Programming in LC-3 machine language

Lecture Topics

Programming using systematic decomposition

Debugging

LC-3 data path review

Lecture materials

Textbook Ch. 6

Homework

HW3 due Wednesday February 23 at 5pm in the ECE 190 drop-off box

Machine problem

MP2 due March 2, 2011 at 5pm submitted electronically.

Announcements

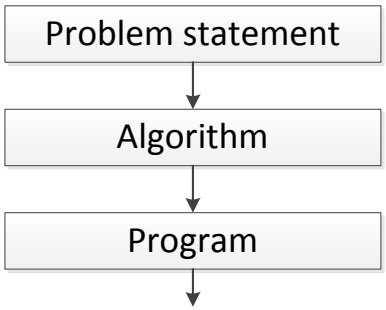
Stats for the exam

	Monday	Tuesday	Wednesday
# of students that took the exam	97	96	90
Average	37.7	38.9	35.2
Standard deviation	14.0	16.9	14.8
# of zeros on programming part	23	26	30
Highest score	60	60	59
Lowest score	6	7	2

Programming using systematic decomposition

Systematic decomposition

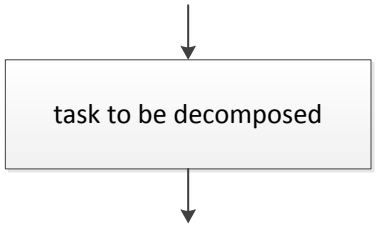
- In order for electrons to solve a problem for us, we need to go through several levels of transformation to get
  - from the natural language in which the problem statement is initially expressed
  - to the level at which electrons can be manipulated to do the work for us



- Problem statement can be imprecise, thus, we first translate it into a precise algorithm which should have the following 3 properties:
  - finiteness (it terminates)
  - definiteness (each step is precisely stated)
  - effective computability (each step can be carried out by the computer)
- to start with the problem statement and end up with a working program, we will apply a process referred to as *systematic decomposition* or *stepwise refinement*
  - complex tasks are systematically broken down into simpler, smaller tasks such that the collection of these simpler tasks, or units of work, will accomplish the same as the original task
  - the decomposition continues until each simpler task can be implemented as just a few instructions in the programming language we use

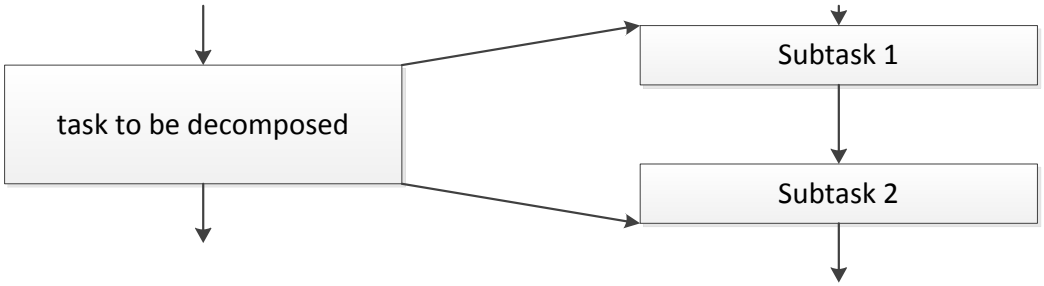
Three constructs

- We want to replace a large unit of work with a set of a few smaller units of work.



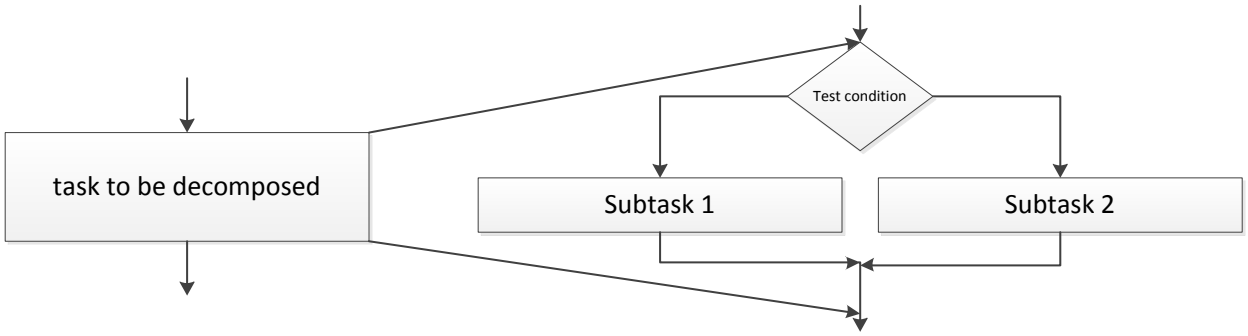
- This can be done using one of the 3 basic constructs: sequential, conditional, and iterative
- **Sequential construct**
  - Is used when we can decompose a given task into two smaller sub-tasks, such that one

of them has to be fully executed before we can execute the other:



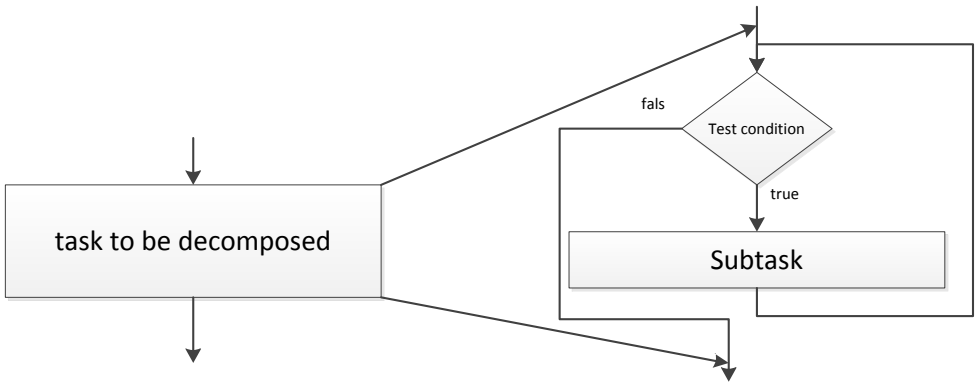
- Once subtask 1 is executed, we never go back to it, we continue with subtask 2

- **conditional construct**
  - Is used when the task consists of some subtasks such that only one of them needs to be executed, but not all, depending on some condition:



  - If the test condition is true, we need to carry out one subtask, otherwise we carry out the other task; but we newer carry out both tasks
  - Once one of these subtasks is executed, we never go back

- **Iterative construct**
  - Is used if one of subtasks needs to be re-done several times, but only as long as some condition is met

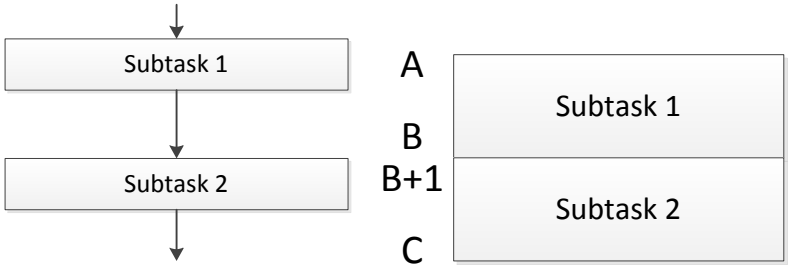


- Each time we finish executing the subtask, we go back and reexamine the condition
- The moment the condition is not met, the program proceeds onwards

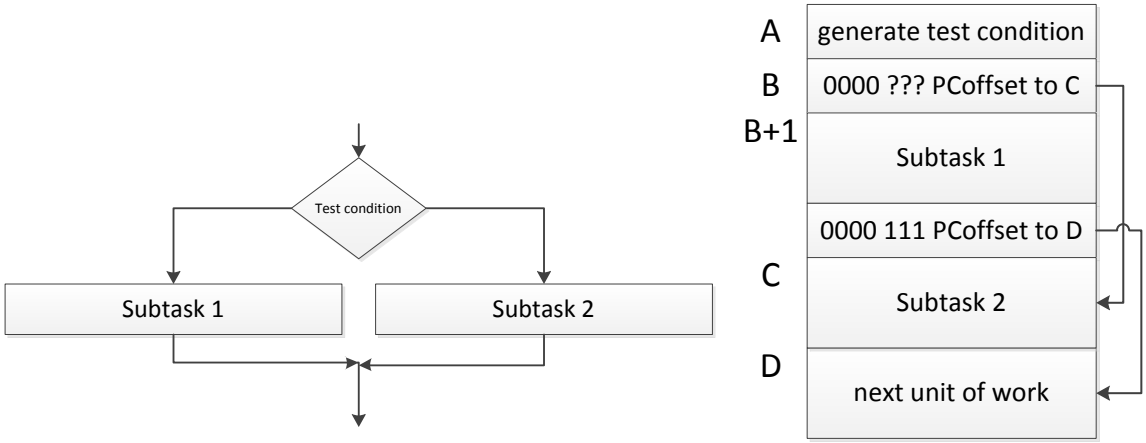
- Few words about flowchart notation
  - Flowchart is built form boxes with one input and one output
  - Each such box "holds" a task
    - Rectangle shape is used to indicate a work task
    - Rhomb shape is used to indicate a decision task
    - Ellipse shape is used to indicate beginning and end of the task sequence for the entire flowchart

LC-3 instructions to implement the constructs

- **Sequential construct**
  - Subtask 1 starts at memory address A and ends at memory address B
  - Subtask 2 then starts at memory address B+1

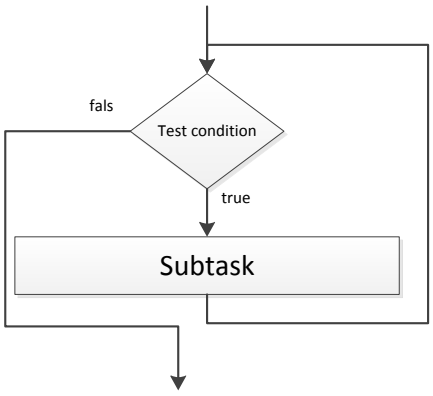


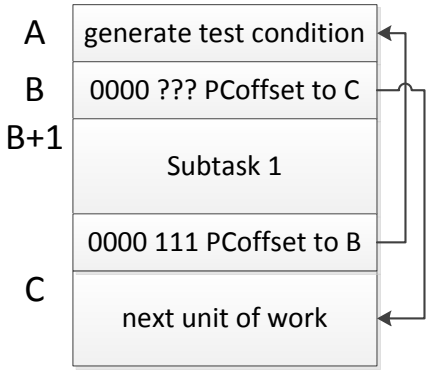
- **Conditional construct**
  - Create code that converts decision condition into one of the condition codes N,Z,P
  - Use conditional BR instruction to transfer control to the proper subtask
  - Use unconditional BR to transfer control to the next task after first subtask is done



- **Iterative construct**
  - Create code that converts decision condition into one of the condition codes N,Z,P
  - Use conditional BR instruction to branch if the condition generated is *false*

- Use unconditional BR to transfer control back to the condition generating code after the subtask id done





Example: character counter

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"

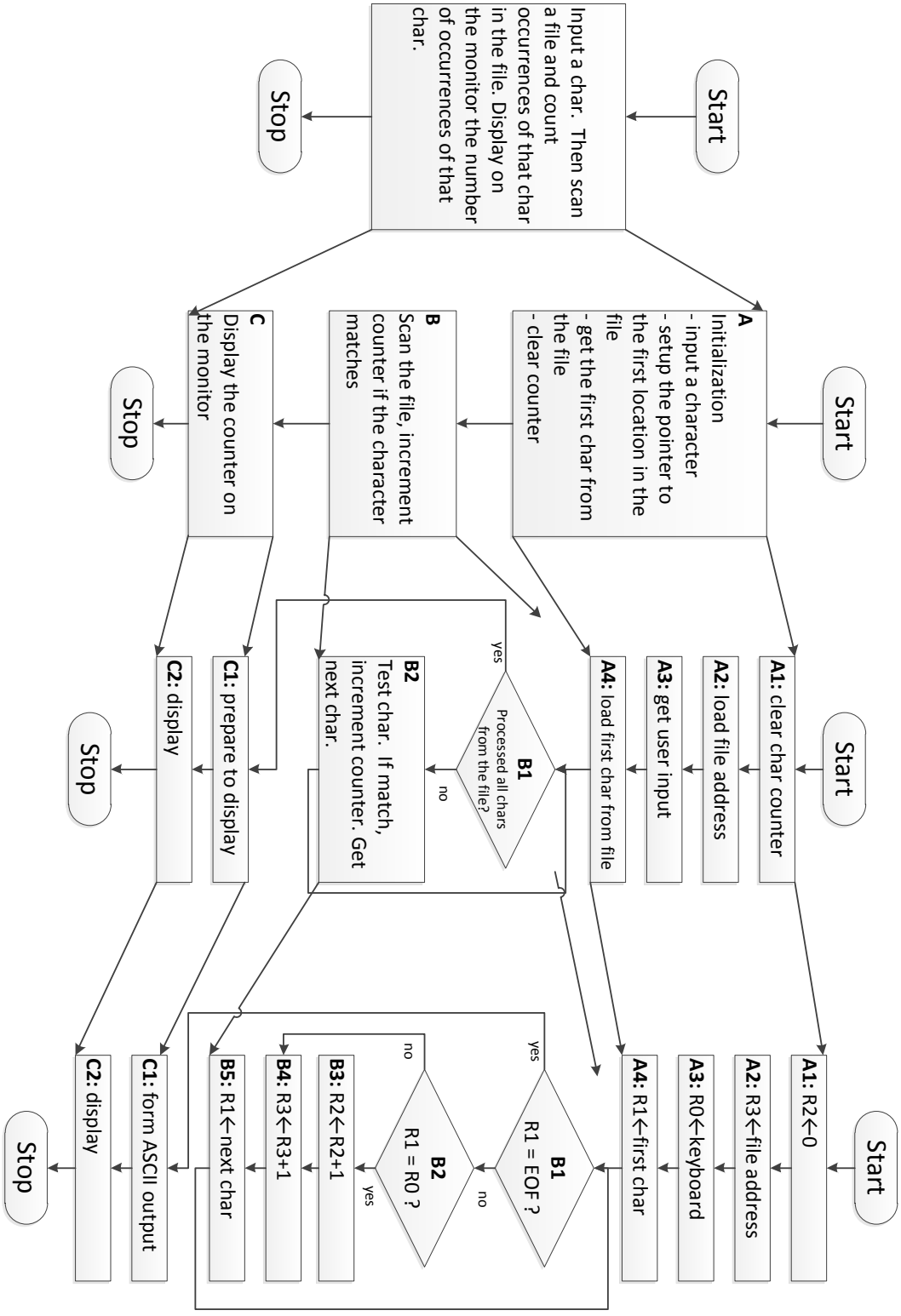
- Step 1: convert this problem statement into an algorithm
- Step 2: convert the algorithm into a program written in LC-3 machine language

Step 1: from problem statement to algorithm/flowchart

- Problem statement is not precise and incomplete
  - What is a "file"?
  - Where is it?
  - How big is it?
  - How should the final count be printed?
- How do we resolve these questions?
  - Ask for clarification from the person who gave you this problem statement
  - Make a decision yourself and document it
- For the given problem statement, we will make the following assumptions:
  - "file" is a sequential record in memory, starting at some known/given address and ending with some special character, say, EOT, whose ASCII value is x4
    - Thus, we will use one memory location, say, x3012, to hold the address of where the file begins
  - The final count is to be printed on the screen as a number
  - Restriction: no more than 9 identical characters in the file (later we will see why)
- We will use the following resisters
  - R0 - character to count
  - R1 - character from the file
  - R2 - occurrence counter

- R3 - location of the next character to read from file

- Systematic decomposition is recursively applied until we arrive at a very fine set of well-defined steps each of which can be translated into just a few instructions in the programming language:



Step 2: from algorithm/flowchart to pseudo code

x3000	R2←0	AND	
x3001	R3←MEM[x3011]	LD	(figure out PC offset later (x10))
x3002	GETC	TRAP x23	
x3003	R1←MEM[R3]	LDR	
x3004	R4←R1-4	ADD	
x3005	BRz, _____		(figure out PC offset later (x8))
x3006	R1←NOT(R1)		
x3007	R1←R1+1		
x3008	R1←R1+R0		
x3009	BRnp, ____		(figure out offset PC later (x1))
x300A	R2←R2+1		
x300B	R3←R3+1		
x300C	R1←MEM[R3]	LDR	
x300D	BRnzp, ____		(figure out offset later (-10))
x300E	R0←M[x3013]	LD	(figure out offset later (x4))
x300F	R0←R0+R2		
x3010	PUTC	TRAP x21	
x3011	HALT	TRAP x25	
x3012	x3016	(starting address of the file)	
x3014	x30	(ASCII offset value for printing a digit)	
x3016	x74	't'	
x3017	x65	'e'	
x3018	x73	's'	
x3019	x74	't'	
x301A	x4	end-of-file symbol	

Step 3: from pseudo code to program in machine language

```
; Program to count occurrences of a character in a file
;
; limitations:
; - program only works if no more than 9 occurrences are found
;
; inputs:
; - character to be input from the keyboard
;
; outputs:
; - result to be displayed on the monitor
;
; register usage:
; R0 - character to count
; R1 - character from the file
; R2 - character occurrence counter
; R3 - location of the next character to read from file
;
; memory usage:
; x3012 - location of the file
```

```
; x3013 - stores x30 value used to convert count to an ASCII char for output
; - actual file, located starting from the address specified in x3012
; - file must end with value x0004 (end-of-text)
;
0011 0000 0000 0000; starting address of the program (x3000)
;
; Initialization
0101 010 010 1 00000  ; (AND R2, R2, #0) clear character counter
0010 011 000010000    ; (LD R3, x10) load starting address of the file
1111 0000 00100011    ; (GETC) get a character from the keyboard
0110 001 011 000000   ; (LDR R1, R3, #0) load next character from the file
;
; Test character for end-of-file
0001 100 001 1 11100  ; (ADD R4, R1, #-4) check if this is end of file
0000 010 000001000    ; (BRz, x8) if so, skip to the output part
;
; Test character for match
1001 001 001 111111   ; (NOT R1, R1) otherwise, check if the character
0001 001 001 1 00001  ; (ADD R1, R1, #1) from the file is the same as the
0001 001 001 0 00 000 ; (ADD R1, R1, R0) character entered from the keyboard
0000 101 000000001    ; (BRnp, x1) if not, skip just one line
0001 010 010 1 00001  ; (ADD R2, R2, #1) if yes, increment the counter
;
; Read next character from file
0001 011 011 1 00001  ; (ADD R3, R3, #1) move to the next character in file
0110 001 011 000000   ; (LDR R1, R3, #0) read it
0000 111 111110110    ; (BRnzp x-A) and repeat the test
;
; Output results and stop
0010 000 000000100    ; (LD R0, #4) load ASCII offset
0001 000 000 0 00 010 ; (ADD R0, R0, R2) prepare counter for output
1111 0000 00100001    ; (PUTC) output counter value
1111 0000 00100101    ; (HALT) halt the execution of the program
;
; Storage for file address and ASCII offset
0011 0000 0001 0110   ; start of file memory address - x3016
0000 0000 0011 0000   ; ASCII offset value - x30
```

```
; file to count occurrences of a character
0011 0000 0001 0110   ; start of file memory address - x3016
; actual file to process, one character per memory location
0000 0000 0111 0100   ; x74 t
0000 0000 0110 0101   ; x65 e
0000 0000 0111 0011   ; x73 s
0000 0000 0111 0100   ; x74 t
0000 0000 0000 0100   ; x4 - end-of-file symbol
```

Debugging

- We have written our program and it does not work! Now what?
- The solution is to trace our program in an attempt to identify where the problem is
  - Examine the sequence of instructions being executed
  - Examine track of results being produced
  - Compare results from each instruction to the expected result

- Errors present in programs are referred to as *bugs* and the process of getting rid of them is referred to as *debugging*.
- Types of errors
    - Syntax error: typing error that results in an illegal operation
    - Logic error: program is legal, but wrong, and so the results do not match the problem statement
    - Data error: input data is different than what you expected
- Tracing the program
    - Execute the program one piece (one instruction) at a time, examine registers and memory to see results at each step
        - Single-stepping: execute one instruction at a time
        - Breakpoints: direct the simulator/debugger to stop executing when it reaches a specific program location
        - Watchpoints: direct the simulator to stop when a register or memory locations changes, or becomes equal to a specified value

LC-3 data path review

- The LC-3 data path consist of all the components that process the information during an instruction cycle
    - The functional unit that operates on the information
    - The registers that store the information
    - Buses and wires that carry information from one point to another
- **Basic components of the data path:**
- The global bus
    - LC-3 global bus consists of 16 wires and associated electronics
    - It allows one structure to transfer up to 16 bits of information to another structure by making the necessary electronic connections on the bus
    - Exactly one value can be translated on the bus at a time
    - Each structure that supplies values to the bus connects to it via a *tri-state device* that allows the computer's control logic to enable exactly one supplier of information to the bus at a time
    - The structure wishing to obtain information from the bus can do so by asserting (setting to 1) its LD.x (load enable) signal.
- Memory
    - Memory in LC-3 is accessed by loading the memory address value into MAR
    - The result of memory read is stored in MDR
    - The computer's control logic supplies the necessary control signals to enable the read/write of memory
- The ALU and the register file
    - The ALU is the processing element
        - Has two inputs and one output

        - Inputs come directly from the register file
        - One of the inputs can also be supplied directly from the IR, controlled by the SR2MUX
        - Output goes to the bus
            - It then is stores in the register file and
            - Processed by some additional circuitry to generate the condition codes N,Z,P
    - The register file consists of 8 16-bit registers
        - It can supply up to two values via its two output ports
        - It can store one value, coming from the bus
        - Read/write access to the register file is controlled by the 3-bit resister signals, DR, SR1, SR2
- The PC and PCMUX
    - The PC supplies via the global bus to the MAR the address of the instruction to be fetched at the start of the instruction cycle. The PC itself is supplied via the three-to-one PCMUX, depending on the instruction being executed
    - During the fetch cycle, the PC is incremented by 1 and written to the PC register
    - If the executed instruction is a control instruction, then the relevant source of the PCMUX depends on the type of the control instruction and is computed using a special ADD unit
- The MARMUX
    - Controls which of two sources will supply memory address to MAR
    - It is either the value coming from the instruction register needed to implement TRAP instruction, or the value computed based on the PC or a value stored in one of the general-purpose registers
- LC-3 instruction cycle revisited
    - FETC
        - IR contains the fetched instruction
        - PC is incremented by 1
    - DECODE
        - Instruction is decoded resulting in control logic providing various control signals to the computer
    - EVALUATE ADDRESS
        - Address of memory to be accessed during the execution of the instruction is computed
        - Memory access instructions require this phase
    - OPERAND FETCH
        - The data from memory is loaded into MDR
    - EXECUTE
        - ALU is directed to perform the operation
        - Memory access instructions do not have this phase

- STORE RESULTS
  - Results are stored to memory or registers, depending on the instruction