Previously we saw the JMP and BR instructions. Both are “one-way jumps”: there is no way back to the code from where you jumped, so you just go in one direction.

What if we wanted to come back to where we had been when we branched? That would be pretty easy: we'd just use two jump or branch statements, like this:

```
BRnzp JUMP_HERE
JUMP_BACK some instruction
    . . .
    the rest of the program
    . . .
JUMP_HERE some instruction(s)
BRnzp JUMP_BACK
```

That is, we branch to “JUMP_HERE”, do whatever we need to do, and then branch to “JUMP_BACK”, which is where we started.

One limitation is that these two jumps are inextricably linked: after we do the stuff at “JUMP_HERE”, we necessarily go to “JUMP_BACK”. But what if we wanted to do be able to go-to-and-come-back-from JUMP_HERE from arbitrary points in the program, not just one?

For instance, how could we do this:

```
BRnzp JUMP_HERE
JUMP_BACK_1 some instruction
    . . .
    more stuff
    . . .
BRnzp JUMP_HERE
JUMP_BACK_2 some instruction
    . . .
    more stuff
    . . .
JUMP_HERE some instruction(s)
BRnzp where???
```

That is, there are two places from which we want to jump to “JUMP_HERE”, but how does the last instruction know where to jump back to when that part of the program is done?

Clearly we need some way of knowing where we came from. That is, when we jump to “JUMP_HERE”, we have to store the value of the PC (which would be referring to the next instruction after BRnzp JUMP_HERE), and then restore the PC when we want to go back.

How can our program get the value of the PC? An easy way is to use the LEA instruction. Remember, LEA puts an address into a register, not the data at that address. And that address is calculated relative
Okay, if we can get the value of the PC, what should we do with it? We can put it in a register. Let's use R7 to do that. So we can modify our program as follows (changes are in bold):

```
LEA R7, JUMP_BACK_1
BRnzp JUMP_HERE
JUMP_BACK_1 some instruction
   . . .
   more stuff
   . . .
LEA R7, JUMP_BACK_2
BRnzp JUMP_HERE
JUMP_BACK_2 some instruction
   . . .
   more stuff
   . . .
JUMP_HERE some instruction(s)
JMP R7
```

The designers of the LC-3 ISA realized that this sequence of instructions (LEA R7, ADDR; BRnzp SOMEWHERE) was common enough that it should have its own instruction. In this case, the code starting at “JUMP_HERE” is considered a **subroutine** and the single instruction that will both store the PC in R7 and then jump down there is JSR (for “jump subroutine”, I guess?).

A subroutine is a piece of code that is separate from the rest of the program but can be used by the program by jumping to that part of the code. In other languages it is referred to as a procedure, function, or method. It often is a piece of code that serves a specific purpose and has well-defined (and well-documented) behavior in terms of its inputs ("operands" or "arguments") and outputs.

In the code above, we could jump to the subroutine starting at “JUMP_HERE” by using the instruction “JSR JUMP_HERE” like this:

```
JSR JUMP_HERE
JUMP_BACK_1 some instruction
   . . .
   more stuff
   . . .
JSR JUMP_HERE
JUMP_BACK_2 some instruction
   . . .
   more stuff
   . . .
JUMP_HERE some instruction(s)
JMP R7
```

The JSR instruction works as follows:

- put the current value of the PC into R7 (note that the PC will already be pointing to the next
instruction when JSR is executed)
• calculate the address to which to jump by adding the offset (from the instruction) to the (already
incremented) PC
• set the PC to that address

When the subroutine is finished, it **must** call JMP R7 to go back. In your LC-3 program, you can also use the instruction “RET”, which is just shorthand for “JMP R7”.

Before we proceed, let’s discuss some of the issues with this. You will surely notice that there is an assumption made that R7 will not be modified by the subroutine, otherwise you won’t be able to jump back to where you came from (or, if R7 is modified, its value is first written to memory and then read right before calling RET).

The other issue is that there is an assumption made that the subroutine will not modify any registers that the calling code is using. This is, of course, much harder to enforce: the author of the subroutine may not know which registers are being used when the subroutine is called. When we start studying C, we will look at how this is addressed in “real” programming languages.

The third issue is that the start of the subroutine must be close enough to the address that is calling JSR so that the offset can fit into nine bits (though there are workarounds that you’ve seen before).

**Multiplication example**

Here’s an example of a subroutine to perform multiplication. Since there’s no multiply operator in LC-3, we need to do a sequence of instructions in order to multiply two numbers. If our program does this more than once, it’s probably worth creating a subroutine so that the code is only written once and does not have to be repeated every time we want to multiply.

Before creating this subroutine, we need to consider:
• where will the operands go? that is, the numbers we’re multiplying?
• where will the result go?
Let’s say that our subroutine will expect the operands to be in registers R1 and R2, and that the result will be in R0.

Here’s a possible implementation of this subroutine:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MULTIPY</td>
<td>AND R0, R0, #0; initialize R0 to zero</td>
</tr>
<tr>
<td>LOOP</td>
<td>ADD R0, R0, R1; add R1 to the running sum</td>
</tr>
<tr>
<td></td>
<td>ADD R2, R2, #1; subtract 1 from R2</td>
</tr>
<tr>
<td></td>
<td>BRp LOOP; if positive, keep looping</td>
</tr>
<tr>
<td>RET</td>
<td>; if we get here, we're done</td>
</tr>
</tbody>
</table>

As you can see, this works by adding R1 to itself R2 times, with the result stored in R0.

Note that there are a few assumptions made here, specifically that R1 and R2 contain positive numbers, and that we don't have to worry about overflow. Also, as a side effect, R2 is modified; this is generally bad but okay for now. =)
How would this fit into an entire program? Let's say we have memory locations A, B, and C. We want to first calculate B * C, and then put the value in A. Then we want to multiply that new value of A by 5 and put it in C.

Our program would look like this (line numbers, of course, are not part of the program):

```
.ORIG x3000

;; calculate A = B * C
1 LD R1, B       ; populate the first operand
2 LD R2, C       ; then the second
3 JSR MULTIPLY   ; multiply them
4 ST R0, A       ; write the result to A

;; calculate C = A * 5
5 ADD R1, R0, 0  ; move A from R0 into R1
6 AND R2, R2, #0 ; set R2 to 5
7 ADD R2, R2, #5
8 JSR MULTIPLY   ; multiply again
9 ST R0, C       ; write the result to C

10 HALT

; here is our multiply subroutine
11 MULTIPLY AND R0, R0, #0
12 LOOP ADD R0, R0, R1
13 ADD R2, R2, #-1
14 BRp LOOP
15 RET

.END
```

A few comments:

Note that the registers R1 and R2 were not arbitrarily chosen on lines 1 and 2; our multiply subroutine expects the values to multiply to be in those registers.

Likewise, on line 4, we know that the product is in R0, so that's why we use it to write to A.

On line 5, since we're going to use this new value of A in the second call to the multiply subroutine, we need it in R1 (or R2). We could just load it from memory using LD R1, A, but this is faster.

Lines 6 and 7 put the value 5 into R2, of course.

Before we wrap this up, consider this: we could have written the above program without using subroutines, but it would be a lot longer. Here is that version, with the original line numbers on the left and the inserted code in bold:
The sequence of instructions is exactly the same, of course, but writing code this way is error-prone (e.g. note that the first loop and second loop have to have different labels) and so using subroutines makes a lot of sense here.

Before we move on, let’s talk about the encoding of JSR. There are actually two versions of JSR:

- JSR, which we just discussed and takes an offset as its operand
- JSRR, in which the address of the start of the subroutine is stored in a register

Both instructions have the same opcode (0100). The next bit indicates the mode: 0 means to use an offset; 1 means to use a register. If using an offset, it is represented in the remaining 11 bits. If using a register (for JSRR), the next two bits are zeroes, then three bits for the register that’s holding the address of the start of the subroutine, then six zeroes at the end.

**Traps**

Sometimes a subroutine may be so common that it should be shared by *all* programs on the computer, e.g. reading from the keyboard or writing to the display. After all, what’s the point of having all LC-3 programmers reimplement those few lines over and over?

Such common subroutines can be implemented in the operating system and are known as service routines, or “traps”. These are subroutines that are available to all programs on the computer (assuming that the operating system has been loaded into memory).
Service routines are not reference by their names because of two important reasons. One is that the operating system usually resides in the very low addresses of memory (starting at x0000), so the start of the subroutine may be too far away from the program.

A more important reason is that if you could access a service routine by its name, e.g. “IN”, then the assembler would have to have “IN” in its symbol table in order to calculate the offset for JSR; however, the code that is labeled “IN” is not part of your program (it's part of the operating system), so the assembler wouldn't know about it. This would also prohibit your program from having an instruction labeled “IN”.

So in order to address this issue, LC-3 (and most real microarchitectures) let programmers access service routines by number, instead of by name. For instance, in LC-3 the service routine to read in a single character from the keyboard is trap x23.

Then how does it figure out where to jump to? The starting address of each service routine is stored in a trap vector table that resides in the very low memory addresses. There can be up to 256 service routines, and their starting addresses are stored in memory at addresses x0000 through x00FF (that's 256 in total).

For instance, when your program wants to read in a single character from the keyboard, it can use the instruction “TRAP x23”, which works as follows:

- the current value of the PC is written to R7 (same as JSR)
- the trap number (or trap vector) is sign-extended to 16 bits; in this case it would be x0023
- that address is read (note that it is an address in the trap vector table) to find the starting address of the service routine; in the LC-3 operating system that you can use for your homework, that starting address is x0234
- the PC is set to the starting address of the service routine (in this example, x0234) and the program continues

Of course, when the service routine ends, it must call RET in order to go back.

The opcode for the TRAP instruction is 1111. The next four bits are unused, and the last eight bits represent the trap vector.