Let’s Solve a Problem

Let’s see how we can translate
◦ a task in human terms
◦ into a program using LC-3 instructions.

Starting with the task, we shall
◦ identify information that we need to store,
◦ assign registers for stored values
◦ draw a flow chart for the code
   (roughly at the level of instructions), and
◦ write instructions in human-readable form.
   (The actual program is also available to you.)

Overview of Our Task

Let the user type a number
◦ from 0 to 32767
◦ using the keyboard, and
◦ pressing <Enter> when done.

Read in the number, convert it to
2’s complement, and store it in memory.

Give errors if
1. a non-digit is pressed, or
2. overflow occurs (> 32767).

Programs are Finite State Machines

Our first question:

What do we need to store?
In other words, what information do we need
to have handy in order to solve the problem?

Does this question remind you of FSMs?
◦ Keys are inputs (including <Enter>),
◦ error messages are outputs, and
◦ number typed is eventually read out.
◦ Our program is like an FSM!
What Information Do We Need to Store?

But back to the question:

What do we need to store?

In other words, what information do we need to have handy in order to solve the problem?

1. the key pressed (one at a time)
2. the current value of the user’s number (from previous keystrokes)

Let’s Also Store the Value xFFD0

It’s also convenient to store xFFD0. Any idea why?

xFFD0 = –x0030 = ‘0’ (the 0 digit in ASCII)

So?

When we want to convert a digit typed in ASCII into a 2’s complement value,
• we need to subtract x0030,
• but we can only subtract x0010 with a single ADD instruction.

Assign a Register for Each Value that We Store

Finally, we need a temporary value for computations.
Let’s assign registers.
When we use GETC (TRAP x20)
• to read a character,
• the keystroke comes back in R0, so
• use R0 for the key pressed.
R1 can be the current value of the number.
R2 can hold xFFD0.
And R3 can be our temporary.

How Do We Update the Current Value?

When the user presses a key, how do we update the “current value?”

For example, suppose that
• the user has typed 3, 2, 7, and 6, (in that order),
• so the “current value” is 3276.
If the user presses ‘7,’ we should
• use 3276 and 7
• to calculate 32767.

How?
A Formula for Updating the Current Value

new value = 10 × current value + new digit

Like that?
Good, we can use the LC-3 MULTIPLY instruction.
Oh.
Well, we’ll figure it out.
Let’s draw a flow chart.

We’re Ready to Write Instructions!

START
initialize variables
read a character
is a digit?
TRUE
multiply by 10
is <Enter>?
FALSE
add new digit
store the number

DONE
print error
overflow?
TRUE
FALSE

Which Registers Need to Be Initialized?

Here’s our table of registers.
R0 will be filled in by GETC.
R3 is just a temporary.
We need to set R2 to xFFD0.

What about R1?
(What value should it have before the user presses a key?)
Derive the Initial Value from the Update Formula

Here is our formula for updating:
\[ \text{new value} = 10 \times \text{current value} + \text{new digit} \]

If the user
- first presses 5,
- we want new value to be 5, so

\[ 5 = 10 \times \text{current value} + 5 \]

What should “current value” be? 0
Ok, so we have to initialize R1 to 0.

Code to Initialize Variables

\[ \text{x3000 LD R2, } \_\_\_\_\_\_ \]

For R2, we can load the desired value from memory.

Where in memory?
After the program.
Where’s that?
We don’t know yet.

For R1, we can use an AND instruction to set it to 0.

What’s Next? Reading a Character from the Keyboard

START
read a character
is a digit?
TRUE
multiply by 10
FALSE
add new digit
is <Enter>?
TRUE
print error
FALSE
store the number

DONE
We need to use a TRAP to read a character from the keyboard.

TRAP x20 (GETC) reads a character into R0.

Why TRAP x21, too?

We want the character typed to be echoed to the display (with an OUT trap).

TRAP x21

ADD R3, R0, #10

If the result is 0, the key pressed (R0) was <Enter>.

The <Enter> key produces ASCII character #10.

Subtract #10 to make a comparison.

Where does the result go? Discard it (into R3).

R0 is the key pressed.

The <Enter> key produces ASCII character #10.

The <Enter> key produces ASCII character #10.

If the result is 0, the key pressed (R0) was <Enter>.

Let's branch on zero to the code that stores the number, which is ... somewhere. We'll leave the offset blank.

The <Enter> key produces ASCII character #10.

BRz ______
What’s Next? Checking for a Digit

START
read a character
initialize variables
TRUE
is a digit?
FALSE
<Enter>?
TRUE
print error
FALSE

is a digit?
FALSE
add new digit
TRUE
overflow?
FALSE
TRUE

DONE
store the number

Code to Check for a Digit

<table>
<thead>
<tr>
<th>x3000 LD R2, ___</th>
<th>Remember that R2 has negative ASCII digit 0 (xFFD0).</th>
</tr>
</thead>
<tbody>
<tr>
<td>x3001 AND R1,R1,#0</td>
<td>Let’s first convert our key to binary, assuming it’s a digit.</td>
</tr>
<tr>
<td>x3002 TRAP x20</td>
<td></td>
</tr>
<tr>
<td>x3003 TRAP x21</td>
<td></td>
</tr>
<tr>
<td>x3004 ADD R3,R0,#-10</td>
<td></td>
</tr>
<tr>
<td>x3005 BRz __________</td>
<td></td>
</tr>
<tr>
<td>x3006 ADD R0,R0,R2</td>
<td></td>
</tr>
<tr>
<td>x3007 BRn _________</td>
<td></td>
</tr>
</tbody>
</table>

If the result is below 0 (negative), the original character was less than x30, and thus not a digit.

Where to go? Later.
**Code to Check for a Digit**

```
x3000 LD R2, ______
x3001 AND R1,R1,#0
x3002 TRAP x20
x3003 TRAP x21
x3004 ADD R3,R0,#-10
x3005 BRz ______
x3006 ADD R0,R0,R2
x3007 BRn ______
x3008 ADD R3,R0,#-10
x3009 BRzp ______
```

Now R0 holds the key minus x30. If a digit, 0 to 9.

If the result was equal or greater to 0, the key was greater than '9.'

Where to go? Later.

**What's Next? Multiplying by 10**

**Code to Multiply R1 by 10**

```
x3000 LD R2, ______
x3001 AND R1,R1,#0
x3002 TRAP x20
x3003 TRAP x21
x3004 ADD R3,R0,#-10
x3005 BRz ______
x3006 ADD R0,R0,R2
x3007 BRn ______
x3008 ADD R3,R0,#-10
x3009 BRzp ______
```

Multiply the current value (R1) by 10.

Anyone want to learn ARM now? Or maybe x86? It has "MULT!"

Let me just write some code...

**Code to Multiply R1 by 10**

```
x300A ADD R3,R1,R1
x300B ADD R3,R3,R3
x300C ADD R1,R1,R1
x300D ADD R1,R1,R1
```

Multiply the current value (R1) by 10.

Look good?

Great.

Let's move on?
Code to Multiply R1 by 10

; This is a comment.
x300A ADD R3,R1,R1
; Now R3 has 2V.
x300B ADD R3,R3,R3
; Now R3 has 4V.
x300C ADD R1,R1,R3
; Now R1 has 5V.
x300D ADD R1,R1,R1
; Now R1 has 10V.

Multiply the current value (R1) by 10.

Let's use V to denote the original value of R1.

Code to Add the New Digit

x300A ADD R3,R1,R1
x300B ADD R3,R3,R3
x300C ADD R1,R1,R3
x300D ADD R1,R1,R1
x300E ADD R1,R1,R0

Add the new digit into R1.

Where is the new digit? We already computed it and stored it in R0.

What’s Next? Adding the New Digit

START
initialize variables
read a character
is a digit?
TRUE multiply by 10
FALSE

TRUE
FALSE
is <Enter>?
add new digit

store the number
print error

DONE

What’s Next? Checking for Overflow

START
initialize variables
read a character
is a digit?
TRUE multiply by 10
FALSE

is <Enter>?
add new digit

store the number
print error

DONE

overflow?
TRUE
FALSE
over-
flow?
How do we check for overflow?
Unfortunately, it’s not so easy.
Checking for overflow requires checking all of the ADD instructions.
We won’t do that here.
To see the overflow checks, look at the full version provided to you.

What’s Next? Storing the Number

START
initialize variables
TRUE
read a character
is a digit?
FALSE
TRUE
MULTIPLY BY 10
FALSE
ADDITION OF THE NEW DIGIT
FALSE
OVERFLOW?
TRUE
PRINT ERROR
FALSE
STORE THE NUMBER
DONE

Code to Add the New Digit

x300A ADD R3,R1,R1
x300B ADD R3,R3,R3
x300C ADD R1,R1,R3
x300D ADD R1,R1,R1
x300E ADD R1,R1,R0
x300F BRnzp ______

Go get another digit!

How? Use an unconditional branch.

Let’s figure out the offset later.

Code to Store the Number

x300A ADD R3,R1,R1
x300B ADD R3,R3,R3
x300C ADD R1,R1,R3
x300D ADD R1,R1,R1
x300E ADD R1,R1,R0
x300F BRnzp ______

Store the number to memory.

Let’s use an ST to store it nearby.

The number is in R1.

Where? Well… later.
Code to End the Program

- ADD R3, R1, R1
- ADD R3, R3, R3
- ADD R1, R1, R0
- BRnzp ______
- ST R1, ______
- TRAP x25

And then we're done!

Use a HALT trap (number x25).

Code for Data

- ADD R3, R1, R1
- ADD R3, R3, R3
- ADD R1, R1, R0
- BRnzp ______
- ST R1, ______
- TRAP x25
- xFFD0
- place for number

But we still need a couple more things.

First, we need the value xFFD0.

Second, we need a place to store the number.

Here's the Whole Program

- LD R2, ______
- AND R1, R1, #0
- TRAP x20
- TRAP x21
- ADD R3, R0, #10
- BRz ______
- ADD R0, R0, R2
- BRn ______
- ADD R3, R0, #10
- BRzp ______
- ADD R3, R1, R1
- ADD R3, R3, R3
- ADD R1, R1, R3
- ADD R1, R1, R1
- ADD R1, R1, R0
- BRnzp ______
- ST R1, ______
- TRAP x25
- xFFD0
- x3013 place for number

Now for Some Real Fun!

It's time for...

Well, yes, we'll turn them into bits.

But I meant counting!

Almost as exciting as bits.
Help Me Count (Please!)

x3000 LD R2, x11
x3001 AND R1,R1,#0
x3002 TRAP x20
x3003 TRAP x21
x3004 ADD R3,R0,#-10
x3005 BRz
x3006 ADD R0,R0,R2
x3007 BRn
x3008 ADD R3,R0,#-10
x3009 BRzp

PC will point here

x300A ADD R3,R1,R1
x300B ADD R3,R3,R3
x300C ADD R1,R1,R3
x300D ADD R1,R1,R1
x300E ADD R1,R1,R0
x300F BRnzp
x3010 ST R1,

PC will point here

x3011 TRAP x25
x3012 xFFD0
x3013 place for number

Help Me Count (Please!)

x3000 LD R2, x11
x3001 AND R1,R1,#0
x3002 TRAP x20
x3003 TRAP x21
x3004 ADD R3,R0,#-10
x3005 BRz
x3006 ADD R0,R0,R2
x3007 BRn
x3008 ADD R3,R0,#-10
x3009 BRzp

PC will point here

x300A ADD R3,R1,R1
x300B ADD R3,R3,R3
x300C ADD R1,R1,R3
x300D ADD R1,R1,R1
x300E ADD R1,R1,R0
x300F BRnzp
x3010 ST R1,

PC will point here

x3011 TRAP x25
x3012 xFFD0
x3013 place for number

Help Me Count (Please!)

x3000 LD R2, x11
x3001 AND R1,R1,#0
x3002 TRAP x20
x3003 TRAP x21
x3004 ADD R3,R0,#-10
x3005 BRz
x3006 ADD R0,R0,R2
x3007 BRn
x3008 ADD R3,R0,#-10
x3009 BRzp

PC will point here

x300A ADD R3,R1,R1
x300B ADD R3,R3,R3
x300C ADD R1,R1,R3
x300D ADD R1,R1,R1
x300E ADD R1,R1,R0
x300F BRnzp
x3010 ST R1,

PC will point here

x3011 TRAP x25
x3012 xFFD0
x3013 place for number
Help Me Count (Please!)

- x3000 LD R2, x11
- x3001 AND R1,R1,#0
- x3002 TRAP x20
- x3003 TRAP x21
- x3004 ADD R3,R0,#-10
- x3005 BRz xA
- x3006 ADD R0,R0,R2
- x3007 BRn xC
- x3008 ADD R3,R0,#-10
- x3009 BRzp xA

PC will point here

ADD R3,R1,R1
ADD R3,R3,R3
ADD R1,R1,R3
ADD R1,R1,R1
ADD R1,R1,R0
BRnzp x-Ex
ST R1, x2

Help Me Count (Please!)

- x3000 LD R2, x11
- x3001 AND R1,R1,#0
- x3002 TRAP x20
- x3003 TRAP x21
- x3004 ADD R3,R0,#-10
- x3005 BRz xA
- x3006 ADD R0,R0,R2
- x3007 BRn xC
- x3008 ADD R3,R0,#-10
- x3009 BRzp xA

PC will point here

ADD R3,R1,R1
ADD R3,R3,R3
ADD R1,R1,R3
ADD R1,R1,R1
ADD R1,R1,R0
BRnzp x-
ST R1, x2

Now We Can Write Bits!

- x3000 LD R2, x11
- x3001 AND R1,R1,#0
- x3002 TRAP x20
- x3003 TRAP x21
- x3004 ADD R3,R0,#-10
- x3005 BRz xA
- x3006 ADD R0,R0,R2
- x3007 BRn xC
- x3008 ADD R3,R0,#-10
- x3009 BRzp xA

PC will point here

ADD R3,R1,R1
ADD R3,R3,R3
ADD R1,R1,R3
ADD R1,R1,R1
ADD R1,R1,R0
BRnzp x-
ST R1, x2

Encode the Instruction at x3000 into Bits

- x3000 LD R2, x11

LD R2, 010 0100 1 0000

In a program, we include the spaces between bits (more readable for humans) and add a comment, either “LD R2, x11” or “R2 ← M[PC + x0011].”
A Binary File Starts with the Starting Address

Also, the starting address, \texttt{x3000}, goes first.

For example...

\begin{verbatim}
0011 0000 0000 0000 ; start at x3000
0010 0100 000010001 ; LD R2, x11
; and so forth...
\end{verbatim}

The Rest is Left to You

I’ll leave the rest for you.
I think you can manage it.
Look at the LC-3 encoding table,
and write the bits.
Compare your answers with the
code provided on the web page.

Encode the Instruction at x3001 into Bits

\begin{verbatim}
x3001 AND R1,R1,#0
\end{verbatim}