USER'S GUIDE
Ben-Ari PCODE Disassembler

Bill Bynum/Tracy Camp
College of William and Mary/University of Alabama

November 2, 1998

1. Introduction

The purpose of this document is to provide a brief description of the Ben-Ari PCODE Disassembler and how to use it. The disassembler will "disassemble" or "de-compile" a PCODE file produced by either the BACI Pascal compiler or the BACI C−− compiler. If the source files that were used to produce the PCODE file are available, then the relevant source lines will be interspersed between the PCODE instructions that the source lines produce.

The Pascal version of the compiler and the interpreter were originally procedures in a program written by M. Ben-Ari, based on the original Pascal compiler by Niklaus Wirth. The program source was included as an appendix in Ben-Ari's book, "Principles of Concurrent Programming". The original version of the BACI compiler and interpreter was created from that source code and was hosted on a PRIME mainframe. After several modifications and additions, this version was ported to a PC version in Turbo Pascal, then to Sun Pascal, and to C. Finally, the compiler and interpreter were split into two separate programs. Recently, the C−− compiler was developed to compile source programs written in a restricted subset of C++ into PCODE executable by the interpreter. The disassembler was developed during the summer of 1997 to facilitate debugging the incorporation of external variables and separate compilation into the BACI system of programs.

The average user of the BACI system will normally not be concerned with the actual PCODE generated by the BACI compilers. However, the information provided by the disassembler might be useful to the user who is curious about how the BACI system works. The information provided by the disassembler is invaluable in tracking down errors in either of the compilers or in the interpreter.

2. PCODE Instruction Format

A PCODE instruction in the BACI system consists of three fields. The leftmost field is the PCODE opcode for the instruction. The middle and right fields of the instruction, called the `x` and `y` fields, respectively, are "modifiers" used to convey to the interpreter other information about the instruction. For example, a typical `LOAD_ADDR` PCODE instruction has the form

```
0    1    6
```

The `0` is the opcode for the `LOAD_ADDR` PCODE instruction. The variable whose address is being pushed on the runtime stack is located at an offset of `6` into the stackframe of the process currently running at level `1`.

For a typical program containing such an instruction, the disassembler would use the symbol table and debugging information included in the PCODE file to provide the user with a mnemonic description of the instruction and the name of the variable whose address is being loaded:

```
0    1    6  LOAD_ADDR, push &i
```

3. An Example

We return to the `incremen.cm` example used in the last section of the User's Guide for the C−− compiler. Here is the compiler listing:

```
BACI System: C-- to PCODE Compiler, 10:31  21 Oct 1997
Source file: incremen.cm  Fri Sep  8 16:51:00 1995
 line  pc
   1    0  const int m = 5;
   2    0  int n;
   3    0
   4    0  void incr(char id)
   5    0  {
   6    0     int i;
   7    0
   8    0     for(i = 1; i <= m; i = i + 1)
   9   14     {
  10   14         n = n + 1;
  11   19         cout << id << "  n =" <<  n << "  i =";
  12   25         cout << i << " " << id << endl;
  13   31     }
  14   32  }
  15   33
  16   33  main()
  17   34  {
  18   34     n = 0;
  19   37     cobegin
  20   38     {
  21   38        incr( 'A'); incr( 'B' ); incr('C');
  22   50     }
  23   51     cout << "The sum is " <<  n << endl;
  24   55  }
```

Disassembling the `incremen.pco` file produces the following output file:

```
BACI System: BenAri PCODE Disassembler, 10:33  21 Oct 1997
PCODE file: incremen.pco  Thu Oct 23 06:51:10 1997

BACI System: C-- to PCODE Compiler, 10:31  21 Oct 1997
Source file: incremen.cm  Fri Sep  8 16:51:00 1995
Reading from source file 'incremen.cm'

      1 const int m = 5;
      2 int n;
      3
      4 void incr(char id)
      5 {
      6    int i;
      7
      8    for(i = 1; i <= m; i = i + 1)

  lc    f    x    y     PCODE
   0    0    1    6  LOAD_ADDR, push &i
   1   24    0    1  PUSH_LIT 1
   2   38    0    0  STORE, s[s[t-1]] = s[t], pop(2)
   3    1    1    6  LOAD_VALUE, push i
   4   24    0    5  PUSH_LIT 5
   5   48    0    0  TEST_LE, pop(1), s[t] = (s[oldt-1] <= s[oldt])
   6   15    0   32  JZER s[t] to 32, pop(1)
   7   14    0   14  JUMP to 14
   8    0    1    6  LOAD_ADDR, push &i
   9    1    1    6  LOAD_VALUE, push i
  10   24    0    1  PUSH_LIT 1
  11   52    0    0  DO_ADD, pop(1), s[t] = (s[oldt-1] + s[oldt])
```

```
12   38    0    0  STORE, s[s[t-1]] = s[t], pop(2)
13   14    0    3  JUMP to 3

    9    {
   10        n = n + 1;

14    0    0    0  LOAD_ADDR, push &n
15    1    0    0  LOAD_VALUE, push n
16   24    0    1  PUSH_LIT 1
17   52    0    0  DO_ADD, pop(1), s[t] = (s[oldt-1] + s[oldt])
18   38    0    0  STORE, s[s[t-1]] = s[t], pop(2)

   11        cout << id << "  n =" <<  n << "  i =";

19    1    1    5  LOAD_VALUE, push id
20   29    0    3  WRITE (char) s[t] to stdout, pop(1)
21   28    0    0  WRITE_RAWSTRING stab[0] to stdout
22    1    0    0  LOAD_VALUE, push n
23   29    0    1  WRITE (int) s[t] to stdout, pop(1)
24   28    0    6  WRITE_RAWSTRING stab[6] to stdout

   12        cout << i << " " << id << endl;

25    1    1    6  LOAD_VALUE, push i
26   29    0    1  WRITE (int) s[t] to stdout, pop(1)
27   28    0   12  WRITE_RAWSTRING stab[12] to stdout
28    1    1    5  LOAD_VALUE, push id
29   29    0    3  WRITE (char) s[t] to stdout, pop(1)
30   63    0    0  WRITELN

   13    }

31   14    0    8  JUMP to 8

   14 }

32   32    0    0  EXIT_PROC

   15
   16 main()

33   80    0   56  SHORTCALL to 56, shortcall_reg = pc, pc = 56

   17 {
   18    n = 0;

34    0    0    0  LOAD_ADDR, push &n
35   24    0    0  PUSH_LIT 0
36   38    0    0  STORE, s[s[t-1]] = s[t], pop(2)

   19    cobegin

37    4    0    0  COBEGIN

   20    {
   21       incr( 'A'); incr( 'B' ); incr('C');

38   18    0    4  MARKSTACK incr
```

```
 39   24    0   65  PUSH_LIT 65
 40   19    0    5  CALL, psize-1 = 5
 41    3    0    1  UPDATE_DISPLAY from level 1 out to level 0
 42   18    0    4  MARKSTACK incr
 43   24    0   66  PUSH_LIT 66
 44   19    0    5  CALL, psize-1 = 5
 45    3    0    1  UPDATE_DISPLAY from level 1 out to level 0
 46   18    0    4  MARKSTACK incr
 47   24    0   67  PUSH_LIT 67
 48   19    0    5  CALL, psize-1 = 5
 49    3    0    1  UPDATE_DISPLAY from level 1 out to level 0

    22    }

 50    5    0    0  COEND

    23    cout << "The sum is " <<  n << endl;

 51   28    0   14  WRITE_RAWSTRING stab[14] to stdout
 52    1    0    0  LOAD_VALUE, push n
 53   29    0    1  WRITE (int) s[t] to stdout, pop(1)
 54   63    0    0  WRITELN

    24 }

 55   31    0    0  HALT

++-outer-++:
 56   81    0    0  SHORTRET, pc = shortcall_reg
```

A thorough discussion of the PCODE above would be tedious to trudge through. However, a couple of things require explanation, since the above example varies considerably from the PCODE described in Moti Ben-Ari's book.

The `SHORTCALL` and `SHORTRET` instructions refer to a "register"-based call added to the PCODE interpreter in the summer of 1997 to handle initialization of the global variables and monitors. The stack cannot be used in this "call", since monitor initialization changes the stack significantly. This particular program has no global or monitor initialization to be performed, so the `SHORTCALL` and `SHORTRET` instruction combination degenerates to a `no-op`.

One particular source instruction deserves careful analysis: the increment of the `n` variable:

```
       10        n = n + 1;

   14    0    0    0  LOAD_ADDR, push &n
   15    1    0    0  LOAD_VALUE, push n
   16   24    0    1  PUSH_LIT 1
   17   52    0    0  DO_ADD, pop(1), s[t] = (s[oldt-1] + s[oldt])
   18   38    0    0  STORE, s[s[t-1]] = s[t], pop(2)
```

The interest in this program stems from the conflict that the concurrent processes have in executing the five PCODE instructions generated by this source instruction. A process attempting to complete the source instruction can be suspended by the interpreter on a context switch sometime between the `LOAD_ADDR` instruction and the `STORE` instruction. Another process can complete its `STORE`, which is then "cancelled" by the original process when it resumes execution and stores its value.

4. Using the Disassembler

Usage: `badis [optional_flags]  pcode_filename`
Optional flags:

```
-h  show this help
-s  don't display source code, even if available
```

The name of the PCODE file is required. The PCODE file is expected to have either a suffix of `.pco` or a suffix of `.pob`, since these are the only two types of object files that the compilers of the BACI system produce. The name of the disassembly output file will have the either the `.dco` or the `.dob` suffix, depending on the suffix of the PCODE file that was disassembled. suffix.

If the source code that generated the PCODE file is available, then the source code lines will be interleaved with the decoded PCODE instructions. The `-s` option disables the display of source code.

5. Using the Disassembly Listing and Interpreter to Debug Concurrent Execution

We return to the disassembly of the `incremen.pco` file presented in Section 2. Execution of this program by the BACI interpreter is interesting, because it demonstrates how concurrently executing threads can conflict when mutual access to a global variable is not exclusive. The three `incr` threads each increment the global variable `n` five times, so the total number of times that one is added to the global variable is 15. Yet, the value of the global variable at the end of the program is almost always less than 15.

As mentioned in Section 3, the increment of the `n` variable consist of the PCODE instructions:

```
   10        n = n + 1;

14    0    0    0  LOAD_ADDR, push &n
15    1    0    0  LOAD_VALUE, push n
16   24    0    1  PUSH_LIT 1
17   52    0    0  DO_ADD, pop(1), s[t] = (s[oldt-1] + s[oldt])
18   38    0    0  STORE, s[s[t-1]] = s[t], pop(2)
```

To illustrate how the conflict between the concurrently executing copies of the `incr` function are occurring, we will use the PCODE debugging feature of the BACI interpreter (the `-d` option).

```
$> bainterp -p -d incremen
Source file: incremen.cm  Fri Sep  8 16:51:00 1995
Executing PCODE ...
  33   80    0   56  SHORTCALL to 56, shortcall_reg = pc, pc = 56
(h = help)> h
Debugger Commands:
   b  lc     -- set a break at location 'lc'
   c         -- continue to the next breakpoint
   d         -- dump the stack of the current process
   d  t      -- dump 10 stack words from s[t] down to s[t-10]
   d  t b    -- dump stack words from s[t] down to s[b]
   h         -- show this help
   i         -- show current breakpoints
   p         -- show process table
   q         -- terminate execution
   s         -- execute one PCODE instruction
   RETURN    -- repeat singlestep or continue
   u  i      -- unset breakpoint[i]
```

```
   w         -- show where current execution is
   x         -- disassemble the next 10 instructions
   x  loc    -- dissassemble 10 instructions starting at 'loc'
(h = help)> b 16
(h = help)> b 18
```

The plan of attack is to stop at location `16`, before the `PUSH_LIT` instruction is executed, and at location `18`, just before the `STORE` instruction is executed. These two places are relevant, because at `16`, the value of `n` that the `incr` thread has read is on the top of its run-time stack, and at `18`, the value that the thread is preparing to store back into the global variable `n` is also on the top of its stack.

Each time each of the two breakpoints occur, we dump the stack of the currently executing thread. Eventually, we hope to see when an increment performed by one of the threads is lost.

Continuing,

```
(h = help)> c
Breakpoint 0   Process #3: incr
  16   24    0    1  PUSH_LIT 1
(h = help)> d
Stack for Process #3: incr from 2809 down to 2801
   0     0     1    67     4     1     0     0     0
(h = help)> c
Breakpoint 1   Process #3: incr
  18   38    0    0  STORE, s[s[t-1]] = s[t], pop(2)
(h = help)> d
Stack for Process #3: incr from 2809 down to 2801
   1     0     1    67     4     1     0     0     0
(h = help)> s
  19    1    1    5  LOAD_VALUE, push id
(h = help)> p

Process Table
 Process        Active  Suspend  PC    xpc   atomic
 0 main             0     -1     51      6      0
 1 incr             1     -1      1      5      0
 2 incr             1     -1      1      5      0
 3 incr             1     -1     19     18      0
Global Variables
type  name         level adr value
int    n              0   0     1
Mainproc Variables
Monitor Variables
Process Variables
Process #1  incr
int    i              1   6     0
char   id             1   5   A
Process #2  incr
int    i              1   6     0
char   id             1   5   B
Process #3  incr
int    i              1   6     1
char   id             1   5   C
```

From this execution sequence, we can see that process #3 has actually stored the value `1` in the global variable `n`. Note that process #3 was able to complete the five PCODE instructions that increment the global variable `n` (the PCODE instructions at locations `14`, `15`, `16`, `17`, and `18`) without being interrupted by a context switch.

These five PCODE instructions form what is called a **critical section**. The `incr` process must not be interrupted by a context switch while it is executing these instructions, otherwise, the increment of the global variable by the interrupted process could be lost.

Continuing,

```
(h = help)> c
C  n =1  i =1 CBreakpoint 0   Process #2: incr
  16   24    0    1  PUSH_LIT 1
(h = help)> d
Stack for Process #2: incr from 2609 down to 2601
   1     0     1    66     4     1     0     0     0
(h = help)> c
Breakpoint 1   Process #2: incr
  18   38    0    0  STORE, s[s[t-1]] = s[t], pop(2)
(h = help)> d
Stack for Process #2: incr from 2609 down to 2601
   2     0     1    66     4     1     0     0     0
```

Note the output "`C  n =1  i =1 C`" from process #3 that occurs before Breakpoint 0 is reached. Everything still seems normal. Process #2 reads the value of `1` from the global variable `n` and is prepared at location `18` to store the value `2` back to the global variable.

Continuing,

```
(h = help)> c

Breakpoint 0   Process #1: incr
  16   24    0    1  PUSH_LIT 1
(h = help)> d
Stack for Process #1: incr from 2409 down to 2401
   2     0     1    65     4     1     0     0     0
(h = help)> c
Breakpoint 0   Process #3: incr
  16   24    0    1  PUSH_LIT 1
(h = help)> d
Stack for Process #3: incr from 2809 down to 2801
   2     0     2    67     4     1     0     0     0
(h = help)> c
Breakpoint 1   Process #3: incr
  18   38    0    0  STORE, s[s[t-1]] = s[t], pop(2)
(h = help)> d
Stack for Process #3: incr from 2809 down to 2801
   3     0     2    67     4     1     0     0     0
(h = help)> c
Breakpoint 1   Process #1: incr
  18   38    0    0  STORE, s[s[t-1]] = s[t], pop(2)
(h = help)> d
Stack for Process #1: incr from 2409 down to 2401
   3     0     1    65     4     1     0     0     0
(h = help)> p

Process Table
 Process        Active  Suspend  PC    xpc   atomic
 0 main            0      -1     51      6      0
 1 incr            1      -1     18     16      0
 2 incr            1      -1     20     18      0
```

```
 3 incr             1     -1     20     48      0
Global Variables
type  name         level adr value
int    n              0   0     3
Mainproc Variables
Monitor Variables
Process Variables
Process #1  incr
int    i              1   6     1
char   id             1   5  A
Process #2  incr
int    i              1   6     1
char   id             1   5  B
Process #3  incr
int    i              1   6     2
char   id             1   5  C
```

First, note the newline that follows the `c` command to the interpreter. This newline is printed by process #3 to complete its output.

Apparently, process #2 was able to store the value of `2` in the global variable `n`, because process #1 has read this value and pushed it onto its runtime stack. However, trouble is beginning to happen here, because process #3 reads the same value of `2` from the global variable `n`. This has is evidently happened because process #1 was interrupted by a context switch before it was able to store its updated value of `3` back to the global variable.

The `-p` option given when the interpreter was invoked causes the interpreter to write to an external file the exact interleaving of PCODE instructions that occurred during the execution of the program. When we check this file (`incremen.xpc`), we can verify that the awkward context switch described in the previous paragraph did indeed occur.

instruction count	process number	PCODE location
81	1	14
82	1	15
83	1	16
84	1	17
85	3	16
86	3	17
87	3	18
88	3	19
89	1	18
90	1	19
99	2	20

Note that after process #1 executed instruction number 84, the `DO_ADD` instruction, a context switch occurs and process #3 then executes the instructions at locations `16`, `17`, `18`, and `19`, The `STORE` instruction at location `18` stores the value of `3` that process #3 has on its stack back to the global variable `n`.

Then, process #1 receives a chance to run and executes the instructions at locations `18` and `19`. When process #1 executes the `STORE` instruction at location `18`, this overwrites the value of `3` stored by process #3 with the value of `3` stored by process #1. In this way, the work of incrementing the global variable by process #3 is lost.

Now that we have seen how things have begun to go wrong, we can allow the program to complete by unsetting the two breakpoints:

```
(h = help)> u 0
(h = help)> u 1
(h = help)> c
A  n =3  i =1 B  n =3C  n =3  i =A
2   i =C1
B
A  n =4  i =2 A
C  n =5B  i =3   n =A  n =7  i =2 B7  i =
3 A
C
A  n =8  i =4 A
C  n =9  i =4 C
A  n =10B  n =10  i =  i =5 3 A
B
C  n =11  i =5 C
B  n =12  i =4 B
B  n =13  i =5 B
The sum is 13

PCODE execution trace stored in incremen.xpc
```

Since the sum kept in the global variable `n` turned out to be `13` at termination, it is clear that the conflict described in the previous paragraphs must have occurred once more during the remainder of execution.