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.

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

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

FORMAT OF TM file is lines of the form:
* <comment> a general full line comment
addr <instruction> <comment> set INSTRUCTION MEMORY at addr to this instruction
addr LIT <value> set DATA MEMORY at addr to this value

LITERAL INSTRUCTIONS (data memory)
LIT 666  load into data memory the single "word" value given at the address.
LIT 'x'  load into data memory the single "word" value given at the address.
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.

REGISTER ONLY INSTRUCTIONS (RO instruction format) (instruction memory)
--------------------------------------------
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, reg[r] <- input char value of register r from stdin
OUT r, X, reg[r] -> output integer value of register r to stdout
OUTB r, X, reg[r] -> output boolean value of register r to stdout
OUTC r, X, reg[r] -> output char value of register r to stdout
OUTNL 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] (always returns the NONNEGATIVE modulus of reg[s] % 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; X ignored,)

REGISTER TO MEMORY INSTRUCTIONS (RA instruction format)
--------------------------------------------
LDC r, c(X) reg[r] = c (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)
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)

TEST INSTRUCTIONS (RO instruction format) (instruction memory)
--------------------------------------------
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)

--------------------------------------------
MOV r, s, t  dMem[reg[r] - (0..reg[t]-1)] = dMem[reg[s] - (0..reg[t]-1)] (overlapping source and target is undefined)

SET r, s, t  dMem[reg[r] - (0..reg[t]-1)] = reg[s]  makes reg[t] copies of reg[s]

CO r, s, t  reg[5] = dMem[reg[r] + k] (for the first k that yields a diff or the last tested if no diff)

reg[6] = 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

CGA r, s, t  reg[5] = reg[r] + k  (for the first k that yields a diff at that address or the last tested if no diff)

reg[6] = 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

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)

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.

TM version 4.1
-------------

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
go 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 TM
r(egs Print the contents of the registers
s(tep <<n>> Execute n (default 1) TM instructions
trace Toggle instruction tracing (printing) during execution
u(nprompt) Unprompted for script input
v Print the version information
x(it Terminate TM
= <<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 TM to halt
after processing the IN or INB commands (e.g. 34# or f#)

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

address: cmd r,s,t comment

or

address: cmd r,d(s) comment

or

* comment

or

address: LIT value  comment

value can be integer or char or string

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

60:  LIT "dogs"  A literal stored at data memory locations 61..57

70:  LIT 'x'  A literal stored at data memory location 70

71:  LIT 666  A literal stored at data memory location 71

A note about string literals:  60: LIT "dogs"
looks like:

63:  0    unused

62:  0    unused

61:  4    readOnly <-- size

60:  100    'd'    readOnly <-- address given in LIT

59:  111    'o'

58:  103    'g'

57:  115    's'

56:  0    unused

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

5
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 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

Location 0 in the data space is initialized with the address of the last element in data space. This 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

```
+---------------------------------------+ HIGH ADDRESSES
  reg 0 --> | global and static variables |
  |                     +-------------------------|
  +---------------------------------------+  
  frame 1 --> | local variables and parameters | |stack frame 1
  |                     +-------------------------+  |
  | temps                +-------------------------+  |
  +---------------------------------------+  
  frame 2 --> | local variables and parameters | |stack frame 2
  |                     +-------------------------+  |
  | temps                +-------------------------+  |
  +---------------------------------------+  
  reg 1 --> | local variables and parameters | |current stack fame
  |                     +-------------------------+  |
  | temps                +-------------------------+  |
  +---------------------------------------+  |
  | free space            +-------------------------+  |
  |                     +-------------------------+  |
  | (heap would go here)  +-------------------------+  |
```
We do not currently have a heap or garbage collection.

THE INSTRUCTION SPACE LAYOUT

Instructions are loaded 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

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

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: LDA 7,XXX(7) Jump to init [backpatch]

( body of code including main goes here )

* INIT
52: LD 0,0(0) Set the global pointer
* INIT GLOBALS AND STATICS

( code to init 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
56: LDA 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 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

* 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 (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
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)
LDA 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 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

(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)
LDA 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
LDA 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
-----------------
* C- compiler version C-F15
* Built: Oct 14, 2015
* Author: Robert B. Heckendorn
* File compiled: tmSample.c-

* 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:  LDA 7,0(3) Return
* END FUNCTION input

* FUNCTION output
6:  ST 3,-1(1)  Store return address
7:  LD 3,-2(1) Load parameter
8: OUT 3,3,3  Output integer
9: LDC 2,0(6) Set return to 0
10: LD 3,-1(1) Load return address
11: LD 1,0(1) Adjust fp
12: LDA 7,0(3) Return
* END FUNCTION output
* FUNCTION inputb
13: ST 3,-1(1) Store return address
14: INB 2,2,2 Grab bool input
15: LD 3,-1(1) Load return address
16: LD 1,0(1) Adjust fp
17: LDA 7,0(3) Return
* END FUNCTION inputb
* FUNCTION outputb
18: ST 3,-1(1) Store return address
19: LD 3,-2(1) Load parameter
20: OUTB 3,3,3 Output bool
21: LDC 2,0(6) Set return to 0
22: LD 3,-1(1) Load return address
23: LD 1,0(1) Adjust fp
24: LDA 7,0(3) Return
* END FUNCTION outputb
* FUNCTION inputc
25: ST 3,-1(1) Store return address
26: INC 2,2,2 Grab char input
27: LD 3,-1(1) Load return address
28: LD 1,0(1) Adjust fp
29: LDA 7,0(3) Return
* END FUNCTION inputc
* FUNCTION outputc
30: ST 3,-1(1) Store return address
31: LD 3,-2(1) Load parameter
32: OUTC 3,3,3 Output char
33: LDC 2,0(6) Set return to 0
34: LD 3,-1(1) Load return address
35: LD 1,0(1) Adjust fp
36: LDA 7,0(3) Return
* END FUNCTION outputc
* FUNCTION outnl
37: ST 3,-1(1) Store return address
38: OUTNL 3,3,3 Output a newline
39: LD 3,-1(1) Load return address
40: LD 1,0(1) Adjust fp
41: LDA 7,0(3) Return
* END FUNCTION outnl
* FUNCTION dog
42: ST 3,-1(1) Store return address.
* COMPOUND
* EXPRESSION
43: LD 3,-2(1) Load variable x
44: ST 3,-5(1) Save left side
45: LDC 3,111(6) Load constant
46: LD 4,-5(1) Load left into ac1
47: MUL 3,4,3 Op *
48: ST 3,-5(1)  Save left side
49: LDC 3,222(6)  Load constant
50: LD 4,-5(1)  Load left into ac1
51: ADD 3,4,3  Op +
52: ST 3,-3(1)  Store variable y
* EXPRESSION
53: LD 3,-3(1)  Load variable y
54: ST 3,-4(1)  Store variable z
* RETURN
55: LD 3,-4(1)  Load variable z
56: LDA 2,0(3)  Copy result to rt register
57: LD 3,-1(1)  Load return address
58: LD 1,0(1)  Adjust fp
59: LDA 7,0(3)  Return
* END COMPOUND
* Add standard closing in case there is no return statement
60: LDC 2,0(6)  Set return value to 0
61: LD 3,-1(1)  Load return address
62: LD 1,0(1)  Adjust fp
63: LDA 7,0(3)  Return
* END FUNCTION dog
* FUNCTION main
64: ST 3,-1(1)  Store return address.
* COMPOUND
* EXPRESSION
* Begin call to output
65: ST 1,-2(1)  Store old fp in ghost frame
* Load param 1
* Begin call to dog
66: ST 1,-4(1)  Store old fp in ghost frame
* Load param 1
67: LDC 3,666(6)  Load constant
68: ST 3,-6(1)  Store parameter
* Jump to dog
69: LDA 1,-4(1)  Load address of new frame
70: LDA 3,1(7)  Return address in ac
71: LDA 7,-30(7)  CALL dog
72: LDA 3,0(2)  Save the result in ac
* End call to dog
73: ST 3,-4(1)  Store parameter
* Jump to output
74: LDA 1,-2(1)  Load address of new frame
75: LDA 3,1(7)  Return address in ac
76: LDA 7,-71(7)  CALL output
77: LDA 3,0(2)  Save the result in ac
* End call to output
* EXPRESSION
* Begin call to outnl
78: ST 1,-2(1)  Store old fp in ghost frame
* Jump to outnl
79: LDA 1,-2(1)  Load address of new frame
80: LDA 3,1(7)  Return address in ac
81: LDA 7,-45(7)  CALL outnl
82: LDA 3,0(2)  Save the result in ac
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();
}
* C- compiler version C-F15
* Built: Oct 14, 2015
* Author: Robert B. Heckendorn
* File compiled: tmSample2.c-

* 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: LDA 7,0(3) Return
* END FUNCTION input

* FUNCTION output
6: ST 3,-1(1) Store return address
7: LD 3,-2(1) Load parameter
8: OUT 3,3,3 Output integer
9: LDC 2,0(6) Set return to 0
10: LD 3,-1(1) Load return address
11: LD 1,0(1) Adjust fp
12: LDA 7,0(3) Return
* END FUNCTION output

* FUNCTION inputb
13: ST 3,-1(1) Store return address
14: INB 2,2,2 Grab bool input
15: LD 3,-1(1) Load return address
16: LD 1,0(1) Adjust fp
17: LDA 7,0(3) Return
* END FUNCTION inputb

* FUNCTION outputb
18: ST 3,-1(1) Store return address
19: LD 3,-2(1) Load parameter
20: OUTB 3,3,3 Output bool
21: LDC 2,0(6) Set return to 0
22: LD 3,-1(1) Load return address
23: LD 1,0(1) Adjust fp
24: LDA 7,0(3) Return
* END FUNCTION outputb

* FUNCTION inputc
25: ST 3,-1(1) Store return address
26: INC 2,2,2 Grab char input
27: LD 3,-1(1) Load return address
28: LD 1,0(1) Adjust fp
29: LDA 7,0(3) Return
* END FUNCTION inputc

* FUNCTION outputc
30: ST 3,-1(1) Store return address
31: LD 3,-2(1) Load parameter
32: OUTC 3,3,3 Output char
33: LDC 2,0(6) Set return to 0
34: LD 3,-1(1) Load return address
35: LD 1,0(1) Adjust fp
36: LDA 7,0(3) Return
* END FUNCTION outputc

16
* FUNCTION outnl
  37:  ST 3,-1(1) Store return address
  38:  OUTNL 3,3,3 Output a newline
  39:  LD 3,-1(1) Load return address
  40:  LD 1,0(1) Adjust fp
  41:  LDA 7,0(3) Return
* END FUNCTION outnl

* FUNCTION gcd
  42:  ST 3,-1(1) Store return address.
  * COMPOUND
  * IF
  43:  LD 3,-3(1) Load variable v
  44:  ST 3,-4(1) Save left side
  45:  LDC 3,0(6) Load constant
  46:  LD 4,-4(1) Load left into ac1
  47:  TEQ 3,4,3 Op ==
  * THEN
  * RETURN
  49:  LD 3,-2(1) Load variable u
  50:  LDA 2,0(3) Copy result to rt register
  51:  LD 3,-1(1) Load return address
  52:  LD 1,0(1) Adjust fp
  53:  LDA 7,0(3) Return
  48:  JZR 3,6(7) Jump around the THEN if false [backpatch]
  * ELSE
  * RETURN
  * Begin call to gcd
  55:  ST 1,-4(1) Store old fp in ghost frame
  * Load param 1
  56:  LD 3,-3(1) Load variable v
  57:  ST 3,-6(1) Store parameter
  * Load param 2
  58:  LD 3,-2(1) Load variable u
  59:  ST 3,-7(1) Save left side
  60:  LD 3,-2(1) Load variable u
  61:  ST 3,-8(1) Save left side
  62:  LD 3,-3(1) Load variable v
  63:  LD 4,-8(1) Load left into ac1
  64:  DIV 3,4,3 Op /
  65:  ST 3,-8(1) Save left side
  66:  LD 3,-3(1) Load variable v
  67:  LD 4,-8(1) Load left into ac1
  68:  MUL 3,4,3 Op *
  69:  LD 4,-7(1) Load left into ac1
  70:  SUB 3,4,3 Op -
  71:  ST 3,-7(1) Store parameter
  * Jump to gcd
  72:  LDA 1,-4(1) Load address of new frame
  73:  LDA 3,1(7) Return address in ac
  74:  LDA 7,-33(7) CALL gcd
  75:  LDA 3,0(2) Save the result in ac
  * End call to gcd
  76:  LDA 2,0(3) Copy result to rt register
  77:  LD 3,-1(1) Load return address

17
78: LD 1,0(1) Adjust fp
79: LDA 7,0(3) Return
54: LDA 7,25(7) Jump around the ELSE [backpatch]
ENDIF
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: LDA 7,0(3) Return
END FUNCTION gcd

FUNCTION main
84: ST 3,-1(1) Store return address.

* COMPOUND
* EXPRESSION
* Begin call to input
85: ST 1,-5(1) Store old fp in ghost frame
* Jump to input
86: LDA 1,-5(1) Load address of new frame
87: LDA 3,1(7) Return address in ac
88: LDA 7,-88(7) CALL input
89: LDA 3,0(2) Save the result in ac
* End call to input
90: ST 3,-2(1) Store variable x

* EXPRESSION
* Begin call to input
91: ST 1,-5(1) Store old fp in ghost frame
* Jump to input
92: LDA 1,-5(1) Load address of new frame
93: LDA 3,1(7) Return address in ac
94: LDA 7,-94(7) CALL input
95: LDA 3,0(2) Save the result in ac
* End call to input
96: ST 3,-3(1) Store variable y

* EXPRESSION
* Begin call to gcd
97: ST 1,-5(1) Store old fp in ghost frame
* Load param 1
98: LD 3,-2(1) Load variable x
99: ST 3,-7(1) Store parameter
* Load param 2
100: LD 3,-3(1) Load variable y
101: ST 3,-8(1) Store parameter
* Jump to gcd
102: LDA 1,-5(1) Load address of new frame
103: LDA 3,1(7) Return address in ac
104: LDA 7,-63(7) CALL gcd
105: LDA 3,0(2) Save the result in ac
* End call to gcd
106: ST 3,-4(1) Store variable result

* EXPRESSION
* Begin call to output
107: ST 1,-5(1) Store old fp in ghost frame
* Load param 1
108: LD 3,-4(1) Load variable result  
109: ST 3,-7(1) Store parameter  
* Jump to output  
110: LDA 1,-5(1) Load address of new frame  
111: LDA 3,1(7) Return address in ac  
112: LDA 7,-107(7) CALL output  
113: LDA 3,0(2) Save the result in ac  
* End call to output  
* EXPRESSION  
* Begin call to outnl  
114: ST 1,-5(1) Store old fp in ghost frame  
* Jump to outnl  
115: LDA 1,-5(1) Load address of new frame  
116: LDA 3,1(7) Return address in ac  
117: LDA 7,-81(7) CALL outnl  
118: LDA 3,0(2) Save the result in ac  
* End call to outnl  
* END COMPOUND  
* Add standard closing in case there is no return statement  
119: LDC 2,0(6) Set return value to 0  
120: LD 3,-1(1) Load return address  
121: LD 1,0(1) Adjust fp  
122: LDA 7,0(3) Return  
* END FUNCTION main  
0: LDA 7,122(7) Jump to init [backpatch]  
* INIT  
123: LD 0,0(0) Set the global pointer  
* INIT GLOBALS AND STATICS  
* END INIT GLOBALS AND STATICS  
124: LDA 1,0(0) set first frame at end of globals  
125: ST 1,0(1) store old fp (point to self)  
126: LDA 3,1(7) Return address in ac  
127: LDA 7,-44(7) Jump to main  
128: HALT 0,0,0 DONE!  
* END INIT