university of groningen

Computer Architecture 2023-24 (WBCS010-05)

Lecture 10: Problem Solving

Reza Hassanpour
r.zare.hassanpour@rug.nl

Problem Solving

- To solve problems using computers we have to write programs.
- Decomposing the problem into smaller tasks simplifies developing programs
- Structured programming is used to simplify decomposing problems into smaller tasks by using
  - control flow constructs (if/then/else)
  - repetition (while and for),
  - block structures,
  - and subroutines.

Systematic Decomposition

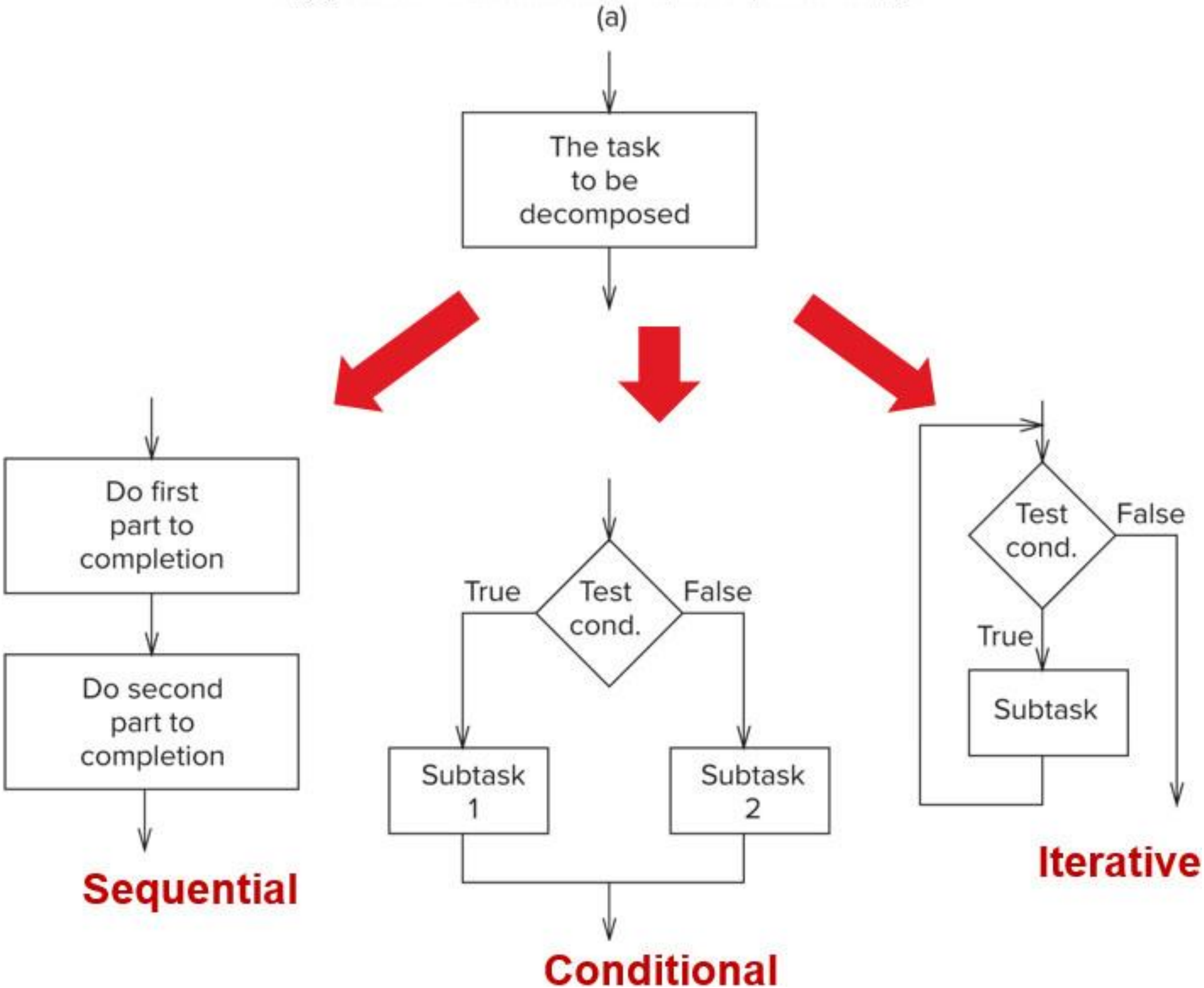
- Start with problem statement:

  "We wish to count the number of occurrences of a character in a file. The character in question is to be input from the keyboard; the result is to be displayed on the monitor"

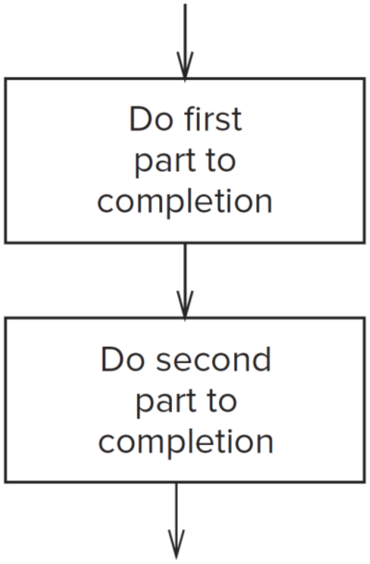
- **Decompose** task into a few simpler subtasks

- Decompose each subtask into smaller subtasks, and these into even smaller subtasks, etc....until you get to the level of a machine instruction or a small group of instructions

Structured Programming: Three Basic Constructs

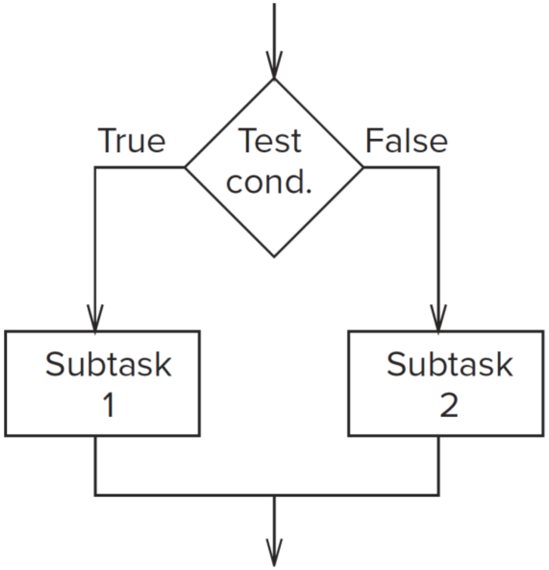


Three Basic Constructs

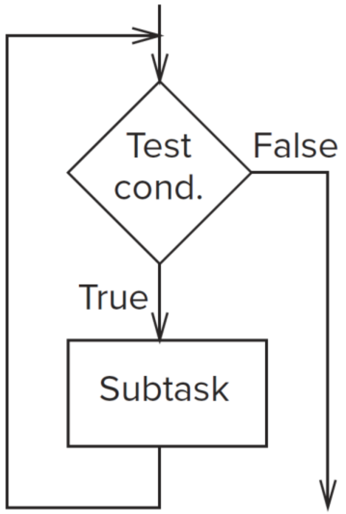


- **Sequential:** Break into two subtasks, both of which must be performed. Do the first task to completion, then the second.

- **Conditional:** Task consists of two subtasks, one of which should be performed and the other not performed. Test condition to choose, then perform subtask to completion. (Either subtask may be empty).

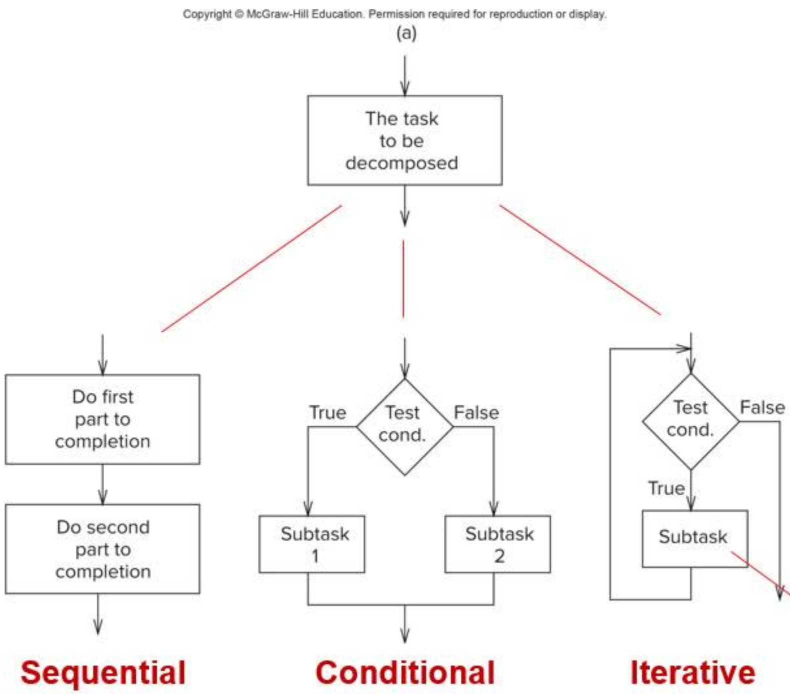


- **Iterative:** Perform a single subtask multiple times, but only if condition is true. When task is done, go back to retest the condition. When the condition is not true, program proceeds.



Structured Programming = Stepwise Refinement

- Replace this…



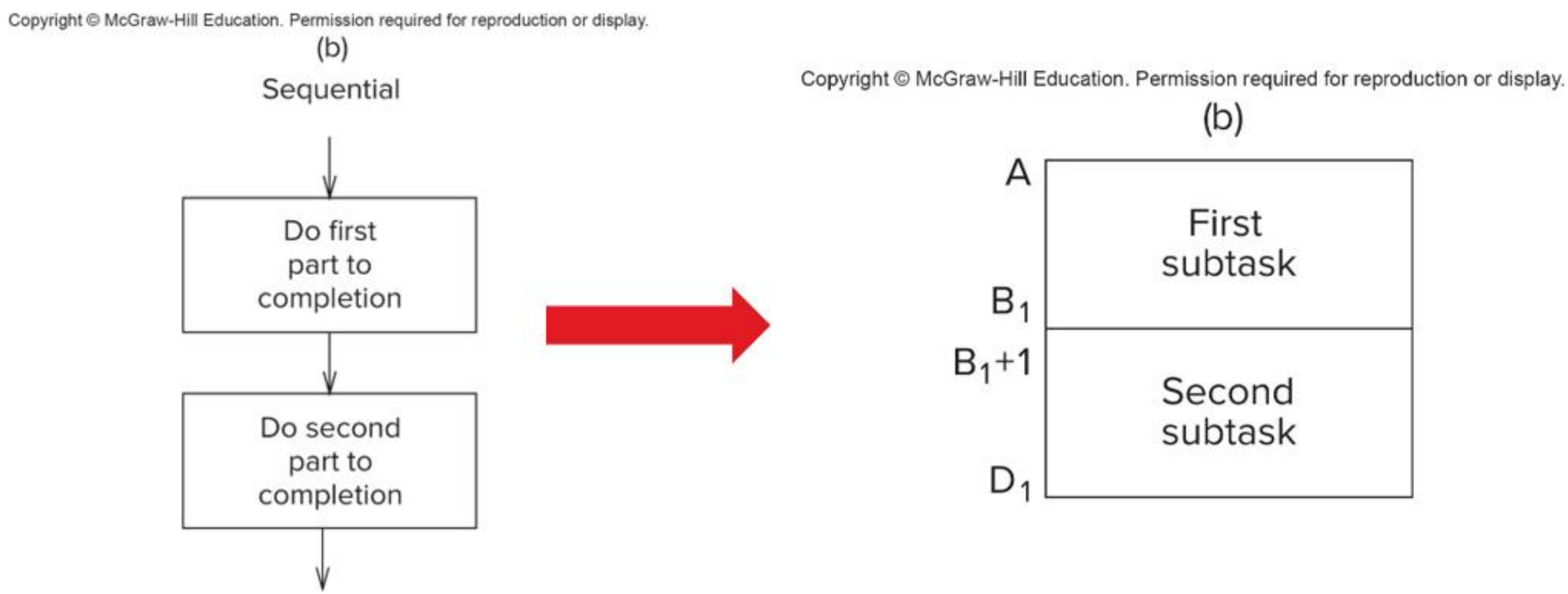
- …with one of these.

And then replace each subtask, etc., etc.

- **NOTE:** In each case, there are exactly one entry and one exit. This means any task / subtask (rectangle) can be replaced by any of the basic constructs.

Implementing Constructs with LC-3 Instructions 1

› **Sequential**

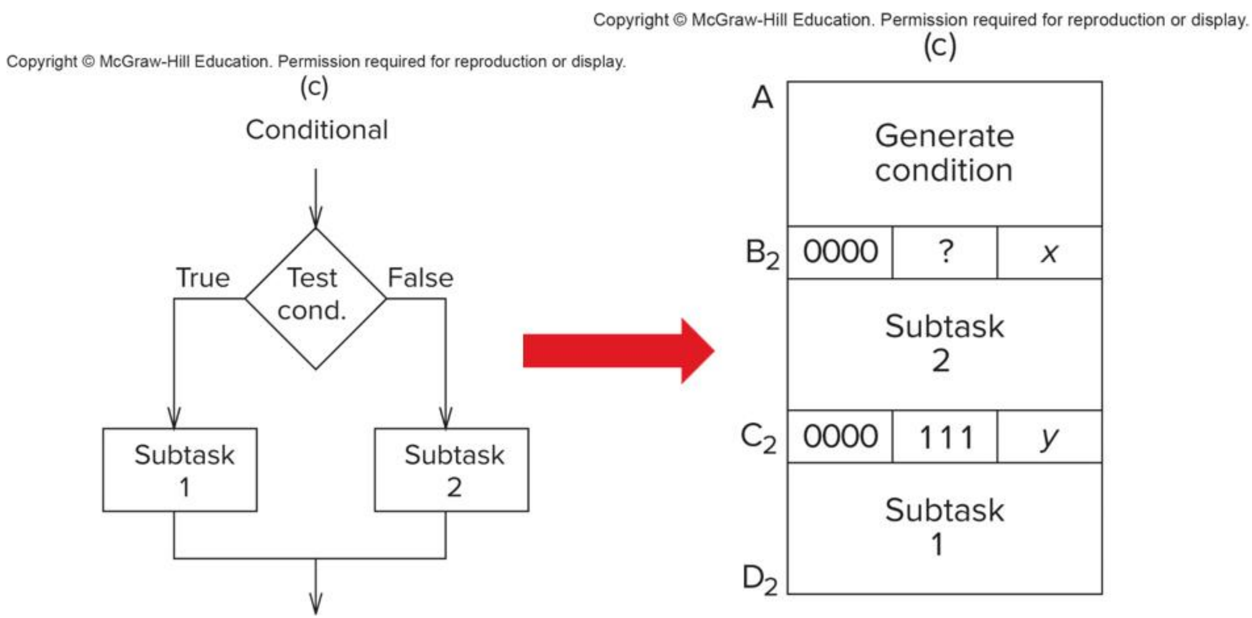


› No control instructions needed. Execution flows sequentially from one subtask to the next

› Instructions from A to B1 for first subtask
Instructions from B1+1 to D1 for second subtask

Implementing Constructs with LC-3 Instructions 2

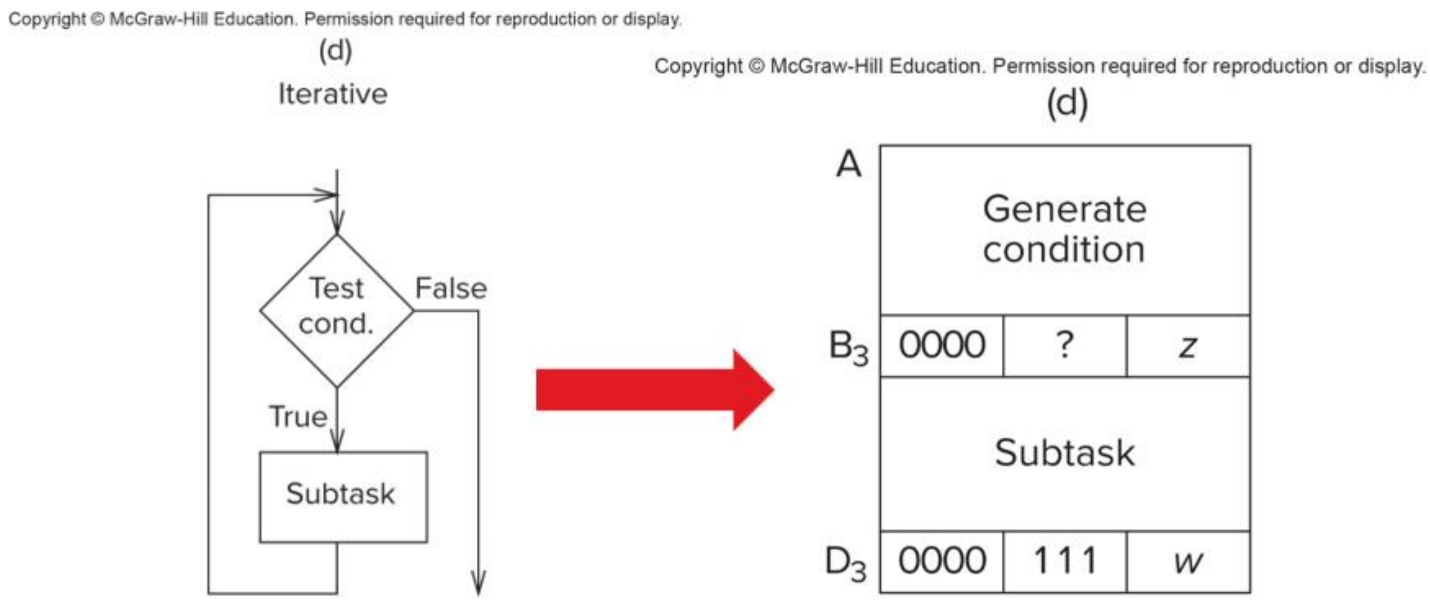
› **Conditional**



- First (A), instructions that will set condition codes according to condition
- Branch at B2 -- taken if condition is true, target = C2+1 (subtask 1)
- If condition is not true, branch is not taken, and subtask 2 (B2+1...) is executed. After subtask 2, C2 is unconditional branch to D2+1, so that subtask 1 is not executed

Implementing Constructs with LC-3 Instructions 3

- **Iterative**



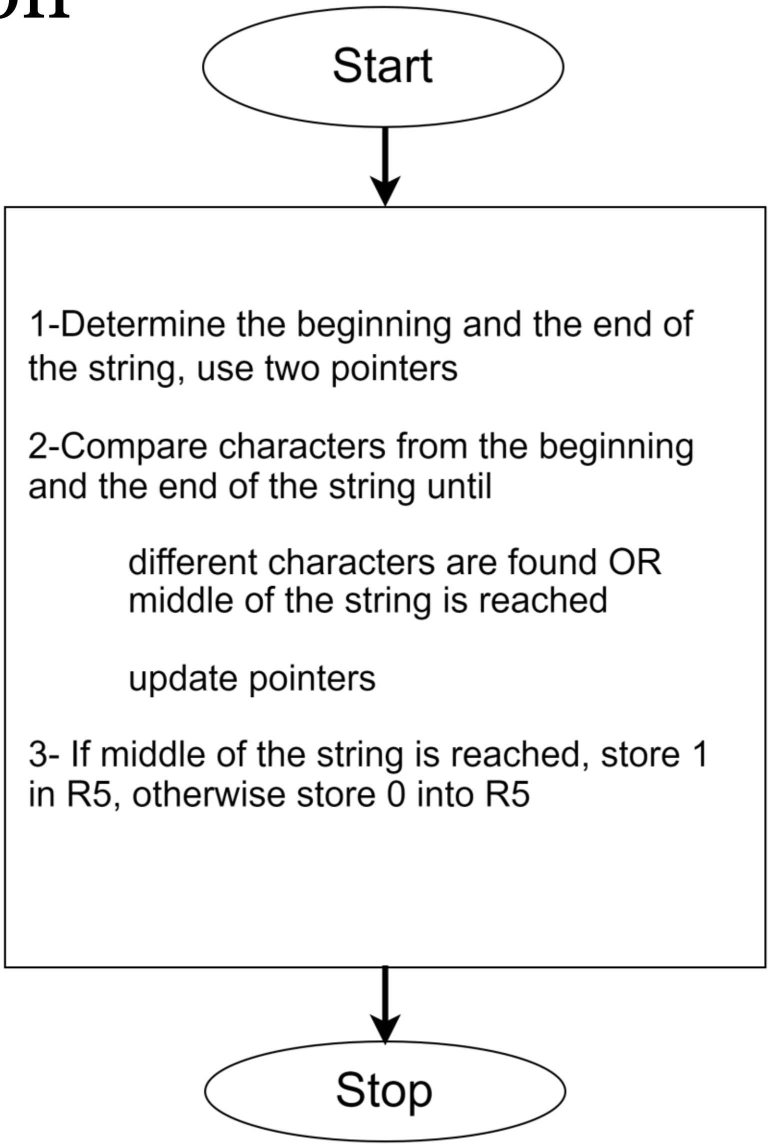
- First (A), instructions that will set condition codes according to condition
- Branch at B3 -- taken if condition is false, target = D3+1
- If condition is true, branch is not taken, and subtask (B3+1…) is executed. After subtask, D3 is unconditional branch to A, to check the condition again

Example: Palindrome

- Determines whether a character string is a palindrome or not.
- A palindrome is a string that reads the same backwards as forwards.
- Example: "racecar" is a palindrome.
- Suppose a string starts at memory location x4000 and is in the .STRINGZ format.
- If the string is a palindrome, the program terminates with the value 1 in R5. If not, the program terminates with the value 0 in R5.

Description of a Possible Solution

- Describe the solution to the problem with some details
- Decompose the solution into tasks



Task A: Sequential

- Determine a pointer to the beginning of the string and load the first character

- Initialize
- Load the first character

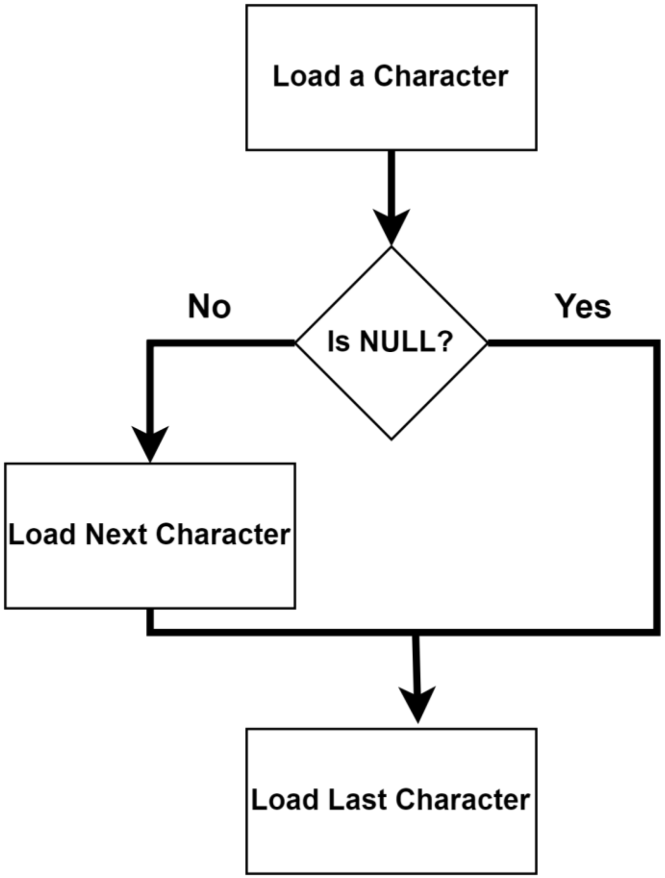
```
LD R0, PTR
LDR R3, R0, #0
```

Task B: Iterative

- Determine a pointer to the end of the string and load the first character

- Initialize
- Load the first character

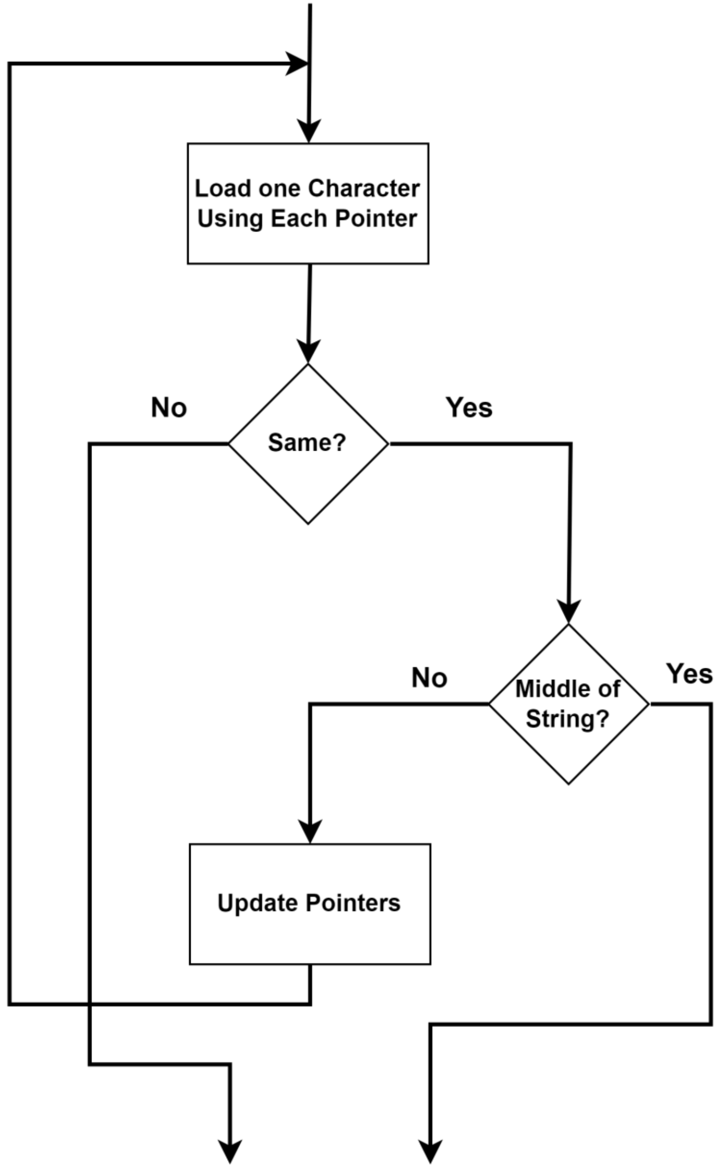
```
      ADD R1, R0, #0
AGAIN LDR R2, R1, #0
      BRz CONT
      ADD R1, R1, #1
      BRnzp AGAIN
CONT  ADD R1, R1, #-1
      LDR R4, R1, #0
```



Task C: Iterative

› Load characters shown by two pointers

```
LOOP LDR R4, R1, #0
        LDR R3, R0, #0
        NOT R4, R4
        ADD R4, R4, #1
        ADD R3, R3, R4
        BRnp NO
;Determine if pointers reached the middle of the string
        NOT R2, R0
        ADD R2, R2, #1
        ADD R2, R1, R2
        BRnz YES
        ADD R0, R0, #1
        ADD R1, R1, #-1
        BRnzp LOOP
```



Task D: Conditional

- If palindrome, load 1 into R5
  else load 0 into R5

```
YES     AND R5, R5, #0
        ADD R5, R5, #1
        BRnzp DONE
NO      AND R5, R5, #0
DONE    HALT
```

Last Step: LC3 Instructions (Assembly)

```
        .ORIG x3000
        LD R0, PTR
        ADD R1, R0, #0
AGAIN LDR R2, R1, #0
        BRz CONT
        ADD R1, R1, #1
        BRnzp AGAIN
; End of the string (null) is found and is
; pointed to by R1
; Subtract 1 from R1 to point to the last
;character
CONT ADD R1, R1, #-1
LOOP LDR R4, R1, #0
        LDR R3, R0, #0
        NOT R4, R4
        ADD R4, R4, #1
        ADD R3, R3, R4
        BRnp NO
;Determine if pointers reached the
;middle of the string
    NOT R2, R0
    ADD R2, R2, #1
    ADD R2, R1, R2
    BRnz YES
    ADD R0, R0, #1
    ADD R1, R1, #-1
    BRnzp LOOP
YES AND R5, R5, #0
    ADD R5, R5, #1
    BRnzp DONE
NO AND R5, R5, #0
DONE HALT
PTR .FILL x4000
    .END
```

Debugging

› Traditionally, a computer error is known as a "bug," and the process of finding and removing errors is known as "debugging"

› How do you know there's an error?

- Run the program with a set of inputs for which the desired output is known or can be confirmed (this is called "testing").
- Testing with a variety of inputs is essential.

- Sometimes, the behavior of the program itself shows that there's an error. For example, the program runs "forever" without halting. Or a desired output is never displayed on the monitor.

Steps for Debugging

- **Trace** the program
- Record the sequence of events as the program executes
- Check the results of each sequence of events -- are they consistent with what you expect?

- When you find a result that does not match your expectation, you can narrow down the location of the bug.

- Once the bug is fixed, you have to test again.
  There might be multiple bugs!!!

Debugging Tools

- A **debugger** is an environment for running your program that allows you to do the following:

1. Write values into registers and memory locations.
2. Execute a sequence of instructions.
3. Stop execution when desired (that is, before the program ends).
4. Examine values in memory and register at any point in the program.

- The **LC-3 simulator** allows you to run interactively in debugging mode, providing the capabilities shown above.

Examining and Changing Values

- The data for your program is stored in two locations: **memory** and **registers**.

- The simulator displays the values of all registers and provides a way to look at any memory location. You can also change the values stored in any of those locations.

- **NOTE:** Instructions are stored in memory, too, so you can change an instruction during the debugging session!

Executing a Sequence of Instructions

- To debug, you want to (1) set memory and registers as desired, (2) run an "interesting" sequence of instructions, and (3) check the values in memory/registers. Then repeat (2) and (3) until you find something interesting.
- There are two fundamental ways of controlling execution:
  - **single-stepping** allows you to execute one instruction at a time
  - **breakpoints** allow you to specify a particular instruction where you want to stop execution

Stepping vs. Breakpoints

- Using breakpoints is much faster, because a sequence of instructions can be executed quickly by the simulator. It's a good way to quickly narrow down the likely location of a problem.
- Set breakpoints between **modules** of your program -- for example: at the end of each task that you created when you designed the program.

- Once you've identified a sequence of interest:

1. **Set a breakpoint** at the beginning of that sequence
2. **Run** the program from the beginning, quickly executing instructions and stopping when the breakpoint is reached
3. **Single-step** through one instruction at a time to see exactly what the processor is doing

- (Instead of 1 and 2, you can set the PC to the start of your sequence and set up registers/memory with known initial values.)

Debugging Examples

- The best way to learn how to debug is to do it. You'll get plenty of practice, because it's rare to write a program correctly the first time!

- We will show four examples of faulty programs, and show how debugging can find the errors. The examples illustrate several common types of errors:

1. Executing a loop the wrong number of times
2. Using the wrong opcode
3. Testing the wrong condition code
4. Not considering all types of input values

Example 1: Multiply 1

- Program: Registers R4 and R5 contain positive integers. Multiply them and put the result in R2
- Code:

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
x3200	0	1	0	1	0	1	0	0	1	0	1	0	0	0	0	0	R2 <- 0
x3201	0	0	0	1	0	1	0	0	1	0	0	0	0	1	0	0	R2 <- R2 + R4
x3202	0	0	0	1	1	0	1	1	0	1	1	1	1	1	1	1	R5 <- R5 - 1
x3203	0	0	0	0	0	1	1	1	1	1	1	1	1	1	0	1	BRzp x3201
x3204	1	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1	HALT

- Each instruction has an annotation (comment) that tells what the instruction is supposed to do. These comments are not part of the program but are extremely helpful as you are trying to interpret the execution.

- Result: When R4 = 10 and R5 = 3, R2 gets a result of 40

Multiply in ASM and C

```
      AND R2, R2, #0
Loop  ADD R2, R2, R4
      ADD R5, R5, #-1
      BRzp Loop
      HALT
```

```
Sum=0;
do
{
    Sum = Sum + Num;
    Count = Count -1;
} while( Count >= 0 );
```

Example 1: Multiply 2

- First step: Read the code for awhile to see if it makes sense
- Warning: Don't do this too long. It's generally much easier to debug by executing the code and watching its actual behavior. But a careful look to better understand the code is a good practice.

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
x3200	0	1	0	1	0	1	0	0	1	0	1	0	0	0	0	0	`R2 <- 0`
x3201	0	0	0	1	0	1	0	0	1	0	0	0	0	1	0	0	`R2 <- R2 + R4`
x3202	0	0	0	1	1	0	1	1	0	1	1	1	1	1	1	1	`R5 <- R5 - 1`
x3203	0	0	0	0	0	1	1	1	1	1	1	1	1	1	0	1	`BRzp x3201`
x3204	1	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1	`HALT`

- R4 gets added to itself multiple times in R2, and R5 gets decreased each time. The loop ends when R5 is zero, so
- R2 = 0 + R4 + R4 + R4 = 0 + 10 + 10 + 10 = 30
- Looks reasonable -- why doesn't it work? Why do we get 40?

university of
groningen

Example 1: Multiply 3

› Next step: Trace execution of the code. We’ll be using **single-stepping**. The following table shows the PC and the state of the relevant registers after each instruction is executed.

(a)

PC	R2	R4	R5
x3201	0	10	3
x3202	10	10	3
x3203	10	10	2
x3201	10	10	2
x3202	20	10	2
x3203	20	10	1
x3201	20	10	1
x3202	30	10	1
x3203	30	10	0
x3201	30	10	0
x3202	40	10	0
x3203	40	10	−1
x3204	40	10	−1
	40	10	−1

› When R5 = 3, the loop (x3201 to x3203) is executed four times. It should only execute three times to add R4 the correct number of times.

› Error: We shouldn't re-execute the loop when R5 becomes zero. We should exit the loop.

› **Fix:** Change BRzp to BRp

Q: What (unfixed) result you would get if R4 = 3 and R5 = 10?

Example 1: Multiply 4

- Using Breakpoints
- Because loop errors are very common, it's often enough to stop execution once per iteration, rather than stepping through every instruction. Here's the trace when we set a breakpoint at the BR instruction.

  - (b)

PC	R2	R4	R5
x3203	10	10	2
x3203	20	10	1
x3203	30	10	0
x3203	40	10	-1

- Fewer stopping points means faster debugging and less chance to get lost in the details of the other instructions
- This allows us to quickly check: (a) how many times is the BR being taken? (b) what is the cause of the BR being taken / not taken?

Example 2: Add a Sequence of Numbers 1

- Program: Add ten numbers starting at x3100, putting the result in R1.
- Code:

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
x3000	0	1	0	1	0	0	1	0	0	1	1	0	0	0	0	0	R1 <- 0
x3001	0	1	0	1	1	0	0	1	0	0	1	0	0	0	0	0	R4 <- 0
x3002	0	0	0	1	1	0	0	1	0	0	1	0	1	0	1	0	R4 <- R4 + 10
x3003	0	0	1	0	0	1	0	0	1	1	1	1	1	1	0	0	R2 <- M[x3100]
x3004	0	1	1	0	0	1	1	0	1	0	0	0	0	0	0	0	R3 <- M[R2]
x3005	0	0	0	1	0	1	0	0	1	0	1	0	0	0	0	1	R2 <- R2 + 1
x3006	0	0	0	1	0	0	1	0	0	1	0	0	0	0	1	1	R1 <- R1 + R3
x3007	0	0	0	1	1	0	0	1	0	0	1	1	1	1	1	1	R4 <- R4 - 1
x3008	0	0	0	0	0	0	1	1	1	1	1	1	1	0	1	1	BRp x3004
x3009	1	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1	HALT

- Result:
- When memory contains the values to the right, result is x0024, but should be x8135

Address	Contents
x3100	x3107
x3101	x2819
x3102	x0110
x3103	x0310
x3104	x0110
x3105	x1110
x3106	x11B1
x3107	x0019
x3108	x0007
x3109	x0004
x310A	x0000
x310B	x0000
x310C	x0000
x310D	x0000
x310E	x0000
x310F	x0000
x3110	x0000
x3111	x0000
x3112	x0000
x3113	x0000

Example 2: Add a Sequence of Numbers 2

- Step 1: Examine code -- what is the algorithm?

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
x3000	0	1	0	1	0	0	1	0	0	1	1	0	0	0	0	0	`R1 <- 0`
x3001	0	1	0	1	1	0	0	1	0	0	1	0	0	0	0	0	`R4 <- 0`
x3002	0	0	0	1	1	0	0	1	0	0	1	0	1	0	1	0	`R4 <- R4 + 10`
x3003	0	0	1	0	0	1	0	0	1	1	1	1	1	1	0	0	`R2 <- M[x3100]`
x3004	0	1	1	0	0	1	1	0	1	0	0	0	0	0	0	0	`R3 <- M[R2]`
x3005	0	0	0	1	0	1	0	0	1	0	1	0	0	0	0	1	`R2 <- R2 + 1`
x3006	0	0	0	1	0	0	1	0	0	1	0	0	0	0	1	1	`R1 <- R1 + R3`
x3007	0	0	0	1	1	0	0	1	0	0	1	1	1	1	1	1	`R4 <- R4 - 1`
x3008	0	0	0	0	0	0	1	1	1	1	1	1	1	0	1	1	`BRp x3004`
x3009	1	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1	`HALT`

  - R1 (sum) initialized to zero. R4 (counter) initialized to 10
  - R2 (address) initialized to x3100
  - R3 gets value from M[R2], added to R1.
  - Increment R2 for next address
  - Decrement R4 and exit when counter is zero

Example 2 (ASM)

```
        .ORIG x3000
        AND R1, R1, #0
        AND R4, R4, #0
        ADD R4, R4, #10
        LD R2, Data
Loop    LDR R3, R2, #0
        ADD R2, R2, #1
        ADD R1, R1, R3
        ADD R4, R4, #-1
        BRp Loop
        HALT
Data .FILL x3107
        .FILL x2819
        .FILL x0110
        .FILL x0310
        .FILL x0110
        .FILL x1110
        .FILL x11B1
        .FILL x0019
        .FILL x0007
        .FILL x0004
        .FILL x0000
        .END
```

Example 2: Add a Sequence of Numbers 3

- Stepping through the code, we see the following:

PC	R1	R2	R4
x3001	0	x	x
x3002	0	x	0
x3003	0	x	#10
x3004	0	x3107	#10

- After x3004, R2 should contain x3100, the starting address of the numbers. Why does it contain x3107?
- Opcode at x3004 is **LD**. Therefore, it loads the content of memory location x3100, which is x3107.
- Instead, we need to use **LEA** to put the **address** x3100 into R2, not the **data** at M[x3100].

Example 3: Look for a Value in a Sequence 1

- Program: Look for the value 5 in the sequence of ten numbers starting at address x3100. If the value is there, put 1 in R0; else put 0 in R0.
- Code:

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
x3000	0	1	0	1	0	0	0	0	0	0	1	0	0	0	0	0	R0 <- 0
x3001	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	1	R0 <- R0 + 1
x3002	0	1	0	1	0	0	1	0	0	1	1	0	0	0	0	0	R1 <- 0
x3003	0	0	0	1	0	0	1	0	0	1	1	1	1	0	1	1	R1 <- R1 - 5
x3004	0	1	0	1	0	1	1	0	1	1	1	0	0	0	0	0	R3 <- 0
x3005	0	0	0	1	0	1	1	0	1	1	1	0	1	0	1	0	R3 <- R3 + 10
x3006	0	0	1	0	1	0	0	0	0	0	0	0	1	0	0	1	R4 <- M[x3010]
x3007	0	1	1	0	0	1	0	1	0	0	0	0	0	0	0	0	R2 <- M[R4]
x3008	0	0	0	1	0	1	0	0	1	0	0	0	0	0	0	1	R2 <- R2 + R1
x3009	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	1	BRz x300F
x300A	0	0	0	1	1	0	0	1	0	0	1	0	0	0	0	1	R4 <- R4 + 1
x300B	0	0	0	1	0	1	1	0	1	1	1	1	1	1	1	1	R3 <- R3 - 1
x300C	0	1	1	0	0	1	0	1	0	0	0	0	0	0	0	0	R2 <- M[R4]
x300D	0	0	0	0	0	0	1	1	1	1	1	1	1	0	1	0	BRp x3008
x300E	0	1	0	1	0	0	0	0	0	0	1	0	0	0	0	0	R0 <- 0
x300F	1	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1	HALT
x3010	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	0	x3100

Example 3: Look for a Value in a Sequence 2

- Result:
- When tested with data that includes one or more 5's, R0 was set to 0.

- Algorithm:
- R0 = 1 -- assuming that we will see a 5 (preset)
- R1 = −5 -- used to test if the value = 5
- R3 = 10 -- counter
- R4 = x3100 -- starting address of the sequence
- Load M[R4] into R2.
- LOOP:
  Test if R2 equals 5. **If so**, branch to x300F -- halt (with R1=1).
  (**otherwise**) Increment R4 to the next address. Decrement loop counter (R3).
  Load M[R4] into R2. Branch to back to LOOP if R3 is not zero.
- Set R0=0 -- counter hit zero before seeing any 5's.

Assembly Code

```
        AND R0, R0, #0
        ADD R0, R0, #1  ; assume 5 will be found
        AND R1, R1, #0
        ADD R1, R1, #-5
        AND R3, R3, #0
        ADD R3, R3, #10  ; counter
        LD R4, Data
        LDR R2, R4, #0
Loop ADD R2, R2, R1
        BRz   Done
        ADD R4, R4, #1
        ADD R3, R3, #-1
        LDR R2, R4, #0
        BRp Loop        ;  jump to Loop if counter is positive
        AND R0, R0, #0
Done HALT
Data  .FILL  8
        .FILL 7    ; 8 more locations initialized with numbers
```

Example 3: Look for a Value in a Sequence 3

- Learning from our earlier loop bug, we set a breakpoint at the bottom of the loop (0x300D), to see how many times it executes.
-

PC	R1	R2	R3	R4
x300D	-5	7	9	3101
x300D	-5	32	8	3102
x300D	-5	0	7	3103

- After three breakpoints, the execution does not hit this BR again, even though R3 is 7.
- *Why??* R2 shows that we did not load 5.

- Looking more closely, the simulator shows that the Z bit **is set** at the last breakpoint. **Why?** Because the LDR at x300C loaded a zero into R2. We were thinking the BR would be based on the previous ADD instruction (decrementing R3) and forgot that LDR also sets the condition codes.
- **Fix:** Move the LDR before the decrement of R3, so that the condition code is set by the relevant instruction.

Example 4: Find the First 1 in a Word 1

- Program: Load a data word from a far-away memory location (x3400). From left to right, look for the first one bit in the value, and put the bit position of that one in R1. If no 1's, make R1 = –1.
- Code:

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
x3000	0	1	0	1	0	0	1	0	0	1	1	0	0	0	0	0	R1 <- 0
x3001	0	0	0	1	0	0	1	0	0	1	1	0	1	1	1	1	R1 <- R1 + 15
x3002	1	0	1	0	0	1	0	0	0	0	0	0	0	1	1	0	R2 <- M[M[x3009]]
x3003	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	BRn x3008
x3004	0	0	0	1	0	0	1	0	0	1	1	1	1	1	1	1	R1 <- R1 - 1
x3005	0	0	0	1	0	1	0	0	1	0	0	0	0	0	1	0	R2 <- R2 + R2
x3006	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	BRn x3008
x3007	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	BRnzp x3004
x3008	1	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1	HALT
x3009	0	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	x3400

- Result: Program usually works, but sometimes fails to terminate!

university of groningen

Example 4: Find the First 1 in a Word 2

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
x3000	0	1	0	1	0	0	1	0	0	1	1	0	0	0	0	0	R1 <- 0
x3001	0	0	0	1	0	0	1	0	0	1	1	0	1	1	1	1	R1 <- R1 + 15
x3002	1	0	1	0	0	1	0	0	0	0	0	0	0	1	1	0	R2 <- M[M[x3009]]
x3003	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	BRn x3008
x3004	0	0	0	1	0	0	1	0	0	1	1	1	1	1	1	1	R1 <- R1 - 1
x3005	0	0	0	1	0	1	0	0	1	0	0	0	0	0	1	0	R2 <- R2 + R2
x3006	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	BRn x3008
x3007	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	BRnzp x3004
x3008	1	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1	HALT
x3009	0	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	x3400

- ## Algorithm:
- R1 = 15, preset to leftmost bit position
- R2 = data value (memory address is in x3009, using LDI to load M[x3400])
- If value is negative, then 15 is the first 1 --> branch to HALT.
- LOOP:
  Decrement R1 (next bit position). Shift value (R2) to the left.
  If negative, then a 1 was shifted into position 15 --> exit loop.

Assembly Code

```
        .ORIG x3000
        AND R1, R1, #0
        ADD R1, R1, #15
        LDI R2, Num
        BRn Done
Loop    ADD R1, R1, #-1
        ADD R2, R2, R2
        BRn Done
        BRnzp Loop
Done    HALT
Num     .FILL x3400
```

Example 4: Find the First 1 in a Word 3

› Set breakpoint at bottom of loop, to see how many times it's executed.

PC	R1
x3007	14
x3007	13
x3007	12
x3007	11
x3007	10
x3007	9
x3007	8
x3007	7
x3007	6
x3007	5
x3007	4
x3007	3
x3007	2
x3007	1
x3007	0
x3007	−1
x3007	−2
x3007	−3
x3007	−4

› Loop should execute at most 16 times. Why does it keep going?

› Problem: A value of zero is loaded into R2. Therefore, a 1 is never shifted into the sign bit. Therefore, the loop is never exited -- INFINITE LOOP.

› **Fix:** Add a check for zero when value is first loaded. Exit and set R1 = −1.

› NOTE: This only fails when value is zero. If you didn't happen to test that case, then your customer will find the bug instead!

Conclusions

› Use breakpoints to observe the behavior of the program, paying special attention to register values that affect control flow.

› Single-step through code as needed, to understand why register are getting their value. (Wrong opcode, wrong operand, ...?)

› Test with a variety of input data.

- Especially look for corner cases that may behave differently, such as negative numbers, or zero.

Don't assume that comments are correct!

- Comments express what the programmer intended, but only the instructions determine how the program actually behaves.

university of groningen

Questions?