The LC-3 Computer, Part 2

Control and TRAP Instructions

CS 350: Computer Organization & Assembler Language Programming

[3/31: Loop N times]

A. Why?

- Control (branch and jump) instructions let us implement decisions and loops.
- Trap instructions let us access operating-system-level routines like I/O.

B. Outcomes

At the end of today, you should know how

- The LC-3 branch and jump instructions work.
- To use the `TRAP` instruction to read/write a character or halt the program.

C. Control Instructions

- Control instructions alter the sequence of instructions being executed and let us go to other instructions.
  - They work by changing the PC during the **EXECUTE INSTRUCTION** phase of instruction cycle.
  - Compare with the PC change done for every instruction during the **FETCH INSTRUCTION** phase of the instruction cycle.
- **Short and Long-distance go-to**
  - On LC-3, the `BR` (branch) instruction specifies target address using PC-offset so you can only do a short-distance jump with one.
  - The `JMP` (jump) instruction specifies target using a base register so, so you can jump anywhere ("long-distance" jump).
- **Conditional and Unconditional go-to**
  - On LC-3, `JMP` is unconditional; `BR` is conditional (though "always" is a possible condition).
- **Jump instruction**: The value in the base register is used as the new PC value; i.e., we go to the address indicated by the base register. Before the copy, the

`PC` points to the next instruction. After we copy the new address to the `PC`, the next fetch instruction will get the instruction at this new address.

- `PC` ← *Base Register*

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JMP	1	1	0	0	0	0	0		Base		0	0	0	0	0	0

D. Branch instruction

- The branch instruction `BR` contains a **mask**, which it combines with the current **condition code** to decide whether or not to do a go to.
- **Condition code**: There is a 3-bit condition code register `CC`; its value is `100`, `010`, or `001`. The 3 bits are named `N`, `Z`, and `P` (left-to-right), short for `Negative`, `Zero`, and `Positive`, and exactly one of `N`, `Z`, and `P` is one at any given time. "`CC` is `N`" or "`N = 1`" means `CC` = `100`, etc.
- The condition code is set automatically. On boot-up, `Z = 1`. Every time we do a load or calculation instruction (`LD`, `LDI`, `LDR`, `LEA`, `NOT`, `ADD`, or `AND`), the LC-3 checks the value being copied to the destination register and sets `N`, `Z`, or `P` accordingly.[1]
- **Mask**: The `BR` instruction contains a 3-bit mask; its bits correspond to the `NZP` bits of the condition code. To execute a `BR` instruction, the `CC`'s 3 `NZP` bits are bitwise ANDed with the instruction's 3 `NZP` mask bits. If the result is not `000`, the `PC` is set to `PC` + *offset*, so that we'll go the instruction at that address. (If the result is `000`, the PC is not changed and continues to point to the next instruction.) Note if the mask is `111`, we always jump; if the mask is `000`, we never jump.
  - If (`CC` AND `IR`[`9`:`11`]) ≠ `000` then `PC` ← `PC` + *offset*

[1] Technical note (won't make sense until we see the `TRAP` and subroutine call instructions): `HALT` sets `CC` to `P`; the other `TRAP`s set the `CC` by loading the return address into `R7` (addresses `x8000`, ..., `xFFFF` are treated as negative).

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
`BR`	0	0	0	0	N	Z	P	PCoffset9								

- **Versions of Branches:** We can control the kind of branch we want to do by how we set the branch mask. There are mnemonic codes for the 8 possible 3-bit masks. Most of them come from concatenating `BR` with some combination of `N`, `Z`, or `P` (in that order).
- In the table below, "`N`" means "negative" not "not," so `BRNZ` means "branch if negative or zero." To get branch on not zero, you use `BRNP` ("branch on negative or positive). The mnemonic "`NOP`" means "no operation" — no goto is done. For unconditional branch you can use `BR` or `BRNZP` (your choice).

Mask	***Mnemonic***	***Branch Condition***
000	`NOP`	Never
001	`BRP`	> 0
010	`BRZ`	= 0
011	`BRZP`	≥ 0
100	`BRN`	< 0
101	`BRNP`	≠ 0
110	`BRNZ`	≤ 0
111	`BR` or `BRNZP`	Unconditional

E. Examples of Using Branch Instructions

- Below, I'm use true/false "arm" of an if-else instead of true/false "branch", to keep from confusing them with the BR instruction.
- **Implementing if-then Statements**
  - An if-then is implemented by jumping around the true arm code if the if test fails.

- **Example**: Say we want `if R1 > 0 then` ...., where the `if` test should begin at `x3015` and the true arm ends at `x3040`:

```
x3015: 0001 010 001 1 00000 ; ADD R1, R1, 0
x3016: 0000 110 000101010   ; if R1 <= 0, skip true arm
x3017: ... true arm ...
       ...
x3041: ... first instruction after if-then ...
```

  - Note the `BR` at `x3016` needs an offset of `x3041` – `x3017` = `x2A`.
- **Implementing if-else Statements**
  - For an if-else, we extend the code for an if-then
    - If the test fails, `BR` around the true arm to go to the code for the false arm.
    - At the end of the true arm, we need a `BR` to jump past the end of the code for the false arm.
- **Example**: Say the if test should begin at `x3015`, the true arm ends at `x3040`, and the false arm ends at `x3050`, then:

```
x3015: 0001 010 001 1 00000 ; ADD R1, R1, 0
x3016: 0000 110 000101011   ; if R1 <= 0, go to false arm
x3017: ... true arm ...
       ...
x3041: 0000 111 000001111   ; skip false arm
x3042: ... false arm ...
...
x3051: ...  first instruction after if-else ...
```

  - Note the `BR` at `x3016` now has an offset of `x3042` – `x3017` = `x2B`; the `BR` at `x3041` has an offset of `x3051` – `x3042` = `xF`.

- **Implementing A Loop that Executes N times**
  - Here is pseudocode for looping with *counter* = N, $N-1$, $N-2$, ..., 1, ~~0~~. [fixed 3/31]

```
counter ← N
Top: if counter ≤ 0
        Quit loop (go to after the loop)
     else
        ... Loop Body ...
        --counter
        go to top of loop
After: ... code after the loop ...
```

- **Example**: Say `R2` holds the counter, N is stored at location `x3030`, the loop should begin at `x3010`, and the loop body ends at `x3020`.

```
x3010: 0010 010 000011111   ; LD R2, N
x3011: 0000 110 000001111   ; if ≤ 0, quit loop
x3012: ... Loop body ...
...
x301F: 001 010 010 1 11111  ; R2--
x3020: 0000 111 111110000   ; go to top of loop
...
x3030: ... value of N ...
```

  - For offsets, the load of N at `x3010` needs `x3030 – x3011` = `x1F`; the `BR` at x3011 to after the loop needs `x3021 – x3012` = `xF`; the `BR` at `x3020` to the top of the loop needs `x3021 – x3011` = `-x10`.

F. TRAP Instruction

- The `TRAP` instruction calls a **service routine** (an operating system routine). It's like calling a subroutine that's owned by the operating system. Control jumps to some code to handle the trap, and when that code finishes, control jumps to the instruction after the `TRAP` instruction (Exception: We don't return after the `HALT` trap.)
- Service routines are identified by an 8-bit **trap vector**. The hardware uses the trap vector to figure out where the OS code for that particular service is: The

location of the code to handle trap T is at memory location T. (The address of the code to handle **TRAP x20** is in M[x20], etc.) The table of addresses (**x0000** – **x00FF**) is called the **TRAP** table.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TRAP	1	1	1	1	0	0	0	0	trap vector							

- The pseudocode for executing the **TRAP** instruction is
  - R7 ← PC // Save location to return to (see below)
    PC ← M[*trap vector*] // Look up location of **TRAP** code; go there
- The R7 ← **PC** saves the location after the **TRAP** instruction. At the end of the **TRAP**-handling code, there's a **JMP R7** instruction that jumps ("through" R7) back to the user's code.
- For us, the interesting traps are:
  - **x20**: **GETC**: Input a character from keyboard into the rightmost byte of R0 (and clear the leftmost byte).
  - **x21**: **OUT**: Output the character in the rightmost byte of **R0** to the monitor (and ignore the leftmost byte).
  - **x22**: **PUTS**: Display the null-terminated string pointed to by **R0**.
  - **x23**: **IN**: Like **GETC** but prints a message before reading the character.
  - **x25**: **HALT**: Halt execution. (Clears the CPU's *running* flag.)
- **A note on TRAPs vs subroutines**: When we start writing subroutines, we'll use R7 the same way that **TRAP** does to return to the calling routine. If our subroutine code calls a **TRAP**, it will overwrite the R7 value we need to return to our caller. We'll have to save our R7 before the **TRAP** and restore it later.

Example: Read and Echo a Character

- Here's code to prompt for input with "**>** ", read one character, print it back out, and halt.
- **GETC** doesn't echo its input, so we print the character read in so that the user sees it after pressing the key.

```
x3000  1110 000 000000100  ; R0 ← address prompt string
x3001  1111 0000 0010 0010 ; PUTS (print prompt)
x3002  1111 0000 0010 0000 ; GETC (read char into R0)
x3003  1111 0000 0010 0001 ; OUT (print char in R0)
x3004  1111 0000 0010 0101 ; HALT
x3005  0000 0000 0011 1110 ; '>' = x3E
x3006  0000 0000 0010 0000 ; ' ' = x20
x3007  0000 0000 0000 0000 ; '\0' = x0
```

The LC-3 Computer, Part 2

CS 350: Computer Organization & Assembler Language Programming

A. Why?

- Control instructions let us implement decisions and loops.
- Trap instructions let us access operating-system-level routines like I/O.

B. Objectives

At the end of today, you should be able to

- Write LC-3 branch instructions to implement a loop or decision.
- Use the `TRAP` instruction to read/write a character or halt the program.

C. Questions

In Questions 1 – 4, the number of question marks doesn't indicate anything about the length of the answer.

1. What does the following (silly) instruction sequence do? (It depends on what's in `R1`.)

Addr	*Op*	*Value*	*Comments*
`x3000`	`AND`	`0101 011 001 1 11111`	`???`
`x3001`	`BR`	`0000 011 000000111`	`If ??? then ???`
`x3002`	`BR`	`0000 100 000000010`	`Else ???`
`x3003`	`...`	`...`	`(Do we ever reach this instruction?)`

2. Implement "`if R0 < 0 then R0 ← 0 else M[x30AC] ← R0`" by filling in the missing pieces of the instructions.

Addr	*Op*	*Value*	*Comments*
x3000	ADD	0001 000 000 ??????	Test value of R0
x3001	BRZP	0000 011 ?????????	If R0 ≥ 0, go to false arm
x3002	AND	0101 000 ?????????	R0 ← 0
x3003	BR	0000 ??? 000000001	Skip over false arm
x3004	ST	0011 000 ?????????	M[x30AC] ← R0
x3005	...		[1st location after if-else]

3. Implement "`if R7 = 1 then go to x5000`" by filling in the missing pieces of the instructions. The code uses `R1` as a temporary register.

Addr	*Op*	*Value*	*Comments*
x3000	ADD	0001 001 ?????????	R1 ← R7 − 1
x3001	BRNP	0000 ??? 000000011	If R7 ≠ 1, skip over jump
x3002	LD	0010 001 000000001	. R1 ← Location to jump to
x3003	JMP	1100 000 001 00000	. Jump to x5000
x3004		x5000	(Location to jump to)
x3005	...		[1st location after if]

4. Fill in the missing pieces of the instructions below to implement "if `R0 ≤ 0` then go to the location pointed to by `x3150`; if `R0 = 1 x3160`; otherwise go to `x3165`". The code uses `R1` as a temporary register.

Addr	*Op*	*Value*	*Comments*
x3100	AND	0101 000 000 ???	Test R0 (by setting R0 ← R0)
x3101	BRP	0000 001 ???	if R0 ≤ 0 then
x3102	LD	0010 000 ???	. Get location pt'd to by x3150
x3103	JMP	1100 000 000 00000	. Jump to location pt'd to by x3150
x3104	ADD	0001 ???	else R1 ← R0−1
x3105	BRZ	???	. if R0 = 1 go to x3160
x3106	BR	???	. else (R0 ≥ 2) go to x3165

Solution

1.

Addr	*Op*	*Value*	*Comments*
x3000	AND	0101 011 001 1 11111	R3 ← R1 AND -1 = R1
x3001	BR	0000 011 000000111	If R1 ≥ 0 then go to x3009
x3002	BR	0000 100 000000010	Else (R1 < 0) go to x3004
x3003	...	...	(Do we ever reach this instruction?) No

2.

Addr	*Op*	*Value*	*Comments*
x3000	ADD	0001 000 000 1 00000	Test value of R0
x3001	BRZP	0000 011 000000010	If R0 ≥ 0, go to false arm
x3002	AND	0101 000 000 1 00000	R0 ← 0 (note we can use any register for the underlined one)
x3003	BR	0000 111 000000001	Skip over false arm
x3004	ST	0011 000 010100111	M[x30AC] ← R0
x3005	...		[1st location after if-else]

3.

Addr	*Op*	*Value*	*Comments*
x3000	ADD	0001 001 111 1 11111	R1 ← R7 – 1
x3001	BRNP	0000 101 000000011	If R7 ≠ 1, skip over jump
x3002	LD	0010 001 000000001	. R1 ← Location to jump to
x3003	JMP	1100 000 001 00000	. Jump to x5000
x3004		x5000	(Location to jump to)
x3005	...		[1st location after if]

4.

Addr	*Op*	*Value*	*Comments*
x3100	AND	0101 000 000 1 11111	Test R0 (by setting R0 ← R0)
x3101	BRP	0000 001 000000010	if R0 ≤ 0 then
x3102	LD	0010 000 001001101	. Get location pt'd to by x3150
x3103	JMP	1100 000 000 00000	. Jump to location pt'd to by x3150
x3104	ADD	0001 001 000 1 11111	else R1 ← R0–1
x3105	BRZ	0000 010 001011001	. if R0 = 1 go to x3160
x3106	BR	0000 111 001011101	. else (R0 ≥ 2) go to x3165