Subroutines

Michael Dann
School of Computing Technologies

What's next...

LC3 overview

Instruction Set

Op	Format	Description	Example
ADD	ADD DR, SR1, SR2 ADD DR, SR1, imm5	Adds the values in SR1 and SR2/imm5 and sets DR to that value.	ADD R1, R2, #5 The value 5 is added to the value in R2 and stored in R1.
AND	AND DR, SR1, SR2 AND DR, SR1, imm5	Performs a bitwise and on the values in SR1 and SR2/imm5 and sets DR to the result.	AND R0, R1, R2 A bitwise and is preformed on the values in R1 and R2 and the result stored in R0.
BR	BR(n/z/p) LABEL Note: (n/z/p) means any combination of those letters can appear there, but must be in that order.	Branch to the code section indicated by LABEL, if the bit indicated by (n/z/p) has been set by a previous instruction. n: negative bit, z: zero bit, p: positive bit. Note that some instructions do not set condition codes bits.	BRz LPBODY Branch to LPBODY if the last instruction that modified the condition codes resulted in zero. BRnp ALT1 Branch to ALT1 if last instruction that modified the condition codes resulted in a positive or negative (non-zero) number.
JMP	JMP SR1	Unconditionally jump to the instruction based upon the address in SR1.	JMP R1 Jump to the code indicated by the address in R1.
JSR	JSR LABEL	Put the address of the next instruction after the JSR instruction into R7 and jump to the subroutine indicated by LABEL.	JSR POP Store the address of the next instruction into R7 and jump to the subroutine POP.
JSRR	JSSR SR1	Similar to JSR except the address stored in SR1 is used instead of using a LABEL.	JSSR R3 Store the address of the next instruction into R7 and jump to the subroutine indicated by R3's value.
LD	LD DR, LABEL	Load the value indicated by LABEL into the DR register.	LD R2, VAR1 Load the value at VAR1 into R2.
LDI	LDI DR, LABEL	Load the value indicated by the address at LABEL's memory location into the DR register.	LDI R3, ADDR1 Suppose ADDR1 points to a memory location with the value x3100. Suppose also that memory location x3100 has the value 8. 8 then would be loaded into R3.
LDR	LDR DR, SR1, offset6	Load the value from the memory location found by adding the value of SR1 to offset6 into DR.	LDR R3, R4, #-2 Load the value found at the address (R4 –2) into R3.
LEA	LEA DR, LABEL	Load the address of LABEL into DR.	LEA R1, DATA1 Load the address of DATA1 into R1.
NOT	NOT DR, SR1	Performs a bitwise not on SR1 and stores the result in DR.	NOT R0, R1 A bitwise not is preformed on R1 and the result is stored in R0.
RET	RET	Return from a subroutine using the value in R7 as the base address.	RET Equivalent to JMP R7.
RTI	RTI	Return from an interrupt to the code that was interrupted. The address to return to is obtained by popping it off the supervisor stack, which is automatically done by RTI.	RTI Note: RTI can only be used if the processor is in supervisor mode.
ST	ST SR1, LABEL	Store the value in SR1 into the memory location indicated by LABEL.	ST R1, VAR3 Store R1's value into the memory location of VAR3.
STI	STI SR1, LABEL	Store the value in SR1 into the memory location indicated by the value that LABEL's memory location contains.	STI R2, ADDR2 Suppose ADDR2's memory location contains the value x3101. R2's value would then be stored into memory location x3101.
STR	STR SR1, SR2, offset6	The value in SR1 is stored in the memory location found by adding SR2 and offset6 together.	STR R2, R1, #4 The value of R2 is stored in memory location (R1 + 4).
TRAP	TRAP trapvector8	Performs the trap service specified by trapvector8. Each trapvector8 service has its own assembly instruction that can replace the trap instruction.	TRAP x25 Calls a trap service to end the program. The assembly instruction HALT can also be used to replace TRAP x25.

Covered already

Covered later

Covered today

Not covered

Unconditional Jumps

JMP	JMP SR1	Unconditionally jump to the instruction based upon the address in SR1.	JMP R1 Jump to the code indicated by the address in R1.

```
.ORIG x3000

AND R0, R0, #0

ADD R0, R0, #1
LEA R1, some_later_point_in_the_program
JMP R1

ADD R0, R0, #1
some_later_point_in_the_program
HALT
.END
```

By the end of this program, R0 will contain 1, not 2, because the second ADD instruction is skipped over.

Subroutines

- Let’s say we want to multiply lots of different pairs of numbers.
- Writing the logic repeatedly is no fun…
- Instead, put the code inside a *subroutine*, allowing us to reuse it whenever we want to perform multiplication.
- Same idea as "methods" (Java) and "functions" (Python).
- We *could* try to use `JMP` to implement this idea, but it would be a bit tedious. We’d have to put labels all throughout our code…

Subroutines

The instruction `JSR` (“jump to subroutine”) provides an easier approach.

It jumps to a label but stores the address of the next instruction after the JSR into a register (R7), so that we can return to where we were without having to label it.

Calling `RET` at the end of the subroutines takes us back to where we were.

JSR	JSR LABEL	Put the address of the next instruction after the JSR instruction into R7 and jump to the subroutine indicated by LABEL.	JSR POP Store the address of the next instruction into R7 and jump to the subroutine POP.

RET	RET	Return from a subroutine using the value in R7 as the base address.	RET Equivalent to JMP R7.

Subroutine Example

```
1          LDI R0, N     ; Argument N is now in R0
2          JSR F         ; Jump to subroutine F.
3          STI R1, FN
4          HALT
5  N       .FILL 3120    ; Address where n is located
6  FN      .FILL 3121    ; Address where fn will be stored.
7                        ; Subroutine F begins
8  F       AND R1, R1, x0 ; Clear R1
9          ADD R1, R0, x0 ; R1 ← R0
10         ADD R1, R1, R1 ; R1 ← R1 + R1
11         ADD R1, R1, x3 ; R1 ← R1 + 3.  Result is in R1
12         RET            ; Return from subroutine
13         END
```

Listing 5.1: A subroutine for the function $f(n) = 2n + 3$.

Being careful with subroutines (!)

- JSR and RET rely on storing the return address in R7.
- If you mess with the value in R7, the program isn't going to return to the right place!
- Calling a subroutine from within a subroutine will mess things up!
- Hacky solution: Save the value in R7 to a temporary storage location, then restore it later.
- Better solution: Stack (unfortunately not in LC-3).

Subroutine example

The integers A and B are stored at locations **x3100** and **x3101**, respectively.
The integers C and D are stored at locations **x3110** and **x3111**, respectively.

Write an LC-3 assembler program that:

- Calculates A * B and stores the result at **x3102**.
- Calculates C * D and stores the result at **x3112**.

Use a subroutine to avoid writing out the multiplication logic twice.

Multiplication starter code

```
; Multiplication example.

.ORIG x3000

; Load the values to be multiplied.
LDI R0, address_a
LDI R1, address_b

; Prepare R2 for holding the result by setting its value to 0.
AND R2, R2, #0

; Calculate A * B via B + B + ... + B (a total of A times).
mul_loop
    ADD R2, R2, R1  ; Add B to the running total
    ADD R0, R0, #-1 ; We now need one less B to be added.
BRp mul_loop        ; Keep looping while there are Bs to be added.

STI R2, address_result

HALT

address_a .FILL x3100
address_b .FILL x3101
address_result .FILL x3102
.END
```

More subroutine practice

The integers X and Y are stored at locations **x3100** and **x3101**, respectively.

Write an LC-3 assembler program that calculates X^Y (X to the power of Y) and stores the result at **x3102**.

Strategy:
X^Y = X * X * … * X (a total of Y times)