Digital System Design 2019 Project Part 2 – RPN Calculator

Professor Jonathan H. Manton

Prehistory

RPN calculators were very popular with engineers and scientists, back before everyone had a computer built into their phones. My most recent RPN calculator, the HP-11C, I still use. (Sadly for me, it is featured on a website called Vintage Calculators.)

Figure 1: HP-11C. Image taken from http://vintage-calculator.com/hp-11c

RPN calculators are easier to build (as you will discover) and faster to use.

The Stack

It is assumed you already have some familiarity with RPN calculators, either because you have used them before, or because you have asked Mr Google.

An RPN calculator has a stack. When you use an RPN calculator, it helps to have a mental image of the stack. Here is an example of how to calculate 3 * (25 – 4). Read the table below from left to right. The top row lists the keys that you press while the bottom four rows represent the contents of the stack immediately after you have pressed each key. (Normally empty spots on the stack are assumed to contain zero. However, for this project, we will denote them as being empty.)

Initialisation	**3**	**25**	**4**	**-**	*
			3		
		3	25	3	
	3	25	4	21	63

Call the elements of the stack, stack[0], stack[1], stack[2] and stack[3]. In the above table, stack[0] is on the bottom and stack[3] is on top.

When you “Enter” or “Push” a number onto the stack, the stack moves up and the new number goes into stack[0]. The contents of stack[3] is lost forever.

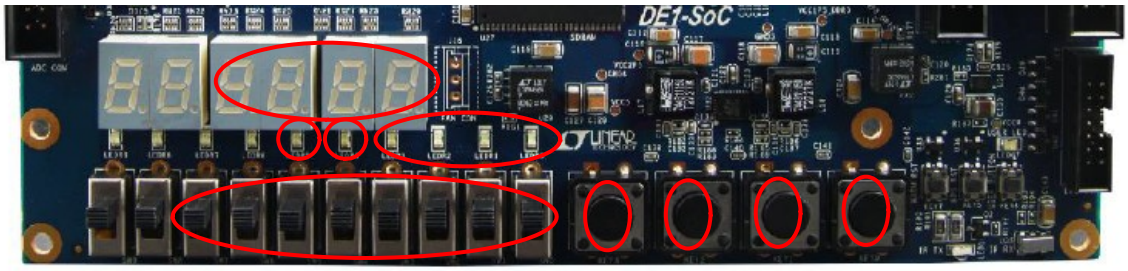
[new number] -> stack[0] -> stack[1] -> stack[2] -> stack[3] -> [lost into the ether]

When you press an arithmetic operation, such as addition, that operation occurs on the first two entries of the stack. (For subtraction and division the order matters.) The other entries of the stack move down one spot.

stack[0] = stack[1] + stack[0]; stack[1] <- stack[2] <- stack[3] <- “empty”

Interface

Parts of the interface are hardwired into the computer that you built in Part 1. The following describes only the parts that can be programmed. Indeed, Part 2 is purely a software project.



Numbered from left to right, top to bottom:

1. Current contents of stack[0] (-128 to 127), or blank if stack is empty.
2. Arithmetic overflow (ON when addition or multiplication produced an overflow)
3. Stack overflow (ON when a Push caused the stack to overflow)
4. Number of values on the stack (LED3 ON means stack is full; LED2 ON means 3 values on the stack; LED1 ON means 2 values on the stack; LED0 ON means 1 value on the stack; all LEDs OFF means the stack is empty)

5. Number entry (binary representation of 8-bit 2's complement number)
6. **Push**. When released, the number represented by the switches will be pushed onto the stack, and hence be displayed on the 7-segment display.
7. **Pop**. When released, stack[0] is discarded and the rest of the stack moves down: stack[0] <- stack[1] <- stack[2] <- stack[3] <- "empty"
8. **Addition**. When released, stack[0] = stack[0] + stack[1]; stack[1] <- stack[2] <- stack[3] <- "empty". The addition is signed addition.
9. **Multiplication**. Same as for Add, but with addition replaced by signed multiplication.

Objective

The functionality of the calculator has been described above. You are to implement the calculator by programming the computer that you built in Part 1 of this project.

In your project report, document the steps you took.

Remark

Everyone should program in machine code at least once in his or her lifetime. It quickly becomes tedious though, which is why assembly language was introduced. And even assembly language becomes tedious, so (eventually) C was introduced. And C becomes tedious so even higher-level languages are being introduced. But in all cases, the end result is machine code.

Some General Suggestions

Perhaps the biggest frustration with machine code is that if you need to add an instruction, you have to renumber all the instructions after it! And worse, you must also check every Jump statement to see whether the Jump Address needs updating too. Therefore, you should plan your program on paper first, and only when you are confident, should you translate it into machine code and type it into ROM.v. (Or you could write your own assembler that will produce the machine code for you – but for a once-off project like this, it might take longer than doing it by hand.)

Similarly, at the planning stage, do not write in machine code, but write using abbreviations. For example, if you want to move the contents of Register 2 to Register 3, instead of writing `MOV `PUR `REG 2 `REG 3, just write something like R2 -> R3. Also, use labels rather than line numbers to denote jump addresses, so that you can insert and delete instructions without having to make changes to jump addresses.

The slow and steady approach would be to build up to the full calculator. Start by just being able to Push a number into stack[0] then have it appear on the display. Then fully implement the stack, including the stack LEDs and stack overflow LED. Then implement Pop. Then addition. And finish with multiplication.

The fast approach is to implement everything at once. This project is sufficiently small that this approach is feasible, but it is only recommended for those already comfortable with programming computers. Debugging in machine code is not an enjoyable experience, so the slow and steady approach may well end up being faster than the fast approach (as the Hare and the Tortoise found out for themselves).

Hints

You should not need these hints.

Everything is a finite state machine. So decide first on what your state is. In other words, what variables are you going to need, and what registers will you store them in?

What is the most convenient representation to use to store the contents of your variables in?

What sequence of instructions is required to carry out a Push?

A Pop?

An Addition?

A Multiplication?

How can you display a number on the 7-segment display?

How can you display a blank on the 7-segment display?

How can you switch the LEDs on and off?

How can you determine if a button has been released?

Draw a flow chart showing how your calculator will function. Do not forget an initialisation step at the start, since after a Reset, the Registers of the CPU are in an unknown state (except for the Flag Register which has been cleared).

Big Hints

You should definitely not need these hints!

We can use the following registers.

Register	Purpose
0	stack[0]
1	stack[1]
2	stack[2]
3	stack[3]
4	Size of stack

It is more convenient to represent the size of the stack, not with the numbers 0 to 4, but in the same way that we must display them to the user.

Value	Meaning
8'b 0	Stack is empty
8'b 1	One element is on the stack
8'b 10	Two elements are on the stack
8'b 100	Three elements are on the stack
8'b 1000	Four elements are on the stack

The `MOV `SHL and `MOV `SHR operations can be used to change the value of Register[4]. Of course, tests will be needed to make sure the stack is not empty or full: the `JMP `EQ command can be used for such tests.

Push and Pop operations can be achieved using a series of `MOV `PUR instructions.

Addition and multiplication can be achieved using the ACC command group.

The following can be a starting point for determining the basic structure of the program. (I might have omitted some small but crucial steps, and the following is not how I implemented the calculator – you are encouraged to write a "nicer" program than the following.)

Label	Instruction
	Initialisation. After a Reset, the CPU Registers will be in an unknown state. Set the stack size to zero, and turn off the LEDs and display.
wait	Jump to "push" if the Push button has been released
	Jump to "pop" if the Pop button has been released
	Jump to "add" if the Addition button has been released
	Jump to "mult" if the Multiplication button has been released
	Jump to "wait"
push	Move stack up
	Move `DINP to stack[0]
	Display stack[0] on 7-segment display
	Jump to "overflow" if Register[4] equals 8

	Turn off both Overflow LEDs
	Shift Register[4] to the left
	Send the contents of Register[4] to the LED pins
	Jump to “wait”
overflow	Turn Stack Overflow LED on; turn Arithmetic Overflow LED off
	Jump to “wait”
pop	Jump to “wait” if Register[4] equals 0
	Turn off both Overflow LEDs
	Shift the stack down
	Shift Register[4] to the right
	Send the contents of Register[4] to the LED pins
	Turn off the 7-segment display
	Jump to “wait” if Register[4] equals zero
	Display stack[0] on the display
	Jump to “wait”
add	Turn off Arithmetic Overflow LED
	Jump to “wait” if Register [4] equals 0 or 1
	Stack[0] = Stack[0] + Stack[1]
	Display Stack[0] on the display
	Turn on Arithmetic Overflow LED if an overflow occurred
	Turn off Stack Overflow LED
	Move rest of stack down
	Shift Register[4] to the right
	Send the contents of Register[4] to the LED pins
	Jump to “wait”
mult	Jump to “wait” if the stack is empty
	Turn off Arithmetic Overflow LED
	Jump to “normal” if the stack has two or more elements
	Set stack[0] to 0
	Display 0 on the 7-segment display
	Jump to “wait”
normal	Stack[0] = Stack[0] * Stack[1]
	Turn on Arithmetic Overflow LED if an overflow occurred
	Turn off Stack Overflow LED
	Display Stack[0] on 7-segment display
	Shift stack down
	Shift Register[4] to the right
	Send the contents of Register[4] to the LED pins
	Jump to “wait”

You should try to see if you can save yourself some time, by identifying common portions of code that could be re-used by using Jump statements. (Without subroutine support, the ability to do this is limited. However, it is still possible to shorten the above code by changing the order in which things are done.)