The LC-3 Computer, Part 2

Control and TRAP Instructions

CS 350: Computer Organization & Assembler Language Programming

[4/6: p.4]

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 the target address using PC-offset so you can only do a short-distance jump with one.
  - The `JMP` (jump) instruction specifies the target using a base register, so you can jump anywhere ("long-distance" jump).
- **Conditional and Unconditional go-to**
  - On LC-3, `JMP` is unconditional; `BR` is conditional (though "always" and "never" are possible conditions).
- **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 is a short-distance conditional go-to. The target of the go-to is calculated using PC-offset addressing (so about ± 255 addresses).
- The branch instruction `BR` contains a **mask** field, which it combines with the current **condition code** to decide whether or not to do a go to.
- **Condition code**: There is a three-bit condition code register `CC`; its value is `100`, `010`, or `001`. The three 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 three-bit mask; its bits correspond to the `NZP` bits of the condition code. To execute a `BR` instruction, the `CC`'s three `NZP` bits are bitwise ANDed with the instruction's three `NZP` mask bits. If the result ≠ `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 execution continues with the next instruction.)
  - If (`CC & 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. E.g., if the mask is `011`, then we will branch if the `CC` is `010` or `001` (but not `100`). So we branch if the `CC` is `Z` or `P`, hence "branch if ≥ 0".
- Note if the mask is `111`, we always jump; if the mask is `000`, we never jump.
- There are mnemonic codes for the 8 possible three-bit masks. In general, you concatenate `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
`100`	`BRN`	< 0
`010`	`BRZ`	= 0
`001`	`BRP`	> 0
`110`	`BRNZ`	≤ 0
`101`	`BRNP`	≠ 0
`011`	`BRZP`	≥ 0
`111`	`BR` or `BRNZP`	Unconditional

E. Examples of Using Branch Instructions

- Below, I'll use "**true/false arm**" of an ***`if-then`***/***`if-else`*** statement instead of "true/false branch", to keep from confusing arms with the `BR` instruction.

If-then Statement

- For an ***`if-then`*** statement, we need to jump around the true arm code whenever the ***`if`*** test fails. If the test succeeds, we fall into the true arm code.
- **Example 1**: Say we want `if R1 > 0 then` …., where the ***`if`*** test should begin at `x3015` and the true arm ends at `x3040`. Note the `BR` at `x3016` needs a PC offset of `x3041 – x3017` = `x2A`.

Addr	*Value*	*Asm*	*Action/Comment*
`x3015`	`0001 010 001 1 00000`	`ADD R1,R1,0`	`Test R1 [fixed 4/6]`
`x3016`	`0000 110 000101010`	`BRNZ x2A`	`if R1 <= 0, go to x3041`
`x3017`	`...`		
`...`	`...`		`(True arm)`
`x3041`	`...`		`(Code after if-then)`

While Loop

- A ***`while`*** loop is like an ***`if-then`*** statement in that we need to jump around the loop body code if the ***`while`*** test fails. If the test succeeds, we fall into the loop body code; at the end of the loop body, we branch back up to the top of the loop (the ***`while`*** test).
- **Example 2**: Say we want ***`while`*** `R1 > 0 do` …., where our code should begin at `x3015` and the loop body ends at `x3040`.
  - Below, the first instruction (R1 ← R1) just sets the condition code according to the value of R1. If the condition code has already been set, we can omit that statement. The branch at `x3016` needs a PC offset of `x3041 – x3017` = `x2A`; the branch at `x3040` needs `x3016 – x3041` = `–x2B` = -43_{10}.

Addr	*Value*	*Asm*	*Action/Comment*
`x3015`	`0001 001 001 1 00000`	`ADD R1,R1,0`	`Test R1`
`x3016`	`0000 110 000101010`	`BRNZ x2A`	`Top: if R1 ≤ 0, go to x3041`
`x3017`	`...`		
`...`	`...`		`(Loop body)`
`x3040`		`BR -43`	`go to Top (of loop)`
`x3041`	`...`		`(Code after loop)`

If-Else Statement

- For an ***if-else*** statement, we need two branch instructions.
  - If the test fails, `BR` around the true arm to the beginning of the false arm.
  - If the test succeeds, fall into the true arm code; at the end of the true arm, `BR` around the code for the false arm.
- **Example 3**: Say we want ***if*** `R1 > 0` ***then*** …. ***else*** …, where the code should begin at `x3015`, the true arm ends at `x3040`, and the false arm ends at `x3050`. Note the branch at `x3016` needs PC offset `x3042 – x3017` = `x2B`; the branch at `x3041` needs offset `x3051 – x3042` = `xF`.

Addr	***Value***	***Asm***	***Action/Comment***
`x3015`	`0001 001 001 1 00000`	`ADD R1,R1,0`	`Test R1`
`x3016`	`0000 110 000101011`	`BRNZ x2B`	`if R1 <= 0, skip true arm`
`x3017` `...`	`...` `...`		`(True arm)`
`x3041`	`0000 111 000001111`	`BR 15`	`skip false arm`
`x3042` `...`	`...` `...`		`(False arm)`
`x3051`			`(Code after if-else)`

Implementing A Loop that Executes N times:

- Here is pseudocode for looping with *counter* = *N*, *N*−1, *N*−2, …, 1.

  counter ← *N*
  `Top:` ***if*** *counter* ≤ `0`
          go to `After` (*exit loop*)
      else
          … Loop Body …
          `--`*counter*`;`
          go to `Top` (*continue loop*)
  `After`: *… code after the loop …*

- **Example 4**: Say `R2` holds the counter, the loop should begin at `x3010`, end at `x3020`, and *N* is stored at location `x3030`. The load of *N* at `x3010` needs

PC offset `x3030 – x3011` = `x1F`; the branch at `x3011` needs `x3021 – x3012` = `xF`; the branch at `x3020` needs `x3021 – x3011` = `–16`.

Addr	***Value***	***Asm***	***Action/Comment***
`x3010`	`0010 010 000011111`	`LD R2,x1F`	`R2 ← N`
`x3011`	`0000 110 000001111`	`BRNZ 15`	`Top: if R2 ≤ 0, exit loop`
`x3012`	`...`		`(Loop body)`
`...`	`...`		
`x301F`	`0001 010 010 1 11111`	`ADD R2,R2,-1`	`R2--`
`x3020`	`0000 111 111110000`	`BR -16`	`(bottom of loop) go to Top`
…			`(Code after loop)`
`x3030`	`...`		`Value of N`

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 **`TRAP`** handler 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							

- Version 1 of the pseudocode for executing the **`TRAP`** instruction is below. (We'll see version 2 when we discuss supervisor mode vs user mode execution.)
  - `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.
- The simple assembler instruction is **`TRAP`** *n* where *n* is the trap code, typically a hex number:
  - `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.

Addr	*Value*	*Asm*	*Action/Comment*
`x3000`	`1110 000 000000100`	`LEA R0,4`	`Pt R0 to prompt string`
`x3001`	`1111 0000 0010 0010`	`TRAP x22`	`PUTS (print prompt)`
`x3002`	`1111 0000 0010 0000`	`TRAP x20`	`GETC (read char into R0)`
`x3003`	`1111 0000 0010 0001`	`TRAP x21`	`OUT (print char in R0)`
`x3004`	`1111 0000 0010 0101`	`TRAP x25`	`HALT`
`x3005`	`0000 0000 0011 1110`		`Prompt: '>' = x3E`
`x3006`	`0000 0000 0010 0000`		`' '`
`x3007`	`0000 0000 0000 0000`		`end of prompt`

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 necessarily indicate anything about the length of the answer.

1. Fill in the missing pieces of the (silly) instruction sequence. What does it do? (It depends on what's in `R1`.)

Addr	***Value***	***Asm***	***Action/Comment***
`x3000`	`0101 011 001 1 11111`	`AND R3,R1,-1`	`???`
`x3001`	`0000 011 000000111`	`BRZP ???`	`If ??? then ???`
`x3002`	`0000 010 000000010`	`BRN  ???`	`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	***Value***	***Asm***	***Action/Comment***
x4000	0001 000 000 ??????	ADD R0,R0,???	Test value of R0
x4001	0000 011 ?????????	BRZP ???	If R0 ≥ 0, go to false arm
x4002	0101 000 ?????????	AND R0,R0,0	. R0 ← 0
x4003	0000 ??? 000000001	BR??? 1	Skip over false arm
x4004	0011 000 ?????????	ST R0,???	. M[x40AC] ← R0
x4005	...		(code after if-else)

3. Implement "`if R7 = 1 then go to x5000`". (`R1` is a temporary register.)

Addr	***Value***	***Asm***	***Action/Comment***
x8000	0001 001 ?????????	ADD R1,R7,-1	R1 ← R7 − 1
x8001	0000 ??? 000000011	BR?? ???	If R7 ≠ 1, go to end if
x8002	0010 001 000000001	LD R1,???	. R1 ← Target location
x8003	1100 000 001 00000	JMP R1	. Jump to target
x8004	x5000		Location of target
x8005	...		(code after if-then)

4. Implement "if `R0 ≤ 0` then go to the location pointed to by `x3150`; else if `R0 = 1` go to `x3160`; else go to `x3165`". (`R1` is a temporary register.)

Addr	***Value***	***Asm***	***Action/Comment***
x3100	0001 000 ?????	ADD R0,???,0	Test R0
x3101	0000 001 ???	BRP ???	if R0 ≤ 0 then
x3102	0010 000 ???	LD R0,???	. Get loc pt'd to by x3150
x3103	1100 000 000 00000	JMP R0	. Jump to that location
x3104	0001 001 000 ????	ADD R1,R0,-1	else R1 ← R0 - 1
x3105	0000 ?????	BRZ ???	. if R0 = 1 go to x3160
x3016	000 111 ?????	BR ???	. else go to x3165
...			(x49 words)
x3150	(address to jump to)		

Solution

1. If `R1 ≥ 0` then go to `x3009` else go to `x3004`.

Addr	*Value*	*Asm*	*Action/Comment*
x3000	0101 011 001 1 11111	AND R3,R2,-1	R3 ← R1 & -1 = R1
x3001	0000 011 000000111	BRNP 7	if R1 ≠ 0 then go to x3009 [11/10]
x3002	0000 100 000000010	BRN 2	else (R1 < 0) go to x3004
x3003	...	...	(We don't get here)

2. (Implement `if R0 < 0 then R0 ← 0 else M[x30AC] ← R0`)

Addr	*Value*	*Asm*	*Action/Comment*
x4000	0001 000 000 1 00000	ADD R0,R0,0	Test value of R0
x4001	0000 011 000000010	BRZP 2	If R0 ≥ 0, go to false arm
x4002	0101 000 000 1 00000	AND R0,R0,0	. R0 ← 0
x4003	0000 111 000000001	BR 1	Skip over false arm
x4004	0011 000 0 1010 0111	ST R0,xA7	. M[x40AC] ← R0
x4005	...		(code after if-else)

3. Implement "`if R7 = 1 then go to x5000`". (`R1` is a temporary register.)

Addr	*Value*	*Asm*	*Action/Comment*
x8000	0001 001 111 1 11111	ADD R1,R7,-1	R1 ← R7 – 1
x8001	0000 101 000000011	BRNP 3	If R7 ≠ 1, go to end if
x8002	0010 001 000000001	LD R1,1	. R1 ← Target location
x8003	1100 000 001 00000	JMP R1	. Jump to target
x8004	x5000		Location of target
x8005	...		(code after if-then)

4. Implement “if `R0 ≤ 0` then go to the location pointed to by `x3150`; else if `R0 = 1` go to `x3160`; else go to `x3165`”. (`R1` is a temporary register.)

Addr	***Value***	***Asm***	***Action/Comment***
`x3100`	`0001 000 000 1 00000`	`ADD R0,R0,0`	`Test R0`
`x3101`	`0000 001 000000010`	`BRP 2`	`if R0 ≤ 0 then`
`x3102`	`0010 000 001001101`	`LD R0,x4D`	`. Get loc pt’d to by x3150`
`x3103`	`1100 000 000 00000`	`JMP R0`	`. Jump to that location`
`x3104`	`0001 001 000 1 11111`	`ADD R1,R0,-1`	`else R1 ← R0 - 1`
`x3105`	`0000 010 001011010`	`BRZ x5A`	`. if R0 = 1 go to x3160`
`x3016`	`0000 111 001011110`	`BR  x5E`	`. else go to x3165`
`...`			`(x49 words)`
`x3150`	`(address to jump to)`		