tmbc 1.0- A virtual machine for CS445 (Compiler Construction)
- plus-
A Description of the Execution Environment for bC

The TM machine is from the original code from the compiler book (Louden) with **lots** of mods including expanded instruction set and much stronger debugging facilities but the same poor parser. The TM code is a single C file as in the original. I haven't had time to rewrite it from scratch, which it desperately needs.

The TM does 64-bit integer arithmetic but the addresses are 32 bit. The TM Language is an odd mix of assembler and machine code.

There is no assembler phase or linker. TM code is loaded and executed directly.

TM 4.6 is not backward compatible with 4.5 due to initialization differences.

DATA LAYOUT

There are 8 registers numbered 0-7.

Register 7 is the program counter and is denoted PC below.

Register 0 is a pointer to the highest address in data memory. It is not required that it be maintained as a constant but for the standard C- execution environment it is.

Register 0 is initialized to the highest address in data memory. All other registers are initialized to 0.

Memory comes in two "segments": instruction memory and data memory.

iMem INSTRUCTION MEMORY

> Each memory location contains both an instruction and a comment. That is when the original assembler reads code into memory it remembers the comment! The comment is very useful in debugging! iMem is initialized to Halt instructions and the comment: "* initially empty"

dMem DATA MEMORY

> In TM 4.5 and earlier dMem[0] was initialized with the address of the last element in dMem. Now that value is stored in R0.
>
> All of dMem is zeroed. Each location in data is commented with whether the memory has been used or not. If it has been used the comment is the instruction address of the last instruction that wrote at that location. dMem can be marked "read only" on an address by address basis see the LIT instruction.

GENERAL FORMAT OF TM INSTRUCTIONS

Lines of TM code look like one of these general forms:

* <comment> a general full line comment

addr <instruction> <comment> set INSTRUCTION MEMORY at addr to this instruction. These come in Register Only, Register to Memory,

addr LIT <value> set DATA MEMORY at (TopOfMemory - addr) to this value. By default addr will be the offset from register 0.

REGISTER TO MEMORY INSTRUCTIONS (RA instruction format)

Form: addr instruction r, d(s)

The "d" in the instruction format below can be an integer or a character denoted by characters enclosed in single quotes. If the first character is a caret it means control. '^M' is Control-M etc. Backslash is understood for '\0', '\t', '\n', '\'' and '\\'.

X is ignored, but must be present.

```
LDC r, d(X) reg[r] = d                (load constant d; immediate; X ignored)
LDA r, d(s) reg[r] = d + reg[s]       (load direct address)
LD  r, d(s) reg[r] = dMem[d + reg[s]] (load indirect)

ST  r, d(s) dMem[d + reg[s]] = reg[r]

JNZ r, d(s) if reg[r]!=0 reg[PC] = d + reg[s]   (jump nonzero)
JZR r, d(s) if reg[r]==0 reg[PC] = d + reg[s]   (jump zero)
JMP X, d(s) reg[PC] = d + reg[s]        (jump,  X is ignored but often is 7)
```

REGISTER ONLY INSTRUCTIONS (RO instruction format) (instruction memory)

Form: addr instruction r, s, t

```
HALT  X, X, X   stop execution (all registers ignored)
NOP   X, X, X   does nothing but take space (all registers ignored)
IN    r, X, X   reg[r] <- input integer value of register r from stdin
INB   r, X, X   reg[r] <- input boolean value of register r from stdin
INC   r, X, X   reg[r] <- input char value of register r from stdin
OUT   r, X, X   reg[r] -> output integer value of register r to stdout
OUTB  r, X, X   reg[r] -> output boolean value of register r to stdout
OUTC  r, X, X   reg[r] -> output char value of register r to stdout
OUTNL X, X, X   output a newline to stdout

ADD   r, s, t   reg[r] = reg[s] + reg[t]
SUB   r, s, t   reg[r] = reg[s] - reg[t]
MUL   r, s, t   reg[r] = reg[s] * reg[t]
DIV   r, s, t   reg[r] = reg[s] / reg[t]  (only a truncating integer divide)
MOD   r, s, t   reg[r] = reg[s] % reg[t] (unlike C, always returns the
                                          NONNEGATIVE modulus of reg[s]
                                          mod reg[t])

AND   r, s, t    reg[r] = reg[s] & reg[t]   (bitwise and)
OR    r, s, t    reg[r] = reg[s] | reg[t]   (bitwise or)
```

```
XOR   r, s, t    reg[r] = reg[s] ^ reg[t]    (bitwise xor)
NOT   r, s, X    reg[r] = ~ reg[s]           (bitwise complement)
NEG   r, s, X    reg[r] = - reg[s]           (negative)
SWP   r, s, X    reg[r] = min(reg[r], reg[s]), reg[s] = max(reg[r], reg[s])
                                             (useful for min or max)
RND r, s, X reg[r] = random(0, |reg[s]-1|)   (get random num between
                                              0 and |reg[s]-1| inclusive)
```

TEST INSTRUCTIONS (RO instruction format) (instruction memory)

Form: addr instruction r, s, t

```
SLT and SGT are signed test instructions useful for things like for-loops.

TLT r, s, t    if reg[s]<reg[t] reg[r] = 1 else reg[r] = 0
TLE r, s, t    if reg[s]<=reg[t] reg[r] = 1 else reg[r] = 0
TEQ r, s, t    if reg[s]==reg[t] reg[r] = 1 else reg[r] = 0
TNE r, s, t    if reg[s]!=reg[t] reg[r] = 1 else reg[r] = 0
TGE r, s, t    if reg[s]>=reg[t] reg[r] = 1 else reg[r] = 0
TGT r, s, t    if reg[s]>reg[t] reg[r] = 1 else reg[r] = 0
SLT r, s, t    if (reg[r]>=0) reg[r] = (reg[s]<reg[t] ? 1 : 0);
               else reg[r] = (-reg[s] < -reg[t] ? 1 : 0);
SGT r, s, t    if (reg[r]>=0) reg[r] = (reg[s]>reg[t] ? 1 : 0);
               else reg[r] = (-reg[s] > -reg[t] ? 1 : 0);
```

BLOCK MEMORY TO MEMORY INSTRUCTIONS (MM instructions in RO format)

Form: addr instruction r, s, t

These instructions use the registers as parameters for block moves of memory without the need for loops. It is as if a memory controller could move masses of memory by itself. The test instructions CO and COA set a pair of registers based on comparing two blocks of memory. Overlapping source and target blocks of memory is undefined.

```
MOV r, s, t   dMem[reg[r] - (0..reg[t]-1)] = dMem[reg[s] - (0..reg[t]-1)]

SET r, s, t   dMem[reg[r] - (0..reg[t]-1)] = reg[s]
                              Makes reg[t] copies of reg[s].
                              Useful for zeroing out memory.

CO  r, s, t   reg[r] = dMem[reg[r]  + k] (for the first k that yields a diff
                                          or the last tested if no diff)
              reg[s] = dMem[reg[s] + k]  (for the first k that yields a diff
                                          or the last tested if no diff)
              WARNING: memory is scanned from higher addresses to lower.
              reg[t] is the size of the arrays  compared.
              Both reg[r] and reg[s] are set so they can then be tested using
              any of the test instructions.

COA r, s, t   reg[r] = reg[r] + k (for the first k that yields a diff   at
                             that address or   the last tested if no diff)
              reg[s] = reg[s] + k (for the first k that yields a diff at that
                                   address or the last tested if no diff)
              WARNING: memory is scanned from higher addresses to lower.
```

```
            reg[t] is the size of the arrays
```

LITERAL INSTRUCTIONS (data memory)

Form: addr LIT constant

```
LIT 666    load into DATA MEMORY the single "word" 666 at offset
           from top of memory.
           WARNING: the address for this command is the offset from R0.
LIT 'x'    load into DATA MEMORY the single "word" 'x' at offset
           from top of memory
           WARNING: the address for this command is the offset from R0.
LIT "stuff"  load into DATA MEMORY the string starting with the first
             character at the address given and then *decrementing*
             from there. The size is then stored  in the address+1.
             WARNING: the address for this command is the offset
             from R0.
```

SOME TM IDIOMS

```
1. reg[r]++:
    LDA r, 1(r)
2. reg[r] = reg[r] + d:
    LDA r, d(r)
3. reg[r] = reg[s]
    LDA r, 0(s)
4. goto reg[r] + d
    LDA 7, d(r)
   or
    JMP X, d(r)
5. goto relative to pc (d is number of instructions skipped!!)
    LDA 7, d(7)
   or
    JMP X, d(7)
6. NOOP:
    LDA r, 0(r)
7. save address of following command for return in reg[r]
    LDA r, 1(7)
8. jump to address d(s) if reg[s] > reg[t]?
     TGT r, s, t    reg[r] = (reg[s] > reg[t] ? 1 : 0)
     JNZ r, d(s)     if reg[r]>0 reg[PC] = d + reg[s]
9. jump vector at reg[r] > vector at reg[s] of length reg[t]
     CO  r, s, t    compare two vectors -> reg[5] and reg[6]
     TGT r, 5, 6    reg[r] = (reg[s] > reg[t] ? 1 : 0)
     JNZ r, d(s)     if reg[r]>0 reg[PC] = d + reg[s]
```

TM EXECUTION

This is how execution actually works:

```
pc <- reg[7]
test pc in range
reg[7] <- pc+1
```

```
inst <- fetch(pc)
exec(inst)
```

Notice that at the head of the execution loop above reg[7] points to the instruction BEFORE the one about to be executed. Then the first thing the loop will do is increment the PC. During an instruction execution the PC points at the instruction executing.

So LDA 7, 0(7) does nothing but because it leaves pointer at next instr

So LDA 7, -1(7) is infinite loop

Memory comes in two segments: instruction and data. When TM is started, cleared, or loaded then all data memory is zeroed and marked as unused and data memory position 0 is loaded with the address of the last spot in memory (highest accessible address). All instruction memory is filled with halt instructions. The reg[7] is set to the beginning of instruction memory.

TMbC version 1.0

Commands are:

```
a(bortLimit <<n>>     Maximum number of instructions between
                      halts (default is 50000).
b(reakpoint <<n>>     Set a breakpoint for instr n.
                      No n means clear breakpoints.
c(lear                Reset TM for new execution of program
d(Mem <b <n>>         Print n dMem locations (counting down)
                      starting at b (n can be negative to count
                      up). No args means all used memory
                      locations.
e(xecStats            Print execution statistics since last load
                      or clear
g(o                    Execute TM instructions until HALT
h(elp                  Cause this list of commands to be printed
i(Mem <b <n>>          Print n iMem locations (counting up)
                      starting at b. No args means all used
                      memory locations
l(oad filename        Load filename into memory (default is last
                      file)
n(ext                 Print the next command that will be
                      executed
o(utputLimit <<n>>    Maximum combined number of calls to any
                      output  instruction (default is 1000)
p(rint                Toggle printing of total number
                      instructions executed ('go' only)
q(uit                 Terminate TMbC
r(egs                 Print the contents of the registers
s(tep <n>             Execute n (default 1) TMbC instructions
```

```
t(race                  Toggle instruction tracing (printing)
                        during execution
u(nprompt               Unprompted for script input
v                       Print the version information
x(it                    Terminate TMbC
= <r> <n>               Set register number r to value n
                       (e.g. set the pc)
< <addr> <value>        Set dMem at addr to value
(empty line does a step)
Also a # character placed after input will cause TMbC to halt
after processing the IN or INB commands (e.g. 34# or f# )
```

INSTRUCTION INPUT

Instructions are input to the memory segments via the load command. For example:

```
39:   ADD   3,4,3 op +
```

* Add standard closing in case there is no return statement

```
65:   LDC   2,0(6)      Set return value to 0
66:   LD    3,-1(1)     Load return address
67:   LD    1,0(1)      Adjust fp
68:   LDA   7,0(3)      Return
3:    LIT   "dogs"      A literal stored at data memory at
                        (top of memory - 3)
```

A note about string literals: 39: LIT "dogs" looks like if the top of memory is 9999.

```
9963: 0          unused
9962: 0          unused
9961: 4          readOnly   <-- size
9960: 100  'd'   readOnly   <-- address given in LIT
9959: 111  'o'   readOnly
9958: 103  'g'   readOnly
9957: 115  's'   readOnly
9956: 0          unused
```

A Description of the Execution Environment for C-

THE USE OF TM REGISTERS IN C-

These are the assigned registers for our virtual machine. Only register 7 is actually configured by the "hardware" to be what it is defined below. The rest is whatever we have made it to be.

```
0 - global pointer (points to the space for global variables)
1 - the local frame pointer (initially right after the globals)
2 - return value from a function (set at end of function call)
```

```
3,4,5,6 - accumulators
7 - the program counter or pc (used by TM)
```

Memory Layout

THE FRAME LAYOUT

Activation Records or Frames for procedures are laid out in data memory as follows:

reg1 ->	old frame pointer (old reg1)	loc
	addr of instr to execute upon return	loc-1
	parm 1	loc-2
	parm 2	loc-3
	parm 3	loc-4
	local var 1	loc-5
	local var 2	loc-6
	local var 3	loc-7
	temp var 1	loc-8
	temp var 2	loc-9

- The first two positions are the "return ticket" for use when you are done executing this function. They say how to restore reg 1 to the old frame pointer position and the where to set the PC return to just
- after the call to this function. By moving reg 1 away from this frame you destroy access to this local frame.
- Parms are parameters for the function. They are always one "word" per parameter. Arrays are passed a pointer to the 0th element in the array call the "base address" of the array.
- Locals are locals in the function both defined at the beginning of the procedure and in compound statements inside the procedure.

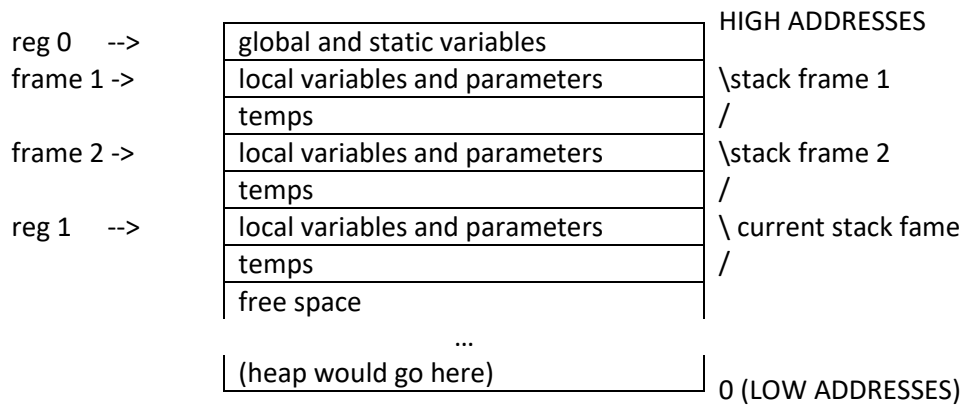
Note that we can save space by overlaying non-concurrent compound statement scopes.

- Temps are used to stretch the meager number of registers we have. For example in doing (3+4)*(5+6)+7 we may need more temps than we have. In many compilers, during the intermediate stage they assume an infinite number of registers and then do a register allocation algorithm to optimize register use and execution time.

THE STACK LAYOUT

register 0 is initialized with the address of the highest address element in data space. The diagram below is how the globals, frames and heap (which we don't have) would be laid out in data memory. Note that temps may be on the stack before a frame is placed on. This happens when a function is called in the middle of an expression. Register 0 points to the first location in global variable space and register 1 points to the currently executing stack frame.

MAP OF DATA SPACE

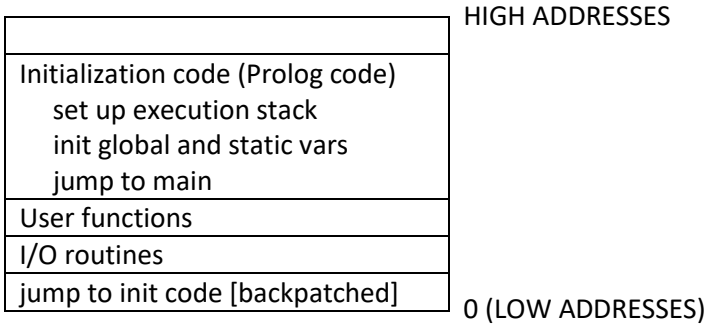


We do not currently have a heap or garbage collection in bC.

THE INSTRUCTION SPACE LAYOUT

Instructions are loaded in instruction memory starting at address 0. When the go command is issued execution begins with address 0.

MAP OF INSTRUCTION SPACE



Important Code Patterns

GENERATING CODE

COMPILE TIME Variables: These are variables you might use when computing where things go in memory

goffset - the global offset is the relative offset of the next available space in the global space.

foffset - the frame offset is the relative offset of the next available space in the frame being built

toffset - the temp offset is the offset from the frame offset next available temp variable.

IMPORTANT: that these values will be negative since memory is growing downward to lower addresses in this implementation!!

PROLOG CODE

This is the initialization code that is called at the beginning of the program. It sets up registers 0 and 1 and jumps to main. Returning from main halts the program.

```
0:    JMP   7,XXX(7)   Jump to init [backpatch]

( body of code including main goes here )

* INIT
* Init globals and statics

( code to init globally allocated variables goes here!  )

* End init globals and statics
53:   LDA   1,XXX(0)   set first frame at end of globals
54:   ST    1,0(1)     store old fp (point to self!)
55:   LDA   3,1(7)     Return address in ac (point to halt)
56:   JMP   7,XXX(7)   Jump to main
57:   HALT  0,0,0 DONE!
* END INIT
```

CALLING SEQUENCE (caller) [version 1]

At this point:

- reg1 points to the old frame
- off in the tm code below is the offset to first available space on stack past the parameters and local variables and any temps currently stored.
- off is relative to the beginning of the caller's frame
- foffset in the tm code below is the offset to first available parameter. relative to top of stack!
- func is the relative address to the called procedure. Use emitGotoAbs function from the emit library to emit this tm instruction with the correct RELATIVE address.

```
* construct the ghost frame
* figure where the new local frame will go
LDA 3, off(1)      * where is current top of stack is

* load the first parameter var1 (foffset = -2)
LD  4, var1(1)     * load in third slot of ghost frame
ST  4, foffset(3) * store in parameter space (then foffset--)

* load the second parameter var2
LD  4, var2(1)     * load in third temp
ST  4, foffset(3) * store in parameter space (then foffset--)

* begin call
```

```
ST  1, 0(3)      * store old fp in first slot of ghost frame
LDA 1, 0(3)      * move the fp to the new frame
LDA 3, 1(7)      * compute the return address at (skip 1 ahead) JMP 7,
func(7)          * call func
* return to here
```

At this point:

- reg1 points to the new frame (top of old local stack)
- reg3 contains return address in code space
- reg7 points to the next instruction to execute

CALLING SEQUENCE (caller) [version 2]

At this point:

- reg1 points to the old frame
- off in the tm code below points to first available space on stack. relative to the beginning of the frame.
- foffset in the tm code below points to first available parameter relative to the beginning of the frame
- func is the relative address to the called procedure. Use emitGotoAbs function from the emit library to emit this tm instruction with the correct RELATIVE address.

```
(foffset = end of current frame and temps)
ST   1, off(1)  * save old frame pointer at first part of new frame

* load the first parameter
LD   4, var1(1) * load in third temp
ST   4, foffset(1)   * store in parameter space (foffset--)

* load the second parameter
LD   4, var2(1) * load in third temp
ST   4, foffset(1)   * store in parameter space

* begin call
LDA 1, off(1)    * move the fp to the new frame
LDA 3, 1(7)      * compute the return address at (skip 1 ahead) JMP 7,
func(7)    * call func
* return to here
```

At this point:

- reg1 points to the new frame (top of old local stack)
- reg3 contains return address in code space
- reg7 points to the next instruction to execute

CALLING SEQUENCE (callee's prolog)

It is the callee's responsibility to save the return address. An optimization is to not do this if you can preserve reg3 throughout the call.

```
ST    3, -1(1)    * save return addr in current frame
```

RETURN FROM A CALL

```
* save return value
LDA 2, 0(x)
          * load the function return (reg2) with the answer from regx

* begin return
LD    3, -1(1)    * recover old pc LD   1, 0(1)     * pop the frame JMP
7, 0(3)     * jump to old pc

At this point:
reg2 will have the return value from the function
```

Examples of variable and constant access

LOAD CONSTANT

```
LDC 3, const(0)
```

RHS LOCAL VAR SCALAR

```
LD    3, var(1)
```

RHS GLOBAL VAR SCALAR

```
LD    3, var(0)
```

LHS LOCAL VAR SCALAR

```
LDA 3, var(1)
```

RHS LOCAL ARRAY

```
LDA 3, var(1) * array  base
SUB 3, 4      * index off of the base
LD  3, 0(3)   * access the element
```

LHS LOCAL ARRAY

```
LDA 3, var(1)    * array  base
SUB 3, 4         * index off of the base
ST  x, 0(3)      * store in array
```

EXAMPLE 1: A Simple C- Program Compiled

THE CODE

```
// C-F15
int dog(int x)
{
     int y;
     int z;

     y = x*111+222;
     z = y;

     return z;
}

main()
{
     output(dog(666));
     outnl();
}
```

THE OBJECT CODE

```
* bC compiler version bC-Su23
* File compiled: inputFiles/drbcBasic.bC
*
* ** ** ** ** ** ** ** ** ** ** ** **
* FUNCTION input
1:    ST    3,-1(1)     Store return address
2:    IN    2,2,2      Grab int input
3:    LD    3,-1(1)     Load return address
4:    LD    1,0(1)      Adjust  fp
5:    JMP   7,0(3)      Return
* END FUNCTION  input
*
* ** ** ** ** ** ** ** ** ** ** ** **
* FUNCTION inputb
6:    ST    3,-1(1)     Store return address


7:    INB   2,2,2       Grab bool input
8:    LD    3,-1(1)     Load return address
```

```
9:    LD    1,0(1)     Adjust  fp
10:   JMP   7,0(3)     Return
* END FUNCTION  inputb
*
* ** ** ** ** ** ** ** ** ** ** ** **
* FUNCTION inputc
11:   ST    3,-1(1)    Store return address
12:   INC   2,2,2       Grab char input
13:   LD    3,-1(1)    Load return address
14:   LD    1,0(1)     Adjust  fp
15:   JMP   7,0(3)     Return
* END FUNCTION  inputc
*
* ** ** ** ** ** ** ** ** ** ** ** **
* FUNCTION output
16:   ST    3,-1(1)    Store return address
17:   LD    3,-2(1)    Load parameter
18:   OUT   3,3,3       Output integer
19:   LD    3,-1(1)    Load return address
20:   LD    1,0(1)     Adjust  fp
21:   JMP   7,0(3)     Return
* END FUNCTION  output
*
* ** ** ** ** ** ** ** ** ** ** ** **
* FUNCTION outputb
22:   ST    3,-1(1)    Store return address
23:   LD    3,-2(1)    Load parameter
24:   OUTB  3,3,3       Output bool
25:   LD    3,-1(1)    Load return address
26:   LD    1,0(1)     Adjust  fp
27:   JMP   7,0(3)     Return
* END FUNCTION  outputb
*
* ** ** ** ** ** ** ** ** ** ** ** **
* FUNCTION outputc
28:   ST    3,-1(1)    Store return address
29:   LD    3,-2(1)    Load parameter
30:   OUTC  3,3,3       Output char
31:   LD    3,-1(1)    Load return address
32:   LD    1,0(1)     Adjust  fp
33:   JMP   7,0(3)     Return
* END FUNCTION  outputc
*
* ** ** ** ** ** ** ** ** ** ** ** **
* FUNCTION outnl
34:   ST    3,-1(1)    Store return address
35:   OUTNL 3,3,3       Output a newline
36:   LD    3,-1(1)    Load return address
```

```
37:   LD    1,0(1)     Adjust  fp
38:   JMP   7,0(3)     Return
* END FUNCTION  outnl
*

* ** ** ** ** ** ** ** ** ** ** ** **
* FUNCTION dog
39:   ST    3,-1(1)    Store return address
* COMPOUND
* Compound Body
* EXPRESSION
40:   LD    3,-2(1)    Load variable x
41:   ST    3,-5(1)    Push left side
42:   LDC   3,111(6)   Load integer constant
43:   LD    4,-5(1)    Pop left into ac1
44:   MUL   3,4,3      Op *
45:   ST    3,-5(1)    Push left side
46:   LDC   3,222(6)   Load integer constant
47:   LD    4,-5(1)    Pop left into ac1
48:   ADD   3,4,3      Op +
49:   ST    3,-3(1)    Store variable y
* EXPRESSION
50:   LD    3,-3(1)    Load variable y
51:   ST    3,-4(1)    Store variable z
* RETURN
52:   LD    3,-4(1)    Load variable z
53:   LDA   2,0(3)     Copy result to return register
54:   LD    3,-1(1)    Load return address
55:   LD    1,0(1)     Adjust  fp
56:   JMP   7,0(3)     Return
* END COMPOUND
* Add standard closing in case there  is no return statement
57:   LDC   2,0(6)     Set return value to 0
58:   LD    3,-1(1)    Load return address
59:   LD    1,0(1)     Adjust  fp
60:   JMP   7,0(3)     Return
* END FUNCTION  dog
*
* ** ** ** ** ** ** ** ** ** ** ** **
* FUNCTION main
61:   ST    3,-1(1)    Store return address
* COMPOUND
* Compound Body
* EXPRESSION
* CALL output
62:   ST    1,-2(1)    Store fp in ghost frame for output
* Param 1
* CALL dog
```

```
63:   ST    1,-4(1)    Store fp in ghost frame for dog
* Param 1
64:   LDC   3,666(6)   Load integer constant
65:   ST    3,-6(1)    Push parameter
* Param end dog
66:   LDA   1,-4(1)    Ghost frame becomes new active frame
67:   LDA   3,1(7)     Return address in ac
68:   JMP   7,-30(7)   CALL dog
69:   LDA   3,0(2)     Save the result in ac
* Call end dog
70:   ST    3,-4(1)    Push parameter

* Param end output
71:   LDA   1,-2(1)    Ghost frame becomes new active frame
72:   LDA   3,1(7)     Return address in ac
73:   JMP   7,-58(7)   CALL output
74:   LDA   3,0(2)     Save the result in ac
* Call end output
* EXPRESSION
* CALL outnl
75:   ST    1,-2(1)    Store fp in ghost frame for outnl
* Param end outnl
76:   LDA   1,-2(1)    Ghost frame becomes new active frame
77:   LDA   3,1(7)     Return address in ac
78:   JMP   7,-45(7)   CALL outnl
79:   LDA   3,0(2)     Save the result in ac
* Call end outnl
* END COMPOUND
* Add standard closing in case there  is no return statement
80:   LDC   2,0(6)     Set return value to 0
81:   LD    3,-1(1)    Load return address
82:   LD    1,0(1)     Adjust  fp
83:   JMP   7,0(3)     Return
* END FUNCTION  main
0:    JMP   7,83(7)    Jump to init [backpatch]
* INIT
84:   LDA   1,0(0)     set first frame at end of globals
85:   ST    1,0(1)     store old fp (point to self)
* INIT GLOBALS AND STATICS
* END INIT GLOBALS AND STATICS
86:   LDA   3,1(7)     Return address in ac
87:   JMP   7,-27(7)   Jump to main
88:   HALT  0,0,0 DONE!
* END INIT
```

EXAMPLE 2: A Simple C- Program Compiled

THE CODE

```
// C-F15
// A program to perform Euclid's
//   Algorithm to compute gcd of two numbers you give.

int gcd(int u; int v)
{
   if (v == 0)   // note you can't say:      if (v)
      return u;
   else
      return gcd(v, u - u/v*v);
}

main()
{
   int x, y;
   int result;

   x = input();
   y = input();
   result = gcd(x, y); output(result); outnl();
}
```

THE OBJECT CODE

```
* bC compiler version bC-Su23
* File compiled: inputFiles/drbcBasic.bC
*
* ** ** ** ** ** ** ** ** ** ** ** **
* FUNCTION input
1:    ST    3,-1(1)     Store return address
2:    IN    2,2,2       Grab int input
3:    LD    3,-1(1)     Load return address
4:    LD    1,0(1)      Adjust  fp
5:    JMP   7,0(3)      Return
* END FUNCTION  input
*
* ** ** ** ** ** ** ** ** ** ** ** **
* FUNCTION inputb
6:    ST    3,-1(1)     Store return address
7:    INB   2,2,2       Grab bool input
8:    LD    3,-1(1)     Load return address
9:    LD    1,0(1)      Adjust  fp
10:   JMP   7,0(3)      Return
* END FUNCTION  inputb
*
* ** ** ** ** ** ** ** ** ** ** ** **
```

```
* FUNCTION inputc
11:   ST    3,-1(1)    Store return address
12:   INC   2,2,2      Grab char input
13:   LD    3,-1(1)    Load return address
14:   LD    1,0(1)     Adjust  fp
15:   JMP   7,0(3)     Return
* END FUNCTION  inputc
*
* ** ** ** ** ** ** ** ** ** ** ** **
* FUNCTION output
16:   ST    3,-1(1)    Store return address
17:   LD    3,-2(1)    Load parameter
18:   OUT   3,3,3      Output integer
19:   LD    3,-1(1)    Load return address
20:   LD    1,0(1)     Adjust  fp

21:   JMP   7,0(3)     Return
* END FUNCTION  output
*
* ** ** ** ** ** ** ** ** ** ** ** **
* FUNCTION outputb
22:   ST    3,-1(1)    Store return address
23:   LD    3,-2(1)    Load parameter
24:   OUTB  3,3,3      Output bool
25:   LD    3,-1(1)    Load return address
26:   LD    1,0(1)     Adjust  fp
27:   JMP   7,0(3)     Return
* END FUNCTION  outputb
*
* ** ** ** ** ** ** ** ** ** ** ** **
* FUNCTION outputc
28:   ST    3,-1(1)    Store return address
29:   LD    3,-2(1)    Load parameter
30:   OUTC  3,3,3      Output char
31:   LD    3,-1(1)    Load return address
32:   LD    1,0(1)     Adjust  fp
33:   JMP   7,0(3)     Return
* END FUNCTION  outputc
*
* ** ** ** ** ** ** ** ** ** ** ** **
* FUNCTION outnl
34:   ST    3,-1(1)    Store return address
35:   OUTNL 3,3,3      Output a newline
36:   LD    3,-1(1)    Load return address
37:   LD    1,0(1)     Adjust  fp
38:   JMP   7,0(3)     Return
* END FUNCTION  outnl
*
```

```
* ** ** ** ** ** ** ** ** ** ** ** **
* FUNCTION gcd
* TOFF set: -4
39:   ST    3,-1(1)    Store return address
* COMPOUND
* TOFF set: -4
* Compound Body
* IF
40:   LD    3,-3(1)    Load variable v
41:   ST    3,-4(1)    Push left side
* TOFF dec: -5
42:   LDC   3,0(6)     Load integer constant
* TOFF inc: -4
43:   LD    4,-4(1)    Pop left into ac1
44:   TEQ   3,4,3 Op ==
* THEN
* RETURN
46:   LD    3,-2(1)    Load  variable u
47:   LDA   2,0(3)     Copy  result to return register
48:   LD    3,-1(1)    Load  return address
49:   LD    1,0(1)     Adjust  fp
50:   JMP   7,0(3)     Return

45:   JZR   3,6(7)     Jump around the THEN if false [backpatch]
* ELSE
* RETURN
* CALL gcd
52:   ST    1,-4(1)    Store fp in ghost frame for gcd
* TOFF dec: -5
* TOFF dec: -6
* Param 1
53:   LD    3,-3(1)    Load variable v
54:   ST    3,-6(1)    Push parameter
* TOFF dec: -7
* Param 2
55:   LD    3,-2(1)    Load variable u
56:   ST    3,-7(1)    Push left side
* TOFF dec: -8
57:   LD    3,-2(1)    Load variable u
58:   ST    3,-8(1)    Push left side
* TOFF dec: -9
59:   LD    3,-3(1)    Load variable v
* TOFF inc: -8
60:   LD    4,-8(1)    Pop left into ac1
61:   DIV   3,4,3      Op /
62:   ST    3,-8(1)    Push left side
* TOFF dec: -9
63:   LD    3,-3(1)    Load variable v
```

```
* TOFF inc: -8
64:   LD    4,-8(1)    Pop left into ac1
65:   MUL   3,4,3      Op *
* TOFF inc: -7
66:   LD    4,-7(1)    Pop left into ac1
67:   SUB   3,4,3      Op -
68:   ST    3,-7(1)    Push parameter
* TOFF dec: -8
* Param end gcd
69:   LDA   1,-4(1)    Ghost frame becomes new active frame
70:   LDA   3,1(7)     Return address in ac
71:   JMP   7,-33(7)   CALL gcd
72:   LDA   3,0(2)     Save the result in ac
* Call end gcd
* TOFF set: -4
73:   LDA   2,0(3)     Copy result to return register
74:   LD    3,-1(1)    Load return address
75:   LD    1,0(1)     Adjust  fp
76:   JMP   7,0(3)     Return
51:   JMP   7,25(7)    Jump around the ELSE [backpatch]
* END IF
* TOFF set: -4
* END COMPOUND
* Add standard closing in case there  is no return statement
77:   LDC   2,0(6)     Set return value to 0
78:   LD    3,-1(1)    Load return address
79:   LD    1,0(1)     Adjust  fp
80:   JMP   7,0(3)     Return
* END FUNCTION  gcd

*
* ** ** ** ** ** ** ** ** ** ** ** **
* FUNCTION main
* TOFF set: -2
81:   ST    3,-1(1)    Store return address
* COMPOUND
* TOFF set: -5
* Compound Body
* EXPRESSION
* CALL input
82:   ST    1,-5(1)    Store fp in ghost frame for input
* TOFF dec: -6
* TOFF dec: -7
* Param end input
83:   LDA   1,-5(1)    Ghost frame becomes new active frame
84:   LDA   3,1(7)     Return address in ac
85:   JMP   7,-85(7)   CALL input
86:   LDA   3,0(2)     Save the result in ac
```

```
* Call end input
* TOFF set: -5
87:   ST    3,-2(1)    Store variable x
* EXPRESSION
* CALL input
88:   ST    1,-5(1)    Store fp in ghost frame for input
* TOFF dec: -6
* TOFF dec: -7
* Param end input
89:   LDA   1,-5(1)    Ghost frame becomes new active frame
90:   LDA   3,1(7)     Return address in ac
91:   JMP   7,-91(7)   CALL input
92:   LDA   3,0(2)     Save the result in ac
* Call end input
* TOFF set: -5
93:   ST    3,-3(1)    Store variable y
* EXPRESSION
* CALL gcd
94:   ST    1,-5(1)    Store fp in ghost frame for gcd
* TOFF dec: -6
* TOFF dec: -7
* Param 1
95:   LD    3,-2(1)    Load variable x
96:   ST    3,-7(1)    Push parameter
* TOFF dec: -8
* Param 2
97:   LD    3,-3(1)    Load variable y
98:   ST    3,-8(1)    Push parameter
* TOFF dec: -9
* Param end gcd
99:   LDA   1,-5(1)    Ghost frame becomes new active frame
100:  LDA   3,1(7)     Return address in ac
101:  JMP   7,-63(7)   CALL gcd
102:  LDA   3,0(2)     Save the result in ac
* Call end gcd
* TOFF set: -5

103:  ST    3,-4(1)    Store variable result
* EXPRESSION
* CALL output
104:  ST    1,-5(1)    Store fp in ghost frame for output
* TOFF dec: -6
* TOFF dec: -7
* Param 1
105:  LD    3,-4(1)    Load variable result
106:  ST    3,-7(1)    Push parameter
* TOFF dec: -8
* Param end output
```

```
107:  LDA   1,-5(1)    Ghost frame becomes new active frame
108:  LDA   3,1(7)     Return address in ac
109:  JMP   7,-94(7)   CALL output
110:  LDA   3,0(2)     Save the result in ac
* Call end output
* TOFF set: -5
* EXPRESSION
* CALL outnl
111:  ST    1,-5(1)    Store fp in ghost frame for outnl
* TOFF dec: -6
* TOFF dec: -7
* Param end outnl
112:  LDA   1,-5(1)    Ghost frame becomes new active frame
113:  LDA   3,1(7)     Return address in ac
114:  JMP   7,-81(7)   CALL outnl
115:  LDA   3,0(2)     Save the result in ac
* Call end outnl
* TOFF set: -5
* TOFF set: -2
* END COMPOUND
* Add standard closing in case there  is no return statement
116:  LDC   2,0(6)     Set return value to 0
117:  LD    3,-1(1)    Load return address
118:  LD    1,0(1)     Adjust  fp
119:  JMP   7,0(3)     Return
* END FUNCTION  main
0:    JMP   7,119(7)   Jump to init [backpatch]
* INIT
120:  LDA   1,0(0)     set first frame at end of globals
121:  ST    1,0(1)     store old fp (point to self)
* INIT GLOBALS AND STATICS
* END INIT GLOBALS AND STATICS
122:  LDA   3,1(7)     Return address in ac
123:  JMP   7,-43(7)   Jump to main
124:  HALT  0,0,0      DONE!
* END INIT
```