```
======================================================================
 TM 3.0 - A virtual machine for CS445
    plus
 A Description of the Execution Environment for C-

Sep 23, 2013  Robert Heckendorn
University of Idaho
======================================================================

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.


DATA LAYOUT
-----------

8 registers:  0-7
 register 7 is the program counter and is denoted PC below
 All registers are initialized to 0.

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 carot it means
control.
'^M' is control-M etc.  Backslash is understood for '\0', '\t', '\n', '\'' and '\\'.

iMem instruction memory
  each memory location contains both an instruction and a comment.  The
     comment is very useful in debugging!
  iMem is initialized to Halt instructions and the comment: "* initially empty"

dMem data memory
 dMem[0] is initialized with the address of the last element in dMem.
 The rest of dMem is zeroed.

REGISTER ONLY INSTRUCTIONS  (RO instruction format)
-------------------------------------------
HALT x, x, x   stop execution (all registers ignored)
NOP  x, x, x   does nothing but take space (all registers ignored)

IN r           reg[r] <- input integer value of register r from stdin
OUT r          reg[r] -> output integer value of register r to stdout

INB r          reg[r] <- input boolean value of register r from stdin
OUTB r         reg[r] -> output boolean value of register r to stdout

INC r          reg[r] <- input char value of register r from stdin
OUTC r         reg[r] -> output char value of register r to stdout

INS r, s       reg[r] <- input string value of length reg[s] to dMem[reg[r]] from
stdin (it will truncate to fit)
OUTS r, s      reg[r] -> output char value of length reg[s] to dMem[reg[r]] to
stdout (it will truncate)

OUTNL                    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)

REGISTER TO MEMORY INSTRUCTIONS  (RA instruction format)
-----------------------------------------
LDC r, d(X)    reg[r] = d                    (load constant; 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)
LDI r, d(s)    reg[r] = dMem[d + reg[s]]; reg[s]++

ST  r, d(s)    dMem[d + reg[s]] = reg[r]
STI r, d(s)    dMem[d + reg[s]] = reg[r]; reg[s]++
SCI v, d(s)    dMem[d + reg[s]] = v; reg[s]++

JLT r, d(s)    if reg[r]<0 reg[PC] = d + reg[s]
JLE r, d(s)    if reg[r]<=0 reg[PC] = d + reg[s]
JEQ r, d(s)    if reg[r]==0 reg[PC] = d + reg[s]
JNE r, d(s)    if reg[r]!=0 reg[PC] = d + reg[s]
JGE r, d(s)    if reg[r]>=0 reg[PC] = d + reg[s]
JGT r, d(s)    if reg[r]>0 reg[PC] = d + reg[s]


MEMORY TO MEMORY INSTRUCTIONS  (MM instructions in RO format)
-----------------------------------------
MOV r, s, t    dMem[reg[r] + (0..reg[t]-1)] = dMem[reg[s] + (0..reg[t]-1)]
(overlapping source and target is undefined)
STR r, s, t    dMem[reg[r] + (0..reg[t]-1)] = reg[s]         makes reg[t] copies of
reg[s]
CMP r, s, t    reg[5] =  0 if equal or dMem[reg[r] + k] - dMem[reg[s] + k] (for the
first k that is different upto reg[t])
               reg[6] =  location of difference (in this case k) if there is a
difference and reg[t] if they are the same

MEMORY TO MEMORY INSTRUCTIONS  (MM instructions in RA format)
-----------------------------------------
SET r, d(s)    dMem[reg[r] + (0..reg[s]-1)] = d         set immediate: makes reg[s]
copies of d



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)
```

```
 5. goto relative to pc  (d is number of instructions skipped)

   LDA 7, d(7)

 6. NOOP:

   LDA r, 0(r)

 7. save address of following command for return in reg[r]

   LDA r, 1(7)



 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


 TM COMMANDS (v2.4)
 ------------------
 Commands are:
  a(bortLimit <<n>> Maximum number of instructions between halts (default=5000).
  b(reakpoint <<n>> Set a breakpoint for instr n.  No n means clear breakpoints.
  c(lear            Reset simulator for new execution of program
  d(Mem <b <n>>     Print n dMem locations starting at b
                     (n can be negative to count up, defaults to last values used)
  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 starting at b
  l(oad filename    Load filename into memory (default is last file)
  n(ext             Print the next command that will be executed
  p(rint            Toggle printing of total instructions executed ('go' only)
  q(uit             Terminate the simulation
  r(egs             Print the contents of the registers
  s(tep <n>         Execute n (default 1) TM instructions
  t(race            Toggle instruction trace
  u(nprompt)        Unprompted for script input
  x(it              Terminate the simulation
  = <r> <n>         Set register number r to value n (e.g. set the pc)
  (empty line)      does a step like the s command
 Also a # character placed after input will cause TM to halt
```

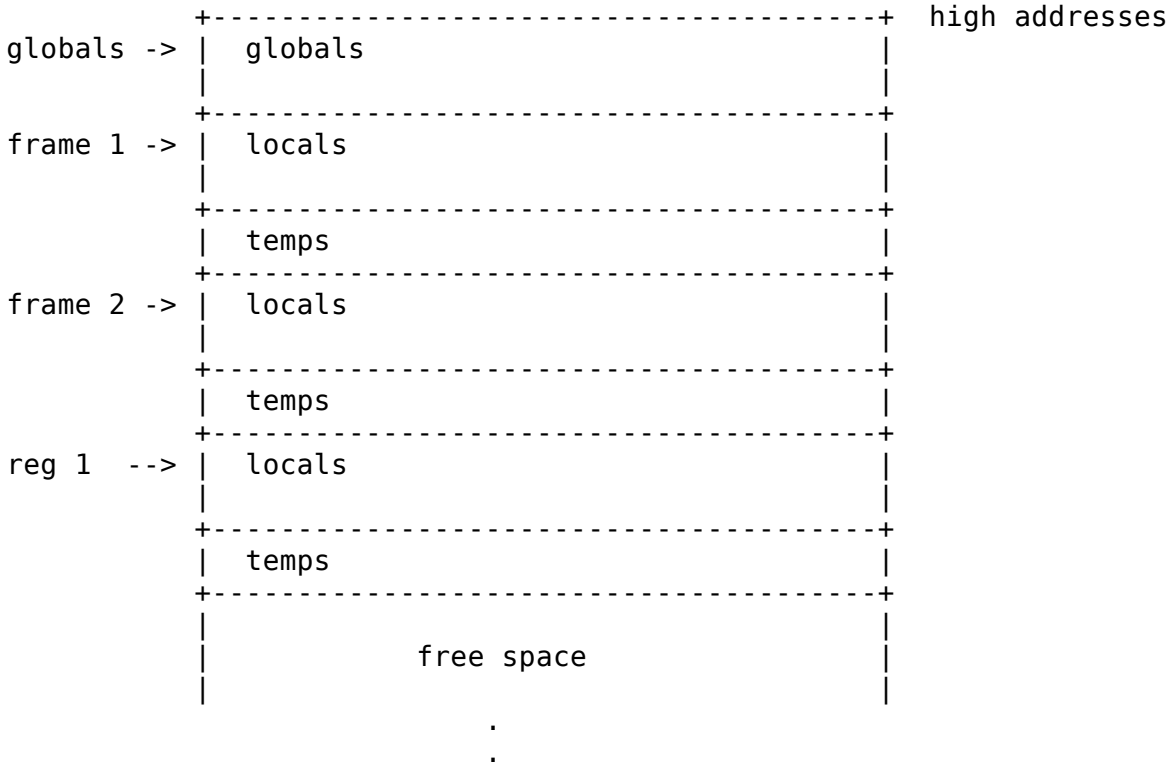
```
   after processing the IN or INB commands (e.g. 34#  or f# )  That way you
   can step after input without setting a breakpoint


 INSTRUCTION INPUT
 -----------------
 Instructions are input via the the load command.
 There commands look like:

 address: cmd r,s,t comment

 or

 address: cmd r,d(s) comment

 or

 * comment

 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


 ========================================================================
  A Description of the Execution Environment for C-
 ========================================================================

 THE TM REGISTERS
 ----------------
 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 frame 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
 ----------------

 Frames for procedures are laid out as follows:

         +----------------------------------------+
 reg1 -> | old frame pointer (old reg1)           |  loc
         +----------------------------------------+
```

```
         | add 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
         +---------------------------------------+
```

* parms are parameters for the function.
* 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 nonconcurrent 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.

## THE STACK LAYOUT

This is how the globals, frames and heap (which we don't have) would be laid out:




```
=====================================================================
 Some Bits of Code to Generate
=====================================================================

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 stack.

toffset - the temp offset is the offset from the frome offset of the
next available temp variable

offset = foffset+toffset and is the current size of the frame

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


PROLOG CODE
-----------
This is the 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:  LD  0, 0(0)        * load gp with top of memory
1:  LDA 1, 0(0)        * set fp to top of memory (no globals)

    * begin call
2:  ST  1, goffset(1)  * store old fp in ghost frame
3:  LDA 1, goffset(1)  * move the fp to the new frame (this is a noop here)
4:  LDA 3, 1(7)        * compute the return address at (skip 1 ahead)
5:  LDA 7, main(7)     * jump to main
    * return here
    * ignore return value
6:  HALT 0, 0, 0       * end of program



CALLING SEQUENCE (caller)  [version 1]
--------------------------
At this point:
reg1 points to the old frame
off in compiler offset to first available space on stack
     relative to the beginning of the frame
```

```
foffset in compiler offset to first available parameter
     relative to top of stack


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

* load the first parameter
LD  4, var1(1)  * load in third temp
ST  4, foffset(3)  * store in parameter space (foffset++)

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

* begin call
ST  1, 0(3)        * store old fp in 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)
LDA 7, func(7)     * call func
* return to here

At this point:
reg1 points to the new frame (top of old local stack)
reg2 has the return value from the function
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 compiler offset to first available space on stack
     relative to the beginning of the frame
foffset in compiler offset to first available parameter
     relative to the beginning of the frame

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)
LDA 7, func(7)  * call func
* return to here

At this point:
reg1 points to the new frame (top of old local stack)
reg2 has the return value from the function
```

```
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 perserve reg3 throughout the
call.

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


RETURN
------

* 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
LDA 7, 0(3)   * jump to old pc



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-06
 int dog(int x)
 {
         int y;
         int z;

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

         return z;
 }

 void main()
 {
         dog(666);
 }



 THE OBJECT CODE
 ---------------
 * C- compiler version C-06
 * Author: Robert B. Heckendorn
 * Backend coauthor: Jorge Williams
 * Apr 1, 2006
 * BEGIN Prolog
   0:     LD  0,0(0)     Set the global pointer
   1:    LDA  1,0(0)     set first frame at end of globals
   2:     ST  1,0(1)     store old fp (point to self)
   3:    LDA  3,1(7)     Return address in ac
   5:   HALT  0,0,0      DONE!
 * END Prolog
 * BEGIN function input
   6:     ST  3,-1(1)    Store return address
   7:     IN  2,2,2      Grab int input
   8:     LD  3,-1(1)    Load return address
   9:     LD  1,0(1)     Adjust fp
  10:    LDA  7,0(3)     Return
 * END of function input
 * BEGIN function output
  11:     ST  3,-1(1)    Store return address
  12:     LD  3,-2(1)    Load parameter
  13:    OUT  3,3,3      Output integer
  14:    LDC  2,0(6)     Set return to 0
  15:     LD  3,-1(1)    Load return address
  16:     LD  1,0(1)     Adjust fp
  17:    LDA  7,0(3)     Return
 * END of function output
```

```
* BEGIN function inputb
 18:     ST  3,-1(1)    Store return address
 19:    INB  2,2,2      Grab bool input
 20:     LD  3,-1(1)    Load return address
 21:     LD  1,0(1)     Adjust fp
 22:    LDA  7,0(3)     Return
* END of function inputb
* BEGIN function outputb
 23:     ST  3,-1(1)    Store return address
 24:     LD  3,-2(1)    Load parameter
 25:   OUTB  3,3,3      Output bool
 26:    LDC  2,0(6)     Set return to 0
 27:     LD  3,-1(1)    Load return address
 28:     LD  1,0(1)     Adjust fp
 29:    LDA  7,0(3)     Return
* END of function outputb
* BEGIN function outnl
 30:     ST  3,-1(1)    Store return address
 31:  OUTNL  3,3,3      Output a newline
 32:     LD  3,-1(1)    Load return address
 33:     LD  1,0(1)     Adjust fp
 34:    LDA  7,0(3)     Return
* END of function outnl
* BEGIN function dog
 35:     ST  3,-1(1)    Store return address. BEGIN FUNC: dog
* BEGIN compound statement
*
 36:     LD  3,-2(1)    Load variable x
 37:     ST  3,-5(1)    Save left side
 38:    LDC  3,111(6)   Load constant
 39:     LD  4,-5(1)    Load left into ac1
 40:    MUL  3,4,3      Op *
 41:     ST  3,-5(1)    Save left side
 42:    LDC  3,222(6)   Load constant
 43:     LD  4,-5(1)    Load left into ac1
 44:    ADD  3,4,3      Op +
 45:     ST  3,-3(1)    Store variable y
*
 46:     LD  3,-3(1)    Load variable y
 47:     ST  3,-4(1)    Store variable z
* RETURN
 48:     LD  3,-4(1)    Load variable z
 49:    LDA  2,0(3)     Copy result to rt register
 50:     LD  3,-1(1)    Load return address
 51:     LD  1,0(1)     Adjust fp
 52:    LDA  7,0(3)     Return
* END compound statement
* Add standard closing in case there is no return statement
 53:    LDC  2,0(6)     Set return value to 0
 54:     LD  3,-1(1)    Load return address
 55:     LD  1,0(1)     Adjust fp
 56:    LDA  7,0(3)     Return
* END of function dog
* BEGIN function main
  4:    LDA  7,52(7)    Jump to main
 57:     ST  3,-1(1)    Store return address. BEGIN FUNC: main
* BEGIN compound statement
*
```

```
  58:     ST  1,-2(1)    Store old fp in ghost frame
  59:    LDC  3,666(6)   Load constant
  60:     ST  3,-4(1)    Store parameter
  61:    LDA  1,-2(1)    Load address of new frame
  62:    LDA  3,1(7)     Return address in ac
  63:    LDA  7,-29(7)   call dog
  64:    LDA  3,0(2)     Save the result in ac
* END compound statement
* 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
* END of function main


=====================================================================
 EXAMPLE 2: A Simple C- Program Compiled
=====================================================================

THE CODE
--------
// C-06
// 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);
}

void main()
{
    int x, y;

    x = input();
    y = input();
    output(gcd(x, y));
}



THE OBJECT CODE
---------------

* C- compiler version C-06
* Author: Robert B. Heckendorn
* Backend coauthor: Jorge Williams
* Apr 1, 2006
* BEGIN Prolog
  0:     LD  0,0(0)     Set the global pointer
  1:    LDA  1,0(0)     set first frame at end of globals
  2:     ST  1,0(1)     store old fp (point to self)
  3:    LDA  3,1(7)     Return address in ac
  5:   HALT  0,0,0      DONE!
```

```
 * END Prolog
 * BEGIN function input
   6:     ST  3,-1(1)    Store return address
   7:     IN  2,2,2      Grab int input
   8:     LD  3,-1(1)    Load return address
   9:     LD  1,0(1)     Adjust fp
  10:    LDA  7,0(3)     Return
 * END of function input
 * BEGIN function output
  11:     ST  3,-1(1)    Store return address
  12:     LD  3,-2(1)    Load parameter
  13:    OUT  3,3,3      Output integer
  14:    LDC  2,0(6)     Set return to 0
  15:     LD  3,-1(1)    Load return address
  16:     LD  1,0(1)     Adjust fp
  17:    LDA  7,0(3)     Return
 * END of function output
 * BEGIN function inputb
  18:     ST  3,-1(1)    Store return address
  19:    INB  2,2,2      Grab bool input
  20:     LD  3,-1(1)    Load return address
  21:     LD  1,0(1)     Adjust fp
  22:    LDA  7,0(3)     Return
 * END of function inputb
 * BEGIN function outputb
  23:     ST  3,-1(1)    Store return address
  24:     LD  3,-2(1)    Load parameter
  25:   OUTB  3,3,3      Output bool
  26:    LDC  2,0(6)     Set return to 0
  27:     LD  3,-1(1)    Load return address
  28:     LD  1,0(1)     Adjust fp
  29:    LDA  7,0(3)     Return
 * END of function outputb
 * BEGIN function outnl
  30:     ST  3,-1(1)    Store return address
  31:  OUTNL  3,3,3      Output a newline
  32:     LD  3,-1(1)    Load return address
  33:     LD  1,0(1)     Adjust fp
  34:    LDA  7,0(3)     Return
 * END of function outnl
 * BEGIN function gcd
  35:     ST  3,-1(1)    Store return address. BEGIN FUNC: gcd
 * BEGIN compound statement
 * IF
  36:     LD  3,-3(1)    Load variable v
  37:     ST  3,-4(1)    Save left side
  38:    LDC  3,0(6)     Load constant
  39:     LD  4,-4(1)    Load left into ac1
  40:    SUB  4,4,3      Op ==
  41:    LDC  3,1(6)     True case
  42:    JEQ  4,1(7)     Jump if true
  43:    LDC  3,0(6)     False case
  44:    LDC  4,1(6)     Load constant 1
  45:    SUB  3,3,4      If cond check
  46:    JGE  3,1(7)     Jump to then part
 * THEN
 * RETURN
  48:     LD  3,-2(1)    Load variable u
```

```
 49:    LDA  2,0(3)     Copy result to rt register
 50:     LD  3,-1(1)    Load return address
 51:     LD  1,0(1)     Adjust fp
 52:    LDA  7,0(3)     Return
* ELSE
 47:    LDA  7,6(7)     Jump around the THEN
* RETURN
 54:     ST  1,-4(1)    Store old fp in ghost frame
 55:     LD  3,-3(1)    Load variable v
 56:     ST  3,-6(1)    Store parameter
 57:     LD  3,-2(1)    Load variable u
 58:     ST  3,-7(1)    Save left side
 59:     LD  3,-2(1)    Load variable u
 60:     ST  3,-8(1)    Save left side
 61:     LD  3,-3(1)    Load variable v
 62:     LD  4,-8(1)    Load left into ac1
 63:    DIV  3,4,3      Op /
 64:     ST  3,-8(1)    Save left side
 65:     LD  3,-3(1)    Load variable v
 66:     LD  4,-8(1)    Load left into ac1
 67:    MUL  3,4,3      Op *
 68:     LD  4,-7(1)    Load left into ac1
 69:    SUB  3,4,3      Op -
 70:     ST  3,-7(1)    Store parameter
 71:    LDA  1,-4(1)    Load address of new frame
 72:    LDA  3,1(7)     Return address in ac
 73:    LDA  7,-39(7)   call gcd
 74:    LDA  3,0(2)     Save the result in ac
 75:    LDA  2,0(3)     Copy result to rt register
 76:     LD  3,-1(1)    Load return address
 77:     LD  1,0(1)     Adjust fp
 78:    LDA  7,0(3)     Return
 53:    LDA  7,25(7)    Jump around the ELSE
* ENDIF
* END compound statement
* Add standard closing in case there is no return statement
 79:    LDC  2,0(6)     Set return value to 0
 80:     LD  3,-1(1)    Load return address
 81:     LD  1,0(1)     Adjust fp
 82:    LDA  7,0(3)     Return
* END of function gcd
* BEGIN function main
  4:    LDA  7,78(7)    Jump to main
 83:     ST  3,-1(1)    Store return address. BEGIN FUNC: main
* BEGIN compound statement
*
 84:     ST  1,-4(1)    Store old fp in ghost frame
 85:    LDA  1,-4(1)    Load address of new frame
 86:    LDA  3,1(7)     Return address in ac
 87:    LDA  7,-82(7)   call input
 88:    LDA  3,0(2)     Save the result in ac
 89:     ST  3,-2(1)    Store variable x
*
 90:     ST  1,-4(1)    Store old fp in ghost frame
 91:    LDA  1,-4(1)    Load address of new frame
 92:    LDA  3,1(7)     Return address in ac
 93:    LDA  7,-88(7)   call input
 94:    LDA  3,0(2)     Save the result in ac
```

```
  95:     ST  3,-3(1)    Store variable y
 *
  96:     ST  1,-4(1)    Store old fp in ghost frame
  97:     ST  1,-6(1)    Store old fp in ghost frame
  98:     LD  3,-2(1)    Load variable x
  99:     ST  3,-8(1)    Store parameter
 100:     LD  3,-3(1)    Load variable y
 101:     ST  3,-9(1)    Store parameter
 102:    LDA  1,-6(1)    Load address of new frame
 103:    LDA  3,1(7)     Return address in ac
 104:    LDA  7,-70(7)   call gcd
 105:    LDA  3,0(2)     Save the result in ac
 106:     ST  3,-6(1)    Store parameter
 107:    LDA  1,-4(1)    Load address of new frame
 108:    LDA  3,1(7)     Return address in ac
 109:    LDA  7,-99(7)   call output
 110:    LDA  3,0(2)     Save the result in ac
 * END compound statement
 * Add standard closing in case there is no return statement
 111:    LDC  2,0(6)     Set return value to 0
 112:     LD  3,-1(1)    Load return address
 113:     LD  1,0(1)     Adjust fp
 114:    LDA  7,0(3)     Return
 * END of function main
```