Project 1--CDA 3101 (Spring 2014)

Worth: 100 points (10% of course grade)

Assigned: Friday, Jan 24, 2014

Due: 1:25 pm, Monday, Feb 24, 2014

1. Purpose

This project is intended to help you understand the instructions of a very

simple assembly language and how to assemble programs into machine language.

2. Problem

This project has three parts. In the first part, you will write a program to

take an assembly-language program and produce the corresponding machine

language. In the second part, you will write a behavioral simulator for the

resulting machine code. In the third part, you will write a short

assembly-language program to multiply two numbers.

3. LC3101 Instruction-Set Architecture

For this project, you will be developing a simulator and assembler for the

LC3101 (Little Computer, used in CDA 3101). The LC3101 is very simple, but

it is general enough to solve complex problems. For this project, you will

only need to know the instruction set and instruction format of the LC3101.

The LC3101 is an 8-register, 32-bit computer. All addresses are

word-addresses (unline MIPS which is byte-addressed). The LC3101 has 65536

words of memory. By assembly-language convention, register 0 will always

contain 0 (i.e. the machine will not enforce this, but no assembly-language

program should ever change register 0 from its initial value of 0).

There are 3 instruction formats (bit 0 is the least-significant bit). Bits

31-25 are unused for all instructions, and should always be 0.

R-type instructions (add, nand):

bits 24-22: opcode

bits 21-19: reg A

bits 18-16: reg B

bits 15-3: unused (should all be 0)

bits 2-0: destReg

I-type instructions (lw, sw, beq):

bits 24-22: opcode

bits 21-19: reg A

bits 18-16: reg B

bits 15-0: offsetField (16-bit, range of -32768 to 32767)

O-type instructions (halt, noop):

bits 24-22: opcode

bits 21-0: unused (should all be 0)

-------------------------------------------------------------------------------

Table 1: Description of Machine Instructions

-------------------------------------------------------------------------------

Assembly language Opcode in binary Action

name for instruction (bits 24, 23, 22)

-------------------------------------------------------------------------------

add (R-type format) 000 add contents of regA with

contents of regB, store

results in destReg.

nand (R-type format) 001 nand contents of regA with

contents of regB, store

results in destReg.

lw (I-type format) 010 load regB from memory. Memory

address is formed by adding

offsetField with the contents of

regA.

sw (I-type format) 011 store regB into memory. Memory

address is formed by adding

offsetField with the contents of

regA.

beq (I-type format) 100 if the contents of regA and

regB are the same, then branch

to the address PC+1+offsetField,

where PC is the address of the

beq instruction.

cmov (R-type) 101 copy the value regA into destReg

if the contents of regB != 0

halt (O-type format) 110 increment the PC (as with all

instructions), then halt the

machine (let the simulator

notice that the machine

halted).

noop (O-type format) 111 do nothing except increment PC.

-------------------------------------------------------------------------------

4. LC3101 Assembly Language and Assembler (40%)

The first part of this project is to write a program to take an

assembly-language program and translate it into machine language. You will

translate assembly-language names for instructions, such as beq, into their

numeric equivalent (e.g. 100), and you will translate symbolic names for

addresses into numeric values. The final output will be a series of 32-bit

instructions (instruction bits 31-25 are always 0).

The format for a line of assembly code is:

label instruction field0 field1 field2 comments

The leftmost field on a line is the label field. Valid labels contain a

maximum of 6 characters and can consist of letters and numbers (but must start

with a letter). The label is optional (the white space following the label

field is required). Labels make it much easier to write assembly-language

programs, since otherwise you would need to modify all address fields each time

you added a line to your assembly-language program!

After the optional label is white space which consists of any number of space

or tab characters. The writespace is followed by the instruction field,

where the instruction can be any of the assembly-language instruction names

listed in the above table. After more white space comes a series of fields.

All fields are given as decimal numbers or labels. The number of fields

depends on the instruction, and unused fields should be ignored (treat them

like comments).

R-type instructions (add, nand) instructions require 3 fields: field0

is regA, field1 is regB, and field2 is destReg.

I-type instructions (lw, sw, beq) require 3 fields: field0 is regA, field1

is regB, and field2 is either a numeric value for offsetField or a symbolic

address. Numeric offsetFields can be positive or negative; symbolic

addresses are discussed below.

O-type instructions (noop and halt) require no fields.

Symbolic addresses refer to labels. For lw or sw instructions, the assembler

should compute offsetField to be equal to the address of the label. This could

be used with a zero base register to refer to the label, or could be used with

a non-zero base register to index into an array starting at the label. For beq

instructions, the assembler should translate the label into the numeric

offsetField needed to branch to that label.

After the last used field comes more white space, then any comments. The

comment field ends at the end of a line. Comments are vital to creating

understandable assembly-language programs, because the instructions themselves

are rather cryptic.

In addition to LC3101 instructions, an assembly-language program may contain

directions for the assembler. The only assembler directive we will use is .fill

(note the leading period). .fill tells the assembler to put a number into the

place where the instruction would normally be stored. .fill instructions use

one field, which can be either a numeric value or a symbolic address. For

example, ".fill 32" puts the value 32 where the instruction would normally be

stored. .fill with a symbolic address will store the address of the label.

In the example below, ".fill start" will store the value 2, because the label

"start" is at address 2.

The assembler should make two passes over the assembly-language program. In the

first pass, it will calculate the address for every symbolic label. Assume

that the first instruction is at address 0. In the second pass, it will

generate a machine-language instruction (in decimal) for each line of assembly

language. For example, here is an assembly-language program (that counts down

from 5, stopping when it hits 0).

lw 0 1 five load reg1 with 5 (uses symbolic address)

lw 1 2 3 load reg2 with -1 (uses numeric address)

start add 1 2 1 decrement reg1

beq 0 1 2 goto end of program when reg1==0

beq 0 0 start go back to the beginning of the loop

noop

done halt end of program

five .fill 5

neg1 .fill -1

stAddr .fill start will contain the address of start (2)

And here is the corresponding machine language:

(address 0): 8454151 (hex 0x810007)

(address 1): 9043971 (hex 0x8a0003)

(address 2): 655361 (hex 0xa0001)

(address 3): 16842754 (hex 0x1010002)

(address 4): 16842749 (hex 0x100fffd)

(address 5): 29360128 (hex 0x1c00000)

(address 6): 25165824 (hex 0x1800000)

(address 7): 5 (hex 0x5)

(address 8): -1 (hex 0xffffffff)

(address 9): 2 (hex 0x2)

Be sure you understand how the above assembly-language program got translated

to machine language.

Since your programs will always start at address 0, your program should only

output the contents, not the addresses.

8454151

9043971

655361

16842754

16842749

29360128

25165824

5

-1

2

4.1. Running Your Assembler

Write your program to take two command-line arguments. The first argument is

the file name where the assembly-language program is stored, and the second

argument is the file name where the output (the machine-code) is written.

For example, with a program name of "assemble", an assembly-language program

in "program.as", the following would generate a machine-code file "program.mc":

assemble program.as program.mc

Note that the format for running the command must use command-line arguments

for the file names (rather than standard input and standard output). Your

program should store only the list of decimal numbers in the machine-code

file, one instruction per line. Any deviation from this format (e.g. extra

spaces or empty lines) will render your machine-code file ungradable. Any

other output that you want the program to generate (e.g. debugging output) can

be printed to standard output.

4.2. Error Checking

Your assembler should catch the following errors in the assembly-language

program: use of undefined labels, duplicate labels, offsetFields that don't fit

in 16 bits, and unrecognized opcodes. Your assembler should exit(1) if it

detects an error and exit(0) if it finishes without detecting any errors. Your

assembler should NOT catch simulation-time errors, i.e. errors that would occur

at the time the assembly-language program executes (e.g. branching to address

-1, infinite loops, etc.).

4.3. Test Cases

An integral (and graded) part of writing your assembler will be to write a

suite of test cases to validate any LC3101 assembler. This is common practice

in the real world--software companies maintain a suite of test cases for their

programs and use this suite to check the program's correctness after a change.

Writing a comprehensive suite of test cases will deepen your understanding of

the project specification and your program, and it will help you a lot as you

debug your program.

The test cases for the assembler part of this project will be short

assembly-language programs that serve as input to an assembler. You will

submit your suite of test cases together with your assembler, and we will grade

your test suite according to how thoroughly it exercises an assembler. Each

test case may be at most 50 lines long, and your test suite may contain up to

20 test cases. These limits are much larger than needed for full credit (the

solution test suite is composed of 5 test cases, each < 10 lines long). See

Section 7 for how your test suite will be graded.

Hints: the example assembly-language program above is a good case to include

in your test suite, though you'll need to write more test cases to get full

credit. Remember to create some test cases that test the ability of an

assembler to check for the errors in Section 4.2.

4.4. Assembler Hints

Since offsetField is a 2's complement number, it can only store numbers ranging

from -32768 to 32767. For symbolic addresses, your assembler will compute

offsetField so that the instruction refers to the correct label.

Remember that offsetField is only an 16-bit 2's complement number. Since most

machines you run your assembler on have 32-bit or 64-bit integers, you will

have to truncate all but the lowest 16 bits for negative values of offsetField.

5. Behavioral Simulator (40%)

The second part of this assignment is to write a program that can simulate any

legal LC3101 machine-code program. The input for this part will be the

machine-code file that you created with your assembler. With a program name

of "simulate" and a machine-code file of "program.mc", your program should be

run as follows:

simulate program.mc > output

This directs all print statements to the file "output".

The simulator should begin by initializing all registers and the program

counter to 0. The simulator will then simulate the program until the program

executes a halt.

The simulator should call printState (included below) before executing each

instruction and once just before exiting the program. This function prints the

current state of the machine (program counter, registers, memory). printState

will print the memory contents for memory locations defined in the machine-code

file (addresses 0-9 in the Section 4 example).

5.1 Test Cases

As with the assembler, you will write a suite of test cases to validate the

LC3101 simulator.

The test cases for the simulator part of this project will be short

assembly-language programs that, after being assembled into machine code, serve

as input to a simulator. You will submit your suite of test cases together

with your simulator, and we will grade your test suite according to how

thoroughly it exercises an LC3101 simulator. Each test case may execute at

most 200 instructions on a correct simulator, and your test suite may contain

up to 20 test cases. These limits are much larger than needed for full credit

(the solution test suite is composed of a couple test cases, each executing

less than 40 instructions). See Section 7 for how your test suite will be

graded.

5.2. Simulator Hints

Be careful how you handle offsetField for lw, sw, and beq. Remember that it's

a 2's complement 16-bit number, so you need to convert a negative offsetField

to a negative 32-bit integer on the Sun workstations (by sign extending it).

To do this, use the following function.

int

convertNum(int num)

{

/\* convert a 16-bit number into a 32-bit Sun integer \*/

if (num & (1<<15) ) {

num -= (1<<16);

}

return(num);

}

An example run of the simulator (not for the specified task of multiplication)

is included at the end of this posting.

6. Assembly-Language Multiplication (20%)

The third part of this assignment is to write an assembly-language program to

multiply two numbers. Input the numbers by reading memory locations called

"mcand" and "mplier". The result should be stored in register 1 when the

program halts. You may assume that the two input numbers are at most 15 bits

and are positive; this ensures that the (positive) result fits in an LC3101

word. See the algorithm on page 252 of the textbook for how to multiply.

Remember that shifting left by one bit is the same as adding the number to

itself. Given the LC3101 instruction set, it's easiest to modify the

algorithm so that you avoid the right shift. Submit a version of the program

that computes (32766 \* 10383).

Your multiplication program must be reasonably efficient--it must be at most

50 lines long and execute at most 1000 instructions for any valid numbers (this

is several times longer and slower than the solution). To achieve this, you

must use a loop and shift algorithm to perform the multiplication; algorithms

such as successive addition (e.g. multiplying 5 \* 6 by adding 5 six times)

will take too long.

7. Grading and Formatting

We will grade primarily on functionality, including error handling, correctly

assembling and simulating all instructions, input and output format, method of

executing your program, correctly multiplying, and comprehensiveness of the

test suites.

The best way to debug your program is to generate your own test cases, figure

out the correct answers, and compare your program's output to the correct

answers. This is also one of the best ways to learn the concepts in the

project.

The student suite of test cases for the assembler and simulator parts of this

project will be graded according to how thoroughly they test an LC3101

assembler or simulator. We will judge thoroughness of the test suite by how

well it exposes potentially bugs in an assembler or simulator.

For the assembler test suite, we will use each test case as input to a set

of buggy assemblers. A test case exposes a buggy assembler by causing it

to generate a different answer from a correct assembler. The test suite

is graded based on how many of the buggy assemblers were exposed by at

least one test case. This is known as "mutation testing" in the research

literature on automated testing.

For the simulator test suite, we will correctly assemble each test case,

then use it as input to a set of buggy simulators. A test case exposes a

buggy simulator by causing it to generate a different answer from a

correct simulator. The test suite is graded based on how many of the buggy

simulators were exposed by at least one test case.

8. Turning in the Project

Submit you files through blackboard.

Each part should be archived in a .tar or .zip file to help with grading.

Here are the files you should submit for each project part:

1) assembler (part 1a)

a. C/C++ program for your assembler

b. suite of test cases (each test case is an assembly-language program

in a separate file)

2) simulator (part 1s)

a. C/C++ program for your simulator

b. suite of test cases (each test case is an assembly-language program

in a separate file)

3) multiplication (part 1m)

a. assembly program for multiplication

Your assembler and simulator must each be in a single C or C++ file. We will compile

your program on linprog using "gcc program.c -lm" (or g++), so your program

should not require additional compiler flags or libraries.

The official time of submission for your project will be the time the last file

is sent. If you send in anything after the due date, your project will be

considered late (and will use up your late days or will receive a zero).

9. Code Fragment for Assembler

The focus of this class is machine organization, not C programming skills. To

"build" your computer, however, you will be doing a lot of C programming. To

help you, here is a fragment of the C program for the assembler. This shows

how to specify command-line arguments to the program (via argc and argv), how

to parse the assembly-language file, etc.. This fragment is provided strictly

to help you, though it may take a bit for you to understand and use the file.

You may also choose to not use this fragment.

/\* Assembler code fragment for LC3101 \*/

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#define MAXLINELENGTH 1000

int readAndParse(FILE \*, char \*, char \*, char \*, char \*, char \*);

int isNumber(char \*);

int

main(int argc, char \*argv[])

{

char \*inFileString, \*outFileString;

FILE \*inFilePtr, \*outFilePtr;

char label[MAXLINELENGTH], opcode[MAXLINELENGTH], arg0[MAXLINELENGTH],

arg1[MAXLINELENGTH], arg2[MAXLINELENGTH];

if (argc != 3) {

printf("error: usage: %s <assembly-code-file> <machine-code-file>\n",

argv[0]);

exit(1);

}

inFileString = argv[1];

outFileString = argv[2];

inFilePtr = fopen(inFileString, "r");

if (inFilePtr == NULL) {

printf("error in opening %s\n", inFileString);

exit(1);

}

outFilePtr = fopen(outFileString, "w");

if (outFilePtr == NULL) {

printf("error in opening %s\n", outFileString);

exit(1);

}

/\* here is an example for how to use readAndParse to read a line from

inFilePtr \*/

if (! readAndParse(inFilePtr, label, opcode, arg0, arg1, arg2) ) {

/\* reached end of file \*/

}

/\* this is how to rewind the file ptr so that you start reading from the

beginning of the file \*/

rewind(inFilePtr);

/\* after doing a readAndParse, you may want to do the following to test the

opcode \*/

if (!strcmp(opcode, "add")) {

/\* do whatever you need to do for opcode "add" \*/

}

return(0);

}

/\*

\* Read and parse a line of the assembly-language file. Fields are returned

\* in label, opcode, arg0, arg1, arg2 (these strings must have memory already

\* allocated to them).

\*

\* Return values:

\* 0 if reached end of file

\* 1 if all went well

\*

\* exit(1) if line is too long.

\*/

int

readAndParse(FILE \*inFilePtr, char \*label, char \*opcode, char \*arg0,

char \*arg1, char \*arg2)

{

char line[MAXLINELENGTH];

char \*ptr = line;

/\* delete prior values \*/

label[0] = opcode[0] = arg0[0] = arg1[0] = arg2[0] = '\0';

/\* read the line from the assembly-language file \*/

if (fgets(line, MAXLINELENGTH, inFilePtr) == NULL) {

/\* reached end of file \*/

return(0);

}

/\* check for line too long (by looking for a \n) \*/

if (strchr(line, '\n') == NULL) {

/\* line too long \*/

printf("error: line too long\n");

exit(1);

}

/\* is there a label? \*/

ptr = line;

if (sscanf(ptr, "%[^\t\n ]", label)) {

/\* successfully read label; advance pointer over the label \*/

ptr += strlen(label);

}

/\*

\* Parse the rest of the line. Would be nice to have real regular

\* expressions, but scanf will suffice.

\*/

sscanf(ptr, "%\*[\t\n ]%[^\t\n ]%\*[\t\n ]%[^\t\n ]%\*[\t\n ]%[^\t\n ]%\*[\t\n ]%[^\t\n ]",

opcode, arg0, arg1, arg2);

return(1);

}

int

isNumber(char \*string)

{

/\* return 1 if string is a number \*/

int i;

return( (sscanf(string, "%d", &i)) == 1);

}

10. Code Fragment for Simulator

Here is some C code that may help you write the simulator. Again, you should

take this merely as a hint. You may have to re-code this to make it do exactly

what you want, but this should help you get started. Remember not to

change stateStruct or printState.

/\* instruction-level simulator for LC3101 \*/

#include <stdio.h>

#include <string.h>

#define NUMMEMORY 65536 /\* maximum number of words in memory \*/

#define NUMREGS 8 /\* number of machine registers \*/

#define MAXLINELENGTH 1000

typedef struct stateStruct {

int pc;

int mem[NUMMEMORY];

int reg[NUMREGS];

int numMemory;

} stateType;

void printState(stateType \*);

int

main(int argc, char \*argv[])

{

char line[MAXLINELENGTH];

stateType state;

FILE \*filePtr;

if (argc != 2) {

printf("error: usage: %s <machine-code file>\n", argv[0]);

exit(1);

}

filePtr = fopen(argv[1], "r");

if (filePtr == NULL) {

printf("error: can't open file %s", argv[1]);

perror("fopen");

exit(1);

}

/\* read in the entire machine-code file into memory \*/

for (state.numMemory = 0; fgets(line, MAXLINELENGTH, filePtr) != NULL;

state.numMemory++) {

if (sscanf(line, "%d", state.mem+state.numMemory) != 1) {

printf("error in reading address %d\n", state.numMemory);

exit(1);

}

printf("memory[%d]=%d\n", state.numMemory, state.mem[state.numMemory]);

}

return(0);

}

void

printState(stateType \*statePtr)

{

int i;

printf("\n@@@\nstate:\n");

printf("\tpc %d\n", statePtr->pc);

printf("\tmemory:\n");

for (i=0; i<statePtr->numMemory; i++) {

printf("\t\tmem[ %d ] %d\n", i, statePtr->mem[i]);

}

printf("\tregisters:\n");

for (i=0; i<NUMREGS; i++) {

printf("\t\treg[ %d ] %d\n", i, statePtr->reg[i]);

}

printf("end state\n");

}

11. Programming Tips

Here are a few programming tips for writing C/C++ programs to manipulate bits:

1) To indicate a hexadecimal constant in, precede the number by 0x. For

example, 27 decimal is 0x1b in hexadecimal.

2) The value of the expression (a >> b) is the number "a" shifted right by "b"

bits. Neither a nor b are changed. E.g. (25 >> 2) is 6. Note that 25 is 11001 in

binary, and 6 is 110 in binary.

3) The value of the expression (a << b) is the number "a" shifted left by "b"

bits. Neither a nor b are changed. E.g. (25 << 2) is 100. Note that 25 is 11001

in binary, and 100 is 1100100 in binary.

4) To find the value of the expression (a & b), perform a logical AND on each

bit of a and b (i.e. bit 31 of a ANDED with bit 31 of b, bit 30 of a ANDED with

bit 30 of b, etc.). E.g. (25 & 11) is 9, since:

11001 (binary)

& 01011 (binary)

---------------------

= 01001 (binary), which is 9 decimal.

5) To find the value of the expression (a | b), perform a logical OR on each bit

of a and b (i.e. bit 31 of a ORED with bit 31 of b, bit 30 of a ORED with bit 30

of b, etc.). E.g. (25 | 11) is 27, since:

11001 (binary)

& 01011 (binary)

---------------------

= 11011 (binary), which is 27 decimal.

6) ~a is the bit-wise complement of a (a is not changed).

Use these operations to create and manipulate machine-code. E.g. to look at bit

3 of the variable a, you might do: (a>>3) & 0x1. To look at bits (bits 15-12) of

a 16-bit word, you could do: (a>>12) & 0xF. To put a 6 into bits 5-3 and a 3

into bits 2-1, you could do: (6<<3) | (3<<1). If you're not sure what an

operation is doing, print some intermediate results to help you debug.

-------------------------------------------------------------------------------

12. Example Run of Simulator

memory[0]=8454151

memory[1]=9043971

memory[2]=655361

memory[3]=16842754

memory[4]=16842749

memory[5]=29360128

memory[6]=25165824

memory[7]=5

memory[8]=-1

memory[9]=2

@@@

state:

pc 0

memory:

mem[ 0 ] 8454151

mem[ 1 ] 9043971

mem[ 2 ] 655361

mem[ 3 ] 16842754

mem[ 4 ] 16842749

mem[ 5 ] 29360128

mem[ 6 ] 25165824

mem[ 7 ] 5

mem[ 8 ] -1

mem[ 9 ] 2

registers:

reg[ 0 ] 0

reg[ 1 ] 0

reg[ 2 ] 0

reg[ 3 ] 0

reg[ 4 ] 0

reg[ 5 ] 0

reg[ 6 ] 0

reg[ 7 ] 0

end state

@@@

state:

pc 1

memory:

mem[ 0 ] 8454151

mem[ 1 ] 9043971

mem[ 2 ] 655361

mem[ 3 ] 16842754

mem[ 4 ] 16842749

mem[ 5 ] 29360128

mem[ 6 ] 25165824

mem[ 7 ] 5

mem[ 8 ] -1

mem[ 9 ] 2

registers:

reg[ 0 ] 0

reg[ 1 ] 5

reg[ 2 ] 0

reg[ 3 ] 0

reg[ 4 ] 0

reg[ 5 ] 0

reg[ 6 ] 0

reg[ 7 ] 0

end state

@@@

state:

pc 2

memory:

mem[ 0 ] 8454151

mem[ 1 ] 9043971

mem[ 2 ] 655361

mem[ 3 ] 16842754

mem[ 4 ] 16842749

mem[ 5 ] 29360128

mem[ 6 ] 25165824

mem[ 7 ] 5

mem[ 8 ] -1

mem[ 9 ] 2

registers:

reg[ 0 ] 0

reg[ 1 ] 5

reg[ 2 ] -1

reg[ 3 ] 0

reg[ 4 ] 0

reg[ 5 ] 0

reg[ 6 ] 0

reg[ 7 ] 0

end state

@@@

state:

pc 3

memory:

mem[ 0 ] 8454151

mem[ 1 ] 9043971

mem[ 2 ] 655361

mem[ 3 ] 16842754

mem[ 4 ] 16842749

mem[ 5 ] 29360128

mem[ 6 ] 25165824

mem[ 7 ] 5

mem[ 8 ] -1

mem[ 9 ] 2

registers:

reg[ 0 ] 0

reg[ 1 ] 4

reg[ 2 ] -1

reg[ 3 ] 0

reg[ 4 ] 0

reg[ 5 ] 0

reg[ 6 ] 0

reg[ 7 ] 0

end state

@@@

state:

pc 4

memory:

mem[ 0 ] 8454151

mem[ 1 ] 9043971

mem[ 2 ] 655361

mem[ 3 ] 16842754

mem[ 4 ] 16842749

mem[ 5 ] 29360128

mem[ 6 ] 25165824

mem[ 7 ] 5

mem[ 8 ] -1

mem[ 9 ] 2

registers:

reg[ 0 ] 0

reg[ 1 ] 4

reg[ 2 ] -1

reg[ 3 ] 0

reg[ 4 ] 0

reg[ 5 ] 0

reg[ 6 ] 0

reg[ 7 ] 0

end state

@@@

state:

pc 2

memory:

mem[ 0 ] 8454151

mem[ 1 ] 9043971

mem[ 2 ] 655361

mem[ 3 ] 16842754

mem[ 4 ] 16842749

mem[ 5 ] 29360128

mem[ 6 ] 25165824

mem[ 7 ] 5

mem[ 8 ] -1

mem[ 9 ] 2

registers:

reg[ 0 ] 0

reg[ 1 ] 4

reg[ 2 ] -1

reg[ 3 ] 0

reg[ 4 ] 0

reg[ 5 ] 0

reg[ 6 ] 0

reg[ 7 ] 0

end state

@@@

state:

pc 3

memory:

mem[ 0 ] 8454151

mem[ 1 ] 9043971

mem[ 2 ] 655361

mem[ 3 ] 16842754

mem[ 4 ] 16842749

mem[ 5 ] 29360128

mem[ 6 ] 25165824

mem[ 7 ] 5

mem[ 8 ] -1

mem[ 9 ] 2

registers:

reg[ 0 ] 0

reg[ 1 ] 3

reg[ 2 ] -1

reg[ 3 ] 0

reg[ 4 ] 0

reg[ 5 ] 0

reg[ 6 ] 0

reg[ 7 ] 0

end state

@@@

state:

pc 4

memory:

mem[ 0 ] 8454151

mem[ 1 ] 9043971

mem[ 2 ] 655361

mem[ 3 ] 16842754

mem[ 4 ] 16842749

mem[ 5 ] 29360128

mem[ 6 ] 25165824

mem[ 7 ] 5

mem[ 8 ] -1

mem[ 9 ] 2

registers:

reg[ 0 ] 0

reg[ 1 ] 3

reg[ 2 ] -1

reg[ 3 ] 0

reg[ 4 ] 0

reg[ 5 ] 0

reg[ 6 ] 0

reg[ 7 ] 0

end state

@@@

state:

pc 2

memory:

mem[ 0 ] 8454151

mem[ 1 ] 9043971

mem[ 2 ] 655361

mem[ 3 ] 16842754

mem[ 4 ] 16842749

mem[ 5 ] 29360128

mem[ 6 ] 25165824

mem[ 7 ] 5

mem[ 8 ] -1

mem[ 9 ] 2

registers:

reg[ 0 ] 0

reg[ 1 ] 3

reg[ 2 ] -1

reg[ 3 ] 0

reg[ 4 ] 0

reg[ 5 ] 0

reg[ 6 ] 0

reg[ 7 ] 0

end state

@@@

state:

pc 3

memory:

mem[ 0 ] 8454151

mem[ 1 ] 9043971

mem[ 2 ] 655361

mem[ 3 ] 16842754

mem[ 4 ] 16842749

mem[ 5 ] 29360128

mem[ 6 ] 25165824

mem[ 7 ] 5

mem[ 8 ] -1

mem[ 9 ] 2

registers:

reg[ 0 ] 0

reg[ 1 ] 2

reg[ 2 ] -1

reg[ 3 ] 0

reg[ 4 ] 0

reg[ 5 ] 0

reg[ 6 ] 0

reg[ 7 ] 0

end state

@@@

state:

pc 4

memory:

mem[ 0 ] 8454151

mem[ 1 ] 9043971

mem[ 2 ] 655361

mem[ 3 ] 16842754

mem[ 4 ] 16842749

mem[ 5 ] 29360128

mem[ 6 ] 25165824

mem[ 7 ] 5

mem[ 8 ] -1

mem[ 9 ] 2

registers:

reg[ 0 ] 0

reg[ 1 ] 2

reg[ 2 ] -1

reg[ 3 ] 0

reg[ 4 ] 0

reg[ 5 ] 0

reg[ 6 ] 0

reg[ 7 ] 0

end state

@@@

state:

pc 2

memory:

mem[ 0 ] 8454151

mem[ 1 ] 9043971

mem[ 2 ] 655361

mem[ 3 ] 16842754

mem[ 4 ] 16842749

mem[ 5 ] 29360128

mem[ 6 ] 25165824

mem[ 7 ] 5

mem[ 8 ] -1

mem[ 9 ] 2

registers:

reg[ 0 ] 0

reg[ 1 ] 2

reg[ 2 ] -1

reg[ 3 ] 0

reg[ 4 ] 0

reg[ 5 ] 0

reg[ 6 ] 0

reg[ 7 ] 0

end state

@@@

state:

pc 3

memory:

mem[ 0 ] 8454151

mem[ 1 ] 9043971

mem[ 2 ] 655361

mem[ 3 ] 16842754

mem[ 4 ] 16842749

mem[ 5 ] 29360128

mem[ 6 ] 25165824

mem[ 7 ] 5

mem[ 8 ] -1

mem[ 9 ] 2

registers:

reg[ 0 ] 0

reg[ 1 ] 1

reg[ 2 ] -1

reg[ 3 ] 0

reg[ 4 ] 0

reg[ 5 ] 0

reg[ 6 ] 0

reg[ 7 ] 0

end state

@@@

state:

pc 4

memory:

mem[ 0 ] 8454151

mem[ 1 ] 9043971

mem[ 2 ] 655361

mem[ 3 ] 16842754

mem[ 4 ] 16842749

mem[ 5 ] 29360128

mem[ 6 ] 25165824

mem[ 7 ] 5

mem[ 8 ] -1

mem[ 9 ] 2

registers:

reg[ 0 ] 0

reg[ 1 ] 1

reg[ 2 ] -1

reg[ 3 ] 0

reg[ 4 ] 0

reg[ 5 ] 0

reg[ 6 ] 0

reg[ 7 ] 0

end state

@@@

state:

pc 2

memory:

mem[ 0 ] 8454151

mem[ 1 ] 9043971

mem[ 2 ] 655361

mem[ 3 ] 16842754

mem[ 4 ] 16842749

mem[ 5 ] 29360128

mem[ 6 ] 25165824

mem[ 7 ] 5

mem[ 8 ] -1

mem[ 9 ] 2

registers:

reg[ 0 ] 0

reg[ 1 ] 1

reg[ 2 ] -1

reg[ 3 ] 0

reg[ 4 ] 0

reg[ 5 ] 0

reg[ 6 ] 0

reg[ 7 ] 0

end state

@@@

state:

pc 3

memory:

mem[ 0 ] 8454151

mem[ 1 ] 9043971

mem[ 2 ] 655361

mem[ 3 ] 16842754

mem[ 4 ] 16842749

mem[ 5 ] 29360128

mem[ 6 ] 25165824

mem[ 7 ] 5

mem[ 8 ] -1

mem[ 9 ] 2

registers:

reg[ 0 ] 0

reg[ 1 ] 0

reg[ 2 ] -1

reg[ 3 ] 0

reg[ 4 ] 0

reg[ 5 ] 0

reg[ 6 ] 0

reg[ 7 ] 0

end state

@@@

state:

pc 6

memory:

mem[ 0 ] 8454151

mem[ 1 ] 9043971

mem[ 2 ] 655361

mem[ 3 ] 16842754

mem[ 4 ] 16842749

mem[ 5 ] 29360128

mem[ 6 ] 25165824

mem[ 7 ] 5

mem[ 8 ] -1

mem[ 9 ] 2

registers:

reg[ 0 ] 0

reg[ 1 ] 0

reg[ 2 ] -1

reg[ 3 ] 0

reg[ 4 ] 0

reg[ 5 ] 0

reg[ 6 ] 0

reg[ 7 ] 0

end state

machine halted

total of 17 instructions executed

final state of machine:

@@@

state:

pc 7

memory:

mem[ 0 ] 8454151

mem[ 1 ] 9043971

mem[ 2 ] 655361

mem[ 3 ] 16842754

mem[ 4 ] 16842749

mem[ 5 ] 29360128

mem[ 6 ] 25165824

mem[ 7 ] 5

mem[ 8 ] -1

mem[ 9 ] 2

registers:

reg[ 0 ] 0

reg[ 1 ] 0

reg[ 2 ] -1

reg[ 3 ] 0

reg[ 4 ] 0

reg[ 5 ] 0

reg[ 6 ] 0

reg[ 7 ] 0

end state