SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 21 (8), 877–889 (AUGUST 1991)

Compiled Instruction Set Simulation

CHRISTOPHER MILLS, STANLEY C. AHALT AND JIM FOWLER
Department of Electrical Engineering, The Ohio State University, 2015 Neil Avenue, Columbus, OH 43210, U.S.A.

SUMMARY

An efficient method for simulating instruction sets is described. The method allows for compiled instruction set simulation using the macro expansion capabilities found in many languages. Additionally, we show how the semantics of the C case statement allows instruction branching to be incorporated in an efficient manner. The method is compared with conventional interpreted techniques and is shown to offer considerable performance benefits.

KEY WORDS Interpreters Simulation Emulation Instruction set design

INTRODUCTION

There are a number of reasons why instruction set simulation can be one of the most important steps in the development of an efficient computer architecture. First, simulation requires that exact specifications of the instructions be developed. Secondly, if a compiler is available for the instruction set, real programs can be tested on the architecture. This also means that the compiler can be debugged before the hardware is constructed, thus, simplifying the debugging of each portion of the system.[1] Thirdly, profiling instruction frequency allows the instruction set designer to determine the effectiveness of individual instructions. For example, RISC architectures[2-5] emphasize the inclusion of only the most frequently used operations in the instruction set. Fourthly, through extensive simulations of synthetic and real programs the computer architect can gain insight into the expected performance of the implemented architecture.[6] Finally, careful matching of the compiler and the computer architecture can yield a particularly elegant computing *system.*[7, 8]

Instruction-set simulation is a well-established technique. Classical descriptions of simulation methods are discussed by Gries[9] and by Calingaert.[10] More recently, May has described efforts to build fast, efficient simulators by grouping source instructions and translating them as a unit.[11] An excellent overview by Dasgupta[12] discusses the use of various simulation techniques.

Unfortunately, regardless of the simulation technique used, simulating an architecture can be very time consuming. Most simulators are based on instruction interpreters which operate at the register transfer level. The simulator is usually written in either a hardware description language (HDL)[13,14] or using a standard language such as C. A simple benchmark which takes a minute to run on a computer may take many hours to simulate. At best this reduces both the number and the

0038–0644/91/080877–14$07.00
Received 13 September 1990
Revised 12 March 1991

size of the programs the designer can test, and may result in an architecture which is good at running only a few benchmarks and not actual programs. At worst, an under-tested architecture can have architectural errors which are not discovered until later in the design phase.[1]

Thus, it is to the designer's advantage to have the simulator operate as efficiently as possible. This paper discusses a simple method to reduce the execution time of instruction set simulators greatly. Although the method introduced here can be of significant utility, interpreted simulation techniques have alternative strengths, particularly when used for the detailed analysis of data and instruction path utilization.

INTERPRETED SIMULATION

Instruction set simulation is normally done interpretively. The program is stored in the simulator's memory in the same way as it would be in the simulated computer. The simulator program repeatedly fetches instructions from memory, using the opcode to select a routine to execute that will simulate the effects of that opcode, as well as maintain statistics, e.g. instruction frequency, execution time, etc.

For example, suppose we were investigating the frequency of use of the instructions in a stack machine. One of the instructions is an 'add top of stack to accumulator' command, which can be represented symbolically as

$$AC \leftarrow AC + [SP];\ SP \leftarrow SP+1;\ PC \leftarrow PC+1;$$

We could simulate this with the following C code:

```
switch (*PC++ ) {
        case ADD:
                   ++ freq[ADD]; AC += *SP++; break;

}
```

The simulator will fetch the ADD instruction at the program counter (PC), increment it and branch to the appropriate case. It will pop the top of the stack and add it to the accumulator, and increment the counter that keeps track of the number of times ADD has been executed.

COMPILED SIMULATION

Simulating a computer architecture is similar to implementing a programming language. Both the programming language and the assembly language of the simulated computer represent theoretical machines which are being mapped onto real hardware. There are two approaches to this mapping: interpretation and compilation.

Compilation has distinct efficiency advantages over interpretation; an interpreter spends the majority of its time fetching and decoding the operations, whereas a compiled simulator spends most of its time in performing the computation. In an interpreted programming language, the overhead of interpretation steps can be reasonably small, because the individual instructions being interpreted perform a lot of work, but in an instruction set simulation the interpretation steps consume much of the time spent to simulate an instruction. Of course, the relative cost of interpretation as opposed to computation depends on the complexity of the instruction, addressing modes, etc. Typically, however, simulators based on interpretation techniques execute from ten to a hundred instructions for each interpreted instruction. [11]

Using C macros

The C compiler, along with other languages, provides an excellent, and well-known, facility for 'compiling' simulated assembly language into the host machine language—in-line macro expansion. All the experimenter needs to do is create a macro for each instruction to be emulated, and code the assembly language as a C function. Using the previous example, ADD would be defined as

```
#define ADD      ++ freq[ADD]; AC += *SP++;
```

When ADD is used in line, the C preprocessor will expand it into the two C statements, which in turn will be compiled directly into machine language on the host machine. Since the machine register AC can be declared to be a register variable in C, the result is usually a one-for-two mapping between the simulated assembly language and the host machine language—one instruction to do the simulation, and one instruction to do the instruction frequency accounting. For example, on the Sun C compiler, based on pee, ADD generates the instructions

```
addql    #1 ,_freq+0x4
addl     a4@+, d7
```

The only problem with using this approach is branching. Although we could use labels and goto statements to accomplish simple branching, indirect jumps, including 'return from subroutine' instructions cannot be directly implemented with a goto.

Luckily, the C switch statement can be used to implement branching. Each macro is given a case label with the address of the instruction, and the simulated program is enclosed in a switch statement based on the program counter. If a branch instruction is encountered, the macro sets the program counter to the destination of the branch and does a break, transferring control back to the enclosing switch statement, which then branches to the correct instruction.

Since the case statement in C acts only as a label, unless there is a break statement the flow of control will fall through to the next case statement, which is the next simulated instruction. This means that only instructions which cause a branch to occur have to modify the program counter and break out of the switch statement. All the other operations simply fall through to the next case. The program counter is *not* advanced, but simply reloaded when the next branch occurs. This allows zero overhead for all non-branching instructions.

We now define our ADD and BAZ ('branch on accumulator zero') instructions as

```
#define ADD(a)       case (a):     ++ freq[ADD]; AC += *SP++;
#define BAZ(a, d)    case (a):     +freq[BAZ]; if (AC = = 0) \
                                   { PC = (d); break; }
```

and use them like this:

```
while (TRUE)
    switch (PC) {
    :
    ADD(0x7402);
    BAZ(0X7404, 0x7402);

    }
```

which means there is an ADD opcode at address 7402_{16} and a BAZ instruction at 7404_{16} which branches back to the ADD.

Since the switch construct will usually be compiled by the C compiler into an efficient jump table, the overhead of a branch is only the overhead of an indexed indirect jump.

The switch construct also has the added benefit of trapping out illegal instruction references. An ill-formed branch cannot jump into the middle of an instruction or into a data area because there is no corresponding case label for that address. Finally, one should note that the case labels do not need to correspond to the actual instruction addresses. As long as the value in the 'program counter' is the same as the case label, the branch will function. It is therefore possible to construct an assembler which would take advantage of this feature, and just output arbitrary integers for each label it encounters. The labels would not even have to be in numerical order, since C does not place any restrictions on the ordering of case labels.

ADDRESSING MODES AND FLAGS

Complex addressing modes and flags present other problems. If instructions have many possible addressing modes, it is best to have a set of macros to calculate the effective address and another set of macros to implement the operations. Likewise if the architecture is such that the CPU flags are set after arithmetic operations, a group of flag-setting macros can be created.

Implementing addressing modes

Separating the macros that define the address modes and the instruction operations results in instruction definitions that are shorter and clearer. For example, suppose an architecture has addressing modes 'direct' and 'indirect' and the following instructions:

```
ADD a   'add accumulator direct'
          AC ← AC + [a];
ADX d   'add accumulator indexed'
          AC ← AC+[X+d];
XAD     'add X to accumulator'
          AC ← AC + X;
```

```
LDD a 'load accumulator direct'
          AC ← [a];
LDX d 'load accumulator direct'
          AC ← [X + d];
XLD n 'load X immediate'
          X ← n:
```

where X is an index register, d is a displacement, a is an absolute address and n is an immediate value. To implement these instructions as macros, we define the operations 'add' and 'load' as

```
# define ADD_OP(d, s)        (d) += (s)
# define LD_OP(d, s)         (d) = (s)
```

and the addressing modes 'direct' and 'indexed' as

```
# define DIRECT(a)          MEM[a]
# define INDEXED(d)         MEM[X + d]
```

and then the instructions as

```
# define ADD(a, da)       case (a): + +freq[ADD]; ADD_OP(AC, Direct);
# define ADX(a, d)        case (a): + +freq[ADX]; ADD_OP(AC, INDEXED(d));
# define XAD(a)           case (a): + +freq[XAD]; ADD_OP(AC, X);
# define LDD(a, da)       case (a): + +freq[LDD]; LD_OP(AC, Direct);
# define LDX(a, d)        case (a): + +freq[LDX]; LD_OP(AC, INDEXED(d));
# define XLD(a, n)        case (a): + +freq[XLD]; LD_OP(X, n);
```

Implementing flags

Consider an architecture that has a status register whose least-significant bit is the 'zero' flag. The 'zero' flag is set when an arithmetic operation yields a zero result. We can define the status register and 'zero' flag as

```
unsigned long int status = 0;
#define ZERO 0x00000001
```

Then we define the following macros to test, set, and clear a flag in the status register:

```
#define FLAG(f)           (status & (f))
#define SETFLAG(f)        status I= (f)
#define CLRFLAG(f)        status &= –(f) – 1
```

These macros are then used in the implementation of the instruction set macros. For example, we can redefine the ADD instruction so that the 'zero' flag is set if the result is zero and the BAZ ('branch on zero') instruction to take advantage of the 'zero' flag:

```
#define ADD(a)          case (a):   + +freq[ADD]; \
                                    AC += *SP++; \
                                    if (AC = = 0) SETFLAG(ZERO); \
                                    else CLRFLAG(ZERO);
#define BAZ(a,d)        case (a):   + +freq[BAZ]; \
                                    if (FLAG(ZERO)) \
                                    { PC = (d); break; }
```

The FLAG, SETFLAG and CLRFLAG macros can also be used with other flags defined in a similar fashion.

LARGE PROGRAMS AND OTHER CONSIDERATIONS

In general, compilers will have some limit on the size of switch statements. This limit, albeit a limitation in the compiler, is a restriction on the length of simulation code that can be placed in a switch construct and will vary between compilers.

For example, the cc and gcc compilers used on Sun 3 workstations were investigated to determine their switch size limitations. These programs compile the switch construct into similar indexed indirect jumps using jump tables. For correct compilation of the switch, the object code generated from the body of the switch statement cannot exceed the range of this indexed indirect jump. Thus the limit on the switch is not a limit on the number of case statements a switch can contain; rather, it is a restriction on the overall size of the code generated from *all* the case statements. For these compilers, tested on a Sun 3/80, the maximum size of a compiled switch statement is approximately 32K bytes. Each compiler handles programs exceeding the 32K byte limit differently. The cc compiler generates an error during assembly of the compiled code. On the other hand, the gcc compiler generates no errors during compilation or assembly, but the object code produces a ‘segmentation fault’ error when executed.

Fortunately, the compiled simulation technique can be adapted to handle larger programs of simulation code by using several switch statements instead of one. Specifically, we can use C code similar to the following:

```
while (TRUE) {
    switch (PC)
        {
            /* Start of program */

            PC = 0x7600;
        }
    switch (PC)
        {
            ADD(0x7600);

            PC = 0x7752;
        }

    switch (PC)
```

```
        {
            SUB(0x7a5b);
            :
            HALT(0x7ab2);
        }
    }
```

The simulation code is broken into blocks which do not exceed the switch size limit. Because the PC is not incremented with each instruction, it is necessary, at the end of each switch block, to set the PC to the address of the first instruction of the following switch. It can be verified that branches will work properly, since the switch statements are enclosed within a while loop. However, by using multiple switches we have sacrificed the ability to trap illegal instruction references because we no longer can have a default statement in any of the switch blocks. A default would prevent branching from one switch block to another. Also, it should be noted that partitioning the program into several switch statements introduces a sequential search among the switch blocks on the execution of a branch instruction. The cost of this search will increase with program size, but it will remain insignificant in relation to the execution of the other instructions.

There are other problems which might be encountered with the use of the compiled simulation technique we describe here. For example, complex source instructions might generate code that is not easily optimized by the particular compiler being used. However, interpretive simulators can also be afflicted by this problem and better compilers will yield faster code in either case.

A related concern is that the code generated by the compiled simulator may adversely affect cache and virtual memory performance on the host machine, thus minimizing the speed advantages of the technique. This is due to the fact that the code generated by this technique largely consists of jump tables, and, during execution, the instructions being simulated may well be distributed over a fairly large address space. Again, interpretive simulators can also be afflicted by this problem. In general the principle of locality of reference will hold for programs written using the compiled technique as well as the interpreted technique.

When simulating large programs, the compilation time of the compiled simulator will become significant and may outweigh the savings in execution time. If a very large compiled simulation program is expected to be executed only a few times, then interpreted simulation may be a more efficient process when one considers the cost of compilation of the compiled simulator. However, if this program needs to be run many times, then the increased execution speed of the compiled simulator will probably offset the one-time cost of compilation.

EXAMPLES AND RESULTS

As a complete example, we discuss the performance characteristics of a simple CPU on the Fibonacci benchmark, given below:

```
unsigned main()
{
      unsigned i, value, fib();
```

```
            for(i = 1; i <= 100; ++i)
                value = fib(24);
            return value;
        }
        unsigned fib(n)
            unsigned n;
        {
            if (n > 2)
                return fib(n–1) + fib(n–2);
            else
                return 1;
        }
```

The processor that we will emulate has four registers: A, the accumulator, P, the program counter, S, the stack pointer, and F, the frame pointer of the currently executing procedure. We will implement a subset of the processor's instructions as macros, since we only need 15 instructions to code the benchmark.

The complete program is listed in Appendix I. It can be broken down into three logical segments: the macro definitions for the instruction set, the execute function which contains the assembly code for the Fibonacci benchmark, and the main program which calls execute and prints the execution statistics. The execute function and the macro definitions could be kept in separate files to simplify changes to the benchmark or instruction set. An example of program output is given in Appendix II.

To determine the effects of compiled simulation, the program was compared against a nearly identical version which used interpreted simulation. In addition, several other configurations were tested to determine the performance of the technique when used in a realistic fashion. The cache was implemented both as a function call, and as an external, independent process which communicated via a pipe. The accounting code kept track of the frequency of use of each instruction. The results of the simulations are shown in Table I.

Most importantly, it can be seen from Table I that the compiled simulator is nearly three times faster than the interpreted simulator. Also note that the savings in execution time outweigh the additional cost of compilation time of the compiled simulator.

Note also that although the incorporation of a cache (either functional or piped)

Table I.

Program	Time (s)	Relative speed
Native code	24	1·00
Compiled, no accounting †	82	3·42
Compiled, no cache †	198	8·25
Interpreted, no cache	549	2288
Compiled, functional cache †	2728	113·67
Compiled, piped cache †	9876	411·50

† Compilation time: approximately 16 s

into the system has an obvious cost in execution time, in a practical system the cache simulation is likely to be a pivotal part of the proposed architecture being simulated. Interpreted simulators are also slowed down when caching is incorporated. We include the cache timing to show that the functionally implemented cache incurs significantly less cost, even though there are associated drawbacks, as discussed below.

Performance statistics on cache hits and misses can be gathered by having (1) the macro pass the address of the current instruction (which is the first parameter to the macro) to a function which performs the cache mechanism, or by (2) writing the address out to standard output, where it can be read by a separate cache program, or by (3) sending output to a file for post-simulation analysis. As shown above, the first approach, a functionally implemented cache, has the advantage of speed, but the second approach, using a piped cache, has the advantage of generality—the cache program and the simulator are distinct; the cache size can be increased without recompiling the simulator, and the instruction set can be modified without changing the cache. The final approach, which we did not benchmark, would also incur the overhead of file I/O and require that an additional program be executed to generate statistics on the cache performance.

CONCLUSIONS

Compiled simulation is clearly superior to interpreted simulation, especially when the architecture being simulated can be easily described by a sequence of C statements. It has the advantages of efficiency, portability and elegance. Although other programming languages could be used, the C preprocessor and the peculiarities of the C switch statement make it particularly well suited. A macro assembler might do a better job of translating certain architectures (i.e. ones with flags), but at a significant loss of portability.

The compiled approach does have the drawback that it does not conceptually model the processor's internal actions as well as the interpreted approach. At this level of abstraction, however, one would probably need a completely different simulator, one which is event driven. Consequently, compiled instruction simulators are best used as an efficient instruction set prototyping tools.

REFERENCES

1. D. R. Ditzel and A. D. Berenbaum, 'Using CAD tools in the design of CRISP', *IEEE Design* and Test, June 1987, pp. 21–31.
2. D. A. Patterson, 'Reduced instruction set computers', *Communications of the ACM,* **28,** (1), 8–21 (1985).
3. J. S. Birnbaum and W. S. Worley, Jr., 'Beyond RISC: high-precision architecture', *IEEE* COMPCON. 1986. pp. 40–47.
4. A. D. Berenbaum; D. R. Ditzel and H. R. McLellan, 'Introduction to the CRISP instruction set architecture', *IEEE* COMPCON, 1987, pp. 86–90.
5. C. E. Gimarc and V. M. Mulutinovic, 'A survey of RISC processors and computers of the mid-1980s', *IEEE Computer,* September 1987, pp. 59–69.
6. A. Tanenbaum, J. Stevenson, E. G. Keizer and H. Van Staveren, 'Description of an experimental machine architecture for use with block structured languages', in *Informatica Rapport '81,* Vrije University, Amsterdam, January 1983.
7. N. Wirth, 'Microprocessor architectures: a comparison based on code generation by compiler', *Communications of the ACM,* **29,** (10), 978–990 (1986).

8. P. Schulthess, ‘A reduced high-level-language instruction set’, *IEEE Micro*, **4,** (June), 8–20 (1984).
9. D. Gries, *Compiler Construction for Digital Computers,* Wiley, New York, 1971.
10. P. Calingaert, *Assemblers, Compilers, and Program Translation,* Computer Science Press, Rockville, MD, 1979.
11. C. May, ‘MIMIC: a fast System/370 simulator’, in *Proceedings of the SIGPLAN ’87 Symposium on Interpreters and Interpretive Techniques,* St. Paul, Minnesota, June 1987, pp. 1–13.
12. S. Dasgupta, *Computer Architecture: A Modern Synthesis, Volume 2,* Wiley, New York, 1989.
13. S. Dasgupta, ‘Hardware description languages in microprogramming systems’, *IEEE Computer,* **18,** (1985), pp. 67–76.
14. W. D. Murray, *Computer and Digital System Architecture,* Prentice Hall, Englewood Cliffs, New Jersey, 1990.

APPENDIX I. EXAMPLE PROGRAM

```
/* Simulated Stack Architecture */

#include <stdio. h>

/* RAM from 0..START-1, program at START..??? */
#define START    0x00001000

#define NCODES 15

typedef unsigned long int UL;

char *opc[NCOD=] = {
   "ADD   ", /*   A + (S++) -> A;                                 */
   "ADI n", /*   A+n->A;                                          */
   "CBLS n", /*   if (S++) <= A (unsigned) then n -> P;           */
   "HALT ", /*   halt;                                            */
   "JSR  n", /*   P -> (--S); n -> P;                             */
   "LAI n", /*   n -> A;                                          */
   "LAR n", /*   (F+n)  ->A;                                      */
   "LDI n", /*   A -> (--S); n -> A;                              */
   "LDR n", /*   A -> (--S); (F + n) -> A;                        */
   "LDS n", /*   n -> S;                                          */
   "LINK n", /*   F-> (--S); S->F; S+n ->S;                       */
   "PA    ", /*   A -> (--S);                                     */
   "PI n", /*   n -> (--S);                                       */
   "SAR n", /*   A-> (F+n);                                       */
   "UNLK n" /*   F -> S; (S++) -> F; (S++) -> P; S + n -> S; */
};

UL cycles[NCODES] = {
   2, 2, 3, 3, 3, 2, 3, 3, 4, 2, 3, 2, 3, 3, 4
};

UL freq[NCODES], PC, FP, SP, AC;

#define paddr(a)   fprintf(stderr, "%1x (START + %ld)", (a), \
                           (a) - START)
```

```
void main()
{
   int i;
   UL tinsr = 0, tcyc = 0, starttime, *mem, *malloc(), time();

   if ((mem = malloc(START * sizeof(UL))) == NULL) {
      fprintf(stderr, "ERROR - cannot allocate RAM.\n|";
      exit(10);
   }
   starttime = time(0);
   if (!execute(mem))
      fprintf(stderr, "Executed bad address at\n");
   else
      fprintf(stderr, "Normal termination at\n");
   for (i = 0; i < NCODES; ++i) {
      tinsr += freq[i];
      tcyc += freq[i] * cycles[i];
   }
   fprintf(stderr, " P = ");
   paddr(PC);
   fprintf(stderr, ".\n S = ");
   paddr(SP);
   fprintf(stderr, ".\n   F=");
   paddr(FP);
   fprintf(stderr, ".\n   A = %lx (%ld).\n\n", AC, AC);
   fprintf(stderr,
           "Total instructions : %lu.  Total cycles : %lu.\n\n",
           tinsr, tcyc);
   fprintf(stderr,
           "Total execution time : %lu seconds.\n\n",
           time(0) - starttime);
   for (i = 0; i < NCODES; ++i)
      fprintf(stderr, "%-10s%1Olu%10.2f%10lu%10.2f\n",
              opc[i], freq[i], freq[i] * 100.0 / tinsr,
              freq[i] * cycles[i], freq[i] * cycles[i] *
              100.0 / tcyc);
}
*ifdef CACHE
#define FETCH(a)   case a: putw(a, stdout);
#else
#define FETCH(a)   case a:
#endif

#define SAVESTATE(a)   PC= a; SP=S-M; FP=F -M; AC=A;

/* opcode definitions */

#define ADD(a)       FETCH(a); ++freq[0]; A += *S++;
#define ADI(a, n)    FETCH(a); ++freq[1]; A += n;
#define CBLS(a, n)   FETCH(a); ++freq[2]; if (*S++ <= A) \
                        {P =n; break; }
```

```
#define HALT(a)      FETCH(a); ++ freq[3]; SAVESTATE(a); return 1;
#define JSR(a, n)    FETCH(a); ++freq[4]; *--S = a + 2; \
                        P = n; break;
#define LAI(a, n)    FETCH(a); ++freq[5]; A = n;
#define LAR(a, n)    FETCH(a); ++freq[6]; A = F[n];
#define LDI(a, n)    FETCH(a); ++freq[7]; *--S = A; A = n;
#define LDR(a, n)    FETCH(a); ++freq[8]; *--S = A; A = F[n];
#define LDS(a, n)    FETCH(a); ++freq[9]; S = M + (n);
#define LINK(a, n)   FETCH(a); ++freq[10]; *--S = F - M; \
                        F= S;S+=n;
#define PA(a)        FETCH(a); ++freq[11]; *--S = A;
#define PI(a, n)     FETCH(a); ++freq[12]; *--S = n;
#define SAR(a, n)    FETCH(a); ++freq[13]; F[n] = A;
#define UNLK(a, n)   FETCH(a); ++freq[14]; S = F; F = M + *S++; \
                        p = *S++; S += n; break;

int execute(n)
   register UL * M;
{
   register UL * S = M, *F = H, A = 0, P = START;

   for (;;)   /* ever */
      switch (P) {
      LDS (START,      START) ;
      JSR (START + 2, START + 5);
      HALT(START + 4);
      LINK(START + 5,  -2) ;  /* main() { int i, value; */
      LAI (START + 7, 1);     /* for (i = 1; i <= 100; ++i) */
      SAR (START+ 9, -1);
      PI (START + 11, 24); /*   value = fib(24);          */
      JSR (START + 13, START + 31);
      SAR (START + 15, -2);
      LAR (START + 17, -1);
      ADI (START + 19, 1);
      SAR (START + 21, -1);
      LDI (START + 23, 100);
      CBLS(START + 25, START + 11)
      LAR (START + 27, -2);   /* return value; }          */
      UNLK(START + 29, 0);
      LINK(START+ 31, 0);     /* fib(x) int x; {          */
      LAR (START + 33, 2);    /* if (x > 2)               */
      LDI (START + 35, 2);
      CBLS(START + 37, START + 56);
      LAR (START + 39, 2); /*   return fib(x - 1) +      */
      ADI (START + 41, -1);
      PA (START + 43);
      JSR (START + 44, START + 31);
      LDR (START+ 46, 2); /*          fib(x - 2);       */
      ADI (START + 48, -2);
      PA (START + 50);
      JSR (START + 51, START + 31);
      ADD (START + 53);
      UNLK(START + 54, 1);
      LAI (START + 56, 1);    /* else return 1; }          */
      UNLK(START + 58, 1);
   default: SAVESTATE(P); return 0;
   }
}
```

APPENDIX II. PROGRAM OUTPUT

```
Normal termination at
   P = 1004 (START + 4).
   s = 1000 (START + 0).
   F = 0 (START + -4096).
   A = b520 (46368).

Total instructions : 92735408. Total cycles : 264296023.

Total execution time :  198 seconds.

ADD           4636700      6.00   9273400      3.51
ADI  n        9273500     10.00  18547000      7.02
CBLS n        9273600     10.00  27820800     10.53
HALT                1      0.00         3      0.00
JSR  n        9273501     10.00  27820503     10.53
LAI  n        4636801      5.00   9273602      3.51
LAR  n       13910301     15.00  41730903     15.79
LDI  n        9273600     10.00  27820800     10.53
LDR  n        4636790      5.00  18546800      7.02
LDS  n              1      0.00         2      0.00
LINK n        9273501     10.00  27820503     10.53
PA            9273400     10.00  18546800      7.02
PI   n            100      0.00       300      0.00
SAR  n            201      0.00       603      0.00
UNLK n        9273501     10.00  37094004     14.04
```