpreter for scalars needs only an *output register* for each instruction, a *control-flow edge number* register to select `phi` instruction operands, and a *PhiSet* buffer to simultaneously commit `phi` instruction result values. Standard SSA Form array *Access* and *Update* functions can be implemented in a straight forward manner.

A prototype ISSA interpreter has been constructed. This demonstrates the feasibility of a mixed-mode (interpreting and compiling) virtual machine using only representations in Static Single Assignment Form.

References

Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–11. ACM Press, 1988. ISBN 0-89791-252-7.

Wolfram Amme, Niall Dalton, Jeffery von Ronne, and Michael Franz. SafeTSA: a type safe and referentially secure mobile-code representation based on static single assignment form. In *Proceedings of the ACM SIGPLAN'01 conference on Programming language design and implementation*, pages 137–147. ACM Press, 2001. ISBN 1-58113-414-2.

C. Scott Ananian. The static single information form. Master's thesis, Massachusetts Institute of Technology, 1999. URL `http://www.cag.lcs.mit.edu/~cananian/Publications/`.

Robert A. Ballance, Arthur B. Maccabe, and Karl J. Ottenstein. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proceedings of the conference on Programming language design and implementation*, pages 257–271. ACM Press, 1990. ISBN 0-89791-364-7.

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991. ISSN 0164-0925.

Kathleen Knobe and Vivek Sarkar. Array SSA form and its use in parallelization. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 107–120. ACM Press, 1998. ISBN 0-89791-979-3.

Chandra Krintz. Improving mobile program performance through the use of a hybrid intermediate representation. In *2nd Workshop on Intermediate Representation Engineering for Virtual Machines*, June 2002. URL `http://www.cs.ucsb.edu/~ckrintz/abstracts/hybrid.html`.

Robert Morgan. *Building an Optimizing Compiler*. Butterworth-Heinemann, Woburn, Massachusetts, United States, 1998. ISBN 155558179X.

Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–27. ACM Press, 1988. ISBN 0-89791-252-7.

A Implementation of the Interpreter Core

A.1 ssa_vm.c

```
#include "inst.h"
#include "ssa_vm.h"
#include "ssa_array.h"
#include "ssa_parser.h"
#include <stdlib.h>
#include <stdio.h>

typedef struct phi_assignment phi_assignment;

struct phi_assignment
{
  ssa_variable v;          // v = value to be transfered
  int ovi;                 // ovi = output variable index
};

typedef struct
{
  inst **ia;               // ia = instruction array
  int ial;                 // ial = instruction array length
  ssa_array_vector *av;    // av = array vector
  ssa_variable *oa;        // oa = output array (size = inst_length)
  phi_assignment *pq;      // pq = phi-assignment queue
  int pqt;                 // pqt = phi-assignment queue top
  int ip;                  // ip = instruction pointer (index to inst_array)
  int cen;                 // cen = CFG edge number
} vm_state;


void init(vm_state *s, inst* inst_array [], int inst_length);
void commit_phis(vm_state *s);
void print_state(vm_state *s);
int execute(vm_state *s, inst * instruction);

inline int pq_empty(vm_state s) {return s.pq == NULL;}

inline ssa_variable decode_immediate(inst * instruction, int i) {
```

```
    ssa_variable v = ((ssa_variable *) instruction->data)[i];
    // printf("decoded immediate %d\n", v.i);
    return v;
}

inline ssa_variable decode_operand(vm_state *s, inst * instruction, int i) {
    int index = ((int*) instruction->data)[i];
    // printf("decoded operand refering to output of instruction %d\n", index);
    // printf("which is %d\n", s->oa[index].i);
    return s->oa[index];
}

inline ssa_variable first_operand(vm_state *s, inst * instruction) {
    return decode_operand(s, instruction, 0);
}

inline ssa_variable second_operand(vm_state *s, inst * instruction) {
    return decode_operand(s, instruction, 1);
}

void ssa_vm(inst * inst_array[], int inst_length)
{
    vm_state s;
    inst * ci;

    init(&s, inst_array, inst_length);
    while (1)
    {
        if (s.ip < 0 || s.ip >= s.ial) abort(); // we jumped out of the method
        // print_state(&s);
        ci = s.ia[s.ip];
        if (execute(&s, ci) == 0) break;
    }
}


void init(vm_state *s, inst * inst_array[], int inst_length)
{
    // instructions -- should we copy these?
    s->ia = inst_array;
```

```
  s->ial = inst_length ;
  // simple registers ;
  s->ip = 0;
  s->cen = 0;
  // complex data
  s->av = av_init ();
  s->oa = (ssa_variable *) calloc ( inst_length , sizeof ( ssa_variable ));
  s->pq = (phi_assignment*) calloc ( inst_length , sizeof (phi_assignment ));
  s->pqt = 0;
}

void print_state (vm_state *s) {
  int i;
  int ial = 26; // should be s->ial

  // Dump the current Output Array
  printf ("OA: ");
  for ( i = 0; i < ial ; i++)
    printf ("%d ", s->oa[i].i );
  printf ("\n");

  // Dump Registers
  printf ("IP=%d CEN=%d ", s->ip, s->cen);

  // Dump Phi Queue
  printf (" PQ={");
  for ( i = 0; i < s->pqt; i++) {
    printf ("(%u<-%d), ", s->pq[i].ovi, s->pq[i].v.i );
  }
  printf ("}");

  // Array Vector?
  // Current Instruction ?
  printf (" instruction [%d]: ", s->ip);
  if ( s->ia[s->ip] == NULL)
    printf ("NULL");
  else
    print_inst (s->ia[s->ip]);
}
```

```
void commit_phis(vm_state *s)
{
  int i;

  for (i = 0; i < s->pqt; i++) {
    s->oa[s->pq[i].ovi] = s->pq[i].v;
  }

  s->pqt = 0;

}

int execute(vm_state *s, inst *ci)
{
  // temporaries
  int t,n,a,i;
  ssa_variable x,y;

  switch (ci->opcode)
    {
    case CONST:
      // puts("decoding constant");
      x = decode_immediate (ci,0);
      // printf ("decoded immediate %d\n", x.i);
      s->oa[s->ip] = x;
      s->ip++;
      break;
    case PRINT:
      // puts(" printing int");
      x = first_operand (s,ci);
      printf ("==> %d\n", x.i);
      s->ip++;
      break;
    case ADD:
      x = first_operand (s,ci);
      y = second_operand(s,ci);
      s->oa[s->ip].i = x.i + y.i;
      s->ip++;
      break;
    case SUB:
```

```
    x = first_operand (s, ci);
    y = second_operand(s, ci);
    s->oa[s->ip].i = x.i - y.i;
    s->ip++;
    break;
  case DIV:
    x = first_operand (s, ci);
    y = second_operand(s, ci);
    s->oa[s->ip].i = x.i / y.i;
    s->ip++;
    break;
  case MUL:
    x = first_operand (s, ci);
    y = second_operand(s, ci);
    s->oa[s->ip].i = x.i * y.i;
    s->ip++;
    break;
  case AND:
    x = first_operand (s, ci);
    y = second_operand(s, ci);
    s->oa[s->ip].i = x.i && y.i;
    s->ip++;
    break;
  case OR:
    x = first_operand (s, ci);
    y = second_operand(s, ci);
    s->oa[s->ip].i = x.i || y.i;
    s->ip++;
    break;
  case NEG:
    x = first_operand (s, ci);
    s->oa[s->ip].i = -x.i;
    s->ip++;
    break;
  case BGE:
    x = first_operand (s, ci);
    y = second_operand(s, ci);
    t = decode_immediate(ci,2). t;          // branch target
    n = decode_immediate(ci,3). n;          // CFG Edge Number
    if (x.i >= y.i)
```

```
    {
      s->cen = n;
      s->ip = t;
    } else {
      s->ip++;
    }
  break;
case BGT:
  x = first_operand (s, ci);
  y = second_operand(s, ci);
  t = decode_immediate(ci,2). t;          // branch target
  n = decode_immediate(ci,3). n;          // CFG Edge Number
  if (x.i > y.i)
    {
      s->cen = n;
      s->ip = t;
    } else {
      s->ip++;
    }
  break;
case BLE:
  x = first_operand (s, ci);
  y = second_operand(s, ci);
  t = decode_immediate(ci,2). t;          // branch target
  n = decode_immediate(ci,3). n;          // CFG Edge Number
  if (x.i <= y.i)
    {
      s->cen = n;
      s->ip = t;
    } else {
      s->ip++;
    }
  break;
case BLT:
  x = first_operand (s, ci);
  y = second_operand(s, ci);
  t = decode_immediate(ci,2). t;          // branch target
  n = decode_immediate(ci,3). n;          // CFG Edge Number
  if (x.i < y.i)
    {
```

```
        s->cen = n;
        s->ip = t;
      } else {
        s->ip++;
      }
    break;
case BNE:
    x = first_operand (s, ci);
    y = second_operand(s, ci);
    t = decode_immediate(ci ,2). t ;          // branch target
    n = decode_immediate(ci ,3). n ;          // CFG Edge Number
    if (x.i != y.i)
      {
        s->cen = n;
        s->ip = t;
      } else {
        s->ip++;
      }
    break;
case BEQ:
    x = first_operand (s, ci);
    y = second_operand(s, ci);
    t = decode_immediate(ci ,2). t ;          // branch target
    n = decode_immediate(ci ,3). n ;          // CFG Edge Number
    if (x.i == y.i)
      {
        s->cen = n;
        s->ip = t;
      } else {
        s->ip++;
      }
    break;
case GOTO:
    t = decode_immediate(ci ,0). t ;          // branch target
    n = decode_immediate(ci ,1). n ;          // CFG Edge Number
    s->cen = n;
    s->ip = t;
    break;
case EXIT:
    puts(" exiting ");
```

```
    // exit (0);    // in order to repeat the execution many times
    return 0;
case RETURN:
    x = decode_operand(s, ci ,0);    // thing to set the element to
    printf ("return %d\n", x.i ); //  in order to repeat the execution many
                                     // times
    return 0;
    // exit (x.i );
case PHI:
    // check that the PHI is big enough for the cfg edge number
    n = ci->opdnum;
    // printf ("checking if %d >= %d ..... (then abort)\n", s->cen, n);

    if (s->cen >= n) abort();          // phi must have enough operands
    if (s->pqt >= s->ial) abort (); //  can't overflow phi queue buffer

    // record the data in the phi-assignment queue
    s->pq[s->pqt].ovi = s->ip;
    s->pq[s->pqt].v = decode_operand(s, ci , s->cen);;
    s->pqt++;

    // next instruction
    s->ip++;
    break;
case PFE:
    commit_phis(s);
    s->cen = 0;
    s->ip++;
    break;
case NOOP:
    s->ip++;
    break;
case NEWARRAY:
    x = decode_operand(s, ci ,0);    // thing to set the element to
    s->oa[s->ip].a = av_newarray(s->av, x.i);
    s->ip++;
    break;
case UPDATE:
    a = decode_operand(s, ci ,0). a;    // array (index of array in array vector)
    i = decode_operand(s, ci ,1). i;    // element (index into array)
```

```
    x = decode_operand(s, ci ,2);       // thing to set the element to
    s->oa[s->ip].a = av_update(s->av, a, i , x );    // result is new array
    s->ip++;
    break;
  case ACCESS:
    a = decode_operand(s, ci ,0). a ;    // array (index of array in array vector)
    i = decode_operand(s, ci ,1). i ;    // element (index into array)
    s->oa[s->ip] = av_access(s->av, a, i );    // result is element
    s->ip++;
    break;
  case FADD:
    x = first_operand (s, ci );
    y = second_operand(s, ci );
    s->oa[s->ip].f = x.f + y.f;
    s->ip++;
    break;
  case FSUB:
    x = first_operand (s, ci );
    y = second_operand(s, ci );
    s->oa[s->ip].f = x.f - y.f;
    s->ip++;
    break;
  case FDIV:
    x = first_operand (s, ci );
    y = second_operand(s, ci );
    s->oa[s->ip].f = x.f / y.f;
    s->ip++;
    break;
  case FMUL:
    x = first_operand (s, ci );
    y = second_operand(s, ci );
    s->oa[s->ip].f = x.f * y.f;
    s->ip++;
    break;
  case FCONST:
    s->oa[s->ip] = decode_immediate (ci,0);
    s->ip++;
    break;
  case FPRINT:
    x = first_operand (s, ci );
```

```
        printf ("==>␣%f\n", x.f);
        s->ip++;
        break;
      default:
        abort ();
      }
  return 1;
}
```

A.2 ssa_vm.h

```
#ifndef SSA_VM_H
#define SSA_VM_H

#include "inst.h"

typedef signed s32;
typedef unsigned u32;
typedef float f32;

typedef union ssa_variable {
    s32 i;    // 32-bit signed integer integer
    f32 f;    // 32-bit floating point value
    u32 a;    // 32-bit unsigned array vector index
    u32 n;    // 32-bit unsigned array CFG Edge Number
    u32 t;    // 32-bit unsigned branch target
} ssa_variable;

void ssa_vm(inst* inst_array [], int inst_length);
#endif
```

A.3 ssa_array.h

```
#ifndef SSA_ARRAY_H
#define SSA_ARRAY_H

#include "ssa_vm.h"
#include <stdlib.h>

typedef struct
{
```

```
  unsigned l;          // array length
   ssa_variable *a;    // array of ssa_variables
} ssa_array;

typedef struct
{
  unsigned na;    // next array
  unsigned l;     // allocated length
   ssa_array *v; // array (vector) of arrays
} ssa_array_vector;

 ssa_array_vector * av_init ();
u32 av_newarray (ssa_array_vector *av, int size);
void av_cleanup (ssa_array_vector *av);
u32 av_update (ssa_array_vector *av, u32 av_index, u32 a_index, ssa_variable v);
 ssa_variable av_access (ssa_array_vector *av, u32 av_index, u32 array_index);
u32 av_fastupdate (ssa_array_vector *av, u32 av_index, u32 array_index, ssa_variable v);

#endif
```

A.4 inst.h

```
#ifndef INST_H
#define INST_H

#include <stdlib.h>

#define MAX_INSTS 1024

#define BASE 257 /* the first 256 is assigned to ascii char */

typedef struct
{
  int      opcode;
  int      opdnum; /* length in words */
  char     * img;
  void     * data;
  int      data_type; /* int, bool, float */
} inst;
```

```
typedef  struct
{
   char * img;
   int  opdnum;
}  inst_attribute  ;

 static    inst_attribute      inst_att   [] = {
   {"const",1},
   {"fconst" , 1},
   {"add"   ,2},
   {"sub"   ,2},
   {"div"   ,2},
   {"mul"   ,2},
   {"and"   ,2},
   {"or"    ,2},
   {"neg"   ,1},
   {"fadd" ,    2},
   {"fsub" ,    2},
   {"fdiv" ,    2},
   {"fmul",     2},
   {"bge"   ,4},
   {"bgt"   ,4},
   {"ble"   ,4},
   {"blt"   ,4},
   {"bne"   ,4},
   {"beq"   ,4},
   {"goto"  ,2},
   {"phi"   ,-1},
   {"pfe"   ,0},
   {"update" , 3},
   {"access" , 2},
   {"newarray" , 1},
   {"exit" , 0},
   {"return" , 1},
   {"print" , 1},
   {" fprint " , 1},
   {"null"  ,0},
};
```

```
extern inst * insts_array [];
extern int insts_size ;

/* inst.cc */
inst *new_inst(int opcode);
inst *new_unary_inst(int opcode, int opd);
inst *new_unary_finst(int opcode, float opd);
inst *new_binary_inst(int opcode, int opd1, int opd2);
inst *new_tenary_inst(int opcode, int opd1, int opd2, int opd3);
inst *new_quandary_inst(int opcode, int opd1, int opd2, int opd3, int opd4);
inst *new_phi_inst(int opcode, int opdnum, int opd[]);
void print_all_insts ( inst * insts [], int size );
void print_inst ( inst * ist );
void delete_all_insts ( inst *insts [], int size );

#endif /* no INST_H */
```

B Benchmarks

B.1 Fibbonacci Sequence (in scalars)

B.1.1 fibonacci.ssa

```
//
// F(0) = 0
// F(1) = 1
// F(n) = F(n−2) + F(n−1) for all n >= 2
//
//
// n = 2
// if (n >= 2)
//   goto exit;
//
// while (n <= max)
//   {
//     F(n) = F(n−2) + F(n−1);
//     print F(n);
//     n = n + 1;
//     F(n−2) = F(n−1);
//     F(n−1) = F(n);
//   }
//  exit (0)
//
//

// Block 0
0 const_0          //
1 const_1          //
2 const_100        // max = 16
3 const_2          // n = 2
// const_2         // min = 2
4 bge    (3 3) [6] 0
5 goto [17]   0

// Block 1
6 phi_2 (0   7)          //  phi (f_0, f_{n−2})
7 phi_2 (1   10)         //  phi (f_1, f_{n−1})
```

```
8 phi_2 (3   12)          //  phi (n,    n+1)
9 pfe
10 add    (6   7)         //  f_{n} = f_{n-2} + f_{n-1}
11 print  10              //  print  f_{n}
12 add    (1   8)         //  n <- n+1
13 add    (0   7)         //  f_{n-2} <- f_{n-1}
14 add    (0   10)        //  f_{n-1} <- f_n
15 ble     (12 2) [6] 1   //  n <= max goto block 1
16 goto   [17] 1

// Block 2
17 exit
```

B.1.2 fibonacci.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void fib ();
int main (int argc, char ** argv)
{
   clock_t start, end, used;
   int i = 0;

   used = 0;
   for (i = 0; i < 1000; ++i){
      start = clock ();
      fib ();
      end = clock ();
      used += end - start;
   }
   fprintf (stderr, " Time used %d\n", used);
}


void fib ()
{
   int f_n_2 = 0, f_n_1 = 1;
   int n = 2, max = 100;
   int f_n;
```

```
   if  ( n < 2)
      return  ;

   while  ( n <= max)
      {
        f_n  =  f_n_2  +  f_n_1 ;
         printf  (”==> %d\n”, f_n );
        n  =  n  + 1;
        f_n_2  =  f_n_1 ;
        f_n_1  =  f_n ;
      }
}
```

B.2 Fibbonacci Sequence (in an array)

B.2.1 fibonacci_array.ssa

```
//   F = newarray (n)
//   F[0] = 0
//   F[1] = 1
//
//   i = 2
//   while (i < n)
//      {
//        F[i] = F[i−1] + F[i−2]
//        i ++
//      }
//
//   i = 0;
//   while (i<n)
//       print F[i];
//
0 const_0        // 0
1 const_1        // 1
2 const_2        // i = 2
3 const_100      // n = 10

4 newarray 3                    // f = newarray (10)
5 update   (4 0) 0              // f (0) = 0
6 update   (5 1) 1              // f (1) = 1

7 blt      (2 3) [9] 0
8 goto  [20]      1

//
9    phi_2    (2 18)            // phi (2, i+1)
10   phi_2    (6 17)            //
11   pfe
12   sub      (9 1)             // j = i − 1
13   sub      (9 2)             // k = i − 2
14   access   (10 12)           // F[i−1]
15   access   (10 13)           // F[i−2]
16   add      (14 15)           // F[i−1] + F[i−2]
```

```
17   update   (10 9) 16              //  F[i] <- 15
18   add      (1 9)
19   blt      (18 3) [9] 1


20   phi_3    (0 0 25)
21   phi_3    (17 6 21)
22   pfe
23   access   (21 20)
24   print    23
25   add      (20 1)
26   blt      (25 3) [20] 2
27   exit
```

B.2.2 fib_array.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void fib ();
int main (int argc, char ** argv)
{
   clock_t start, end, used;
   int i = 0;

  used = 0;
   for (i = 0; i < 1000; ++i){
      start = clock ();
      fib ();
     end = clock ();
     used += end - start;
   }
    fprintf (stderr, " Time used %d\n", used);
}


void fib ()
{
   int f [100];
   int n = 2, max = 100;
```

```
    f [0] = 0;
    f [1] = 1;


    if  ( n < 2)
        return  ;

    while  ( n < max)
        {
          f[ n ] =  f[ n−2] + f[ n−1];
           printf  (”==> %d\n”, f[ n ]);
          n  =  n  + 1;
          f[ n−2] = f[ n−1];
          f[ n−1] = f[ n ];
        }
}
```

B.3 Factorials

B.3.1 factorial.ssa

```
//     f = 1
//     n = 10
//     while n > 0
//       {
//           f := f * n;
//           n := n - 1;
//       }
//     print f;
//
//
//      block 0
//        /\
//       /  \ +-----+
//      /    \|     |
//     /    block 1  |
//     \     /\     |
//      \   /  +----+
//       \ /
//     block 2
//
//
//
//
```

```
// Block 0
0 const_10                  // n_1 = 10
1 const_1                   // f_1 = 1
2 const_0                   // zero

3  ble   (0 3) [12] 0       // n <= 0 goto Block 2
4  goto  [5] 0

// Block 1
5  phi_2  (0 9)             // n_2 = phi (n_1, n_3)
6  phi_2  (1 8)             // f_2 = phi (f_1, f_3)
```

```
7  pfe
8  mul     (6 5)              // f_3 = f_2 * n_2
9  sub     (5 1)              // n_3 = n_2 - 1
10 bgt     (9 2) [5] 1        // n_3 > 0 goto Block 1
11 goto    [12] 1

// Block 2
12 phi_2   (1 8)              // f_4 = phi ( f_1 , f_3 )
13 pfe
14 print   12
15 exit
```

B.3.2 factorial.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void fact ();
int main ( int argc , char ** argv)
{
   clock_t start , end, used;
   int i = 0;

   used = 0;
   for ( i = 0; i < 1000; ++i){
      start = clock ();
      fact ();
      end = clock ();
      used += end - start ;
   }
   fprintf ( stderr , " Time used %d\n", used);
}


void fact ()
{
   int f = 1, n = 10;

   while (n > 0)
     {
```

```
        f = f * n;
        n = n - 1;
        printf ("==> %d\n", f);
    }
}
```