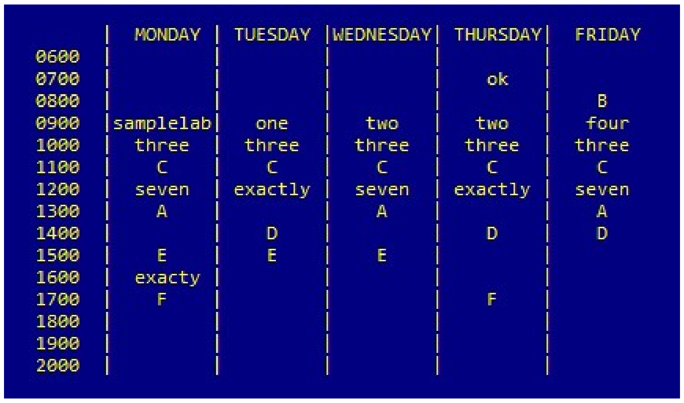
ECE220: Computer Systems & Programming **Fall 2020 ZJUI**
Machine Problem 3 **due: Saturday 17 October, 11:59:59 p.m.**

Solving a Scheduling Problem with DFS

Your task this week is to write an LC-3 program that attempts to find a compatible combination of times at which events can be inserted into an existing weekly schedule. Given a list of extra events, each of which can occur at one or more hours, your program must use a stack to perform a depth-first search (DFS) of possible time combinations until a compatible combination is discovered (or until all possible combinations are eliminated as incompatible). Your program must extend your solution for MP2, and requires roughly another 130 lines of LC-3 assembly code.



```
     |  MONDAY | TUESDAY |WEDNESDAY| THURSDAY|  FRIDAY
0600 |         |         |         |         |
0700 |         |         |         |    ok   |
0800 |         |         |         |         |    B
0900 |samplelab|   one   |   two   |   two   |  four
1000 |  three  |  three  |  three  |  three  |  three
1100 |    C    |    C    |    C    |    C    |    C
1200 |  seven  | exactly |  seven  | exactly |  seven
1300 |    A    |         |    A    |         |    A
1400 |         |    D    |         |    D    |    D
1500 |    E    |    E    |    E    |         |
1600 |  exacty |         |         |         |
1700 |    F    |         |         |    F    |
1800 |         |         |         |         |
1900 |         |         |         |         |
2000 |         |         |         |         |
```

The objective for this week is to give you additional experience with understanding and manipulating arrays of data in memory, and particularly in developing and using a stack.

The Task

In this program, you must try to fit a set of extra events into an existing schedule. Each extra event can be inserted at some number of possible hours, and your program must try to find a way to insert all of the events (each at one of the times allowed for that event) such that no events conflict with each other or with the predefined schedule (the output of your MP2 code's translation step). If no compatible combination of times is possible for the extra events, your program must print an error message and terminate without printing the schedule. If a compatible combination is found, your program should print the final schedule, with all extra events inserted at appropriate times.

Start by making a copy of your MP2 program. Your MP3 code must be written as a single subroutine called **`MP3`**. You should insert a call to the **`MP3`** subroutine between the schedule translation code and the schedule printing code from MP2. The schedule should be printed if and only if the value returned from the **`MP3`** subroutine indicates success (see Specifics).

The extra event list starts at address x4800 in LC-3 memory. Each extra event consists of three fields. The first field is a bit vector of days for the event: Monday is bit 4 (value 16), Tuesday is bit 3 (value 8), and so forth, through Friday (bit 0, value 1). The days on which the event occurs are OR'd together to produce the bit vector. Other bits in this bit vector will always be 0. The second field is a string pointer—the address of a string describing the event. The third field is a bit vector of hour slots for the event: 06:00 is bit 0 (value 1), 07:00 is bit 1 (value 2), and so forth, through 20:00 (bit 14, value x4000). The value of bit 15 is unspecified, and may be either 0 or 1. The hours in which the event can appear in the schedule are OR'd together to produce the bit vector.

The extra event list ends with a -1 bit vector of days—no string pointer nor slot bit vector is provided.

Notice that there are two differences between the events provided in MP2 and the extra events. First, the string is NOT part of the event, but has been replaced with a pointer to a string. Having written MP2, you know how to handle strings already.

Also, the single hour slot is now a bit vector of possible hours. In this MP, you must determine which combination of hours (chosen from the possible hours in each extra event's bit vector) for each event produces a compatible schedule, in which no two events conflict with one another. Your code must prefer early hours to late hours, so you should start by checking bit 0, then bit 1, then bit 2, and so forth. (It's much more difficult to test in other orders with LC-3, and allowing other orders makes grading more difficult.)

To explore the space of possible combinations, your program must make use of a stack and perform a depth-first search (DFS). Use x8000 as the start of your stack, as checks for an empty stack are then slightly easier to perform. In the path-finding problem in class, we used a queue and performed a breadth-first search (BFS). In that case, nodes in the graph were explored in the order that they were added to the queue: first-in, first-out (FIFO). In contrast, as your program adds new events to the stack, it will then explore those events before expanding any previous events. In other words, your code works in a last-in, first-out (LIFO) order.

At first, the stack will be empty. Your program should take the first event from the extra event list and push it onto the stack using a structure that you design, then choose an hour from among the possible hours for that event, and try to insert the event into the schedule. If successful, your code should continue with the next event. If at any point, your program manages to find hours for all events in the extra event list, a compatible schedule has been found, and your code can proceed with printing (the stack can be discarded—no need to clean it up).

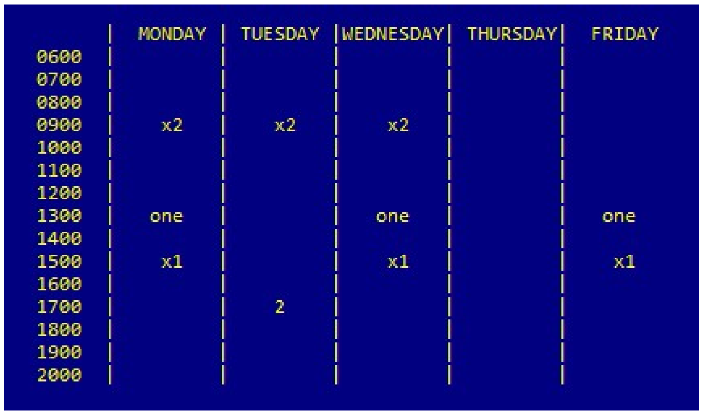
If your program finds that an event on top of the stack has conflicts for every possible hour, that event must be popped from the stack. After popping an event structure, your program should then remove the event on top of the stack from the hour at which it was previously inserted into the schedule and try to find an alternative possible hour for that event (remember that the impossible event has been popped). If your program finds the stack empty when it tries to pop an incompatible event, no compatible combination of the extra events exists, and your program must stop exploring and print an error message, then terminate.

You must decide what information to include in the event structures your stack. Whatever you decide, be sure to add comments describing how events appear on your stack in your code. You may want to keep a copy of the design nearby as you write your code (on paper, for example).

You may want to consider the following questions as you design the structure of events on your stack. Does the "first" (smallest) 1 bit of an event on the stack represent the hour slot currently occupied by the event, or the next slot to be tested if remaining events cannot fit into the schedule? Also, does your event structure include the current hour slot for the event (call it S) and/or the current bit for that slot (1<<S)?

Here is an example (provided to you as **`short.asm`** and inserted into the **`simple.asm`** schedule from MP2 in the figure to the right) of how an extra event list might appear in memory and how the schedule produced by your program should appear after finding and printing a compatible schedule. The data shown list only two extra events starting at x4800 and x4803. The value -1 at x4806 ends the list.

address	contents	Meaning
`x4800`	**`x0015`**	M (16) | W (4) | F (1)
`x4801`	**`x4807`**	points to "x1"
`x4802`	**`x0288`**	09:00 | 13:00 | 15:00
`x4803`	**`x001C`**	M (16) | T (8) | W (4)
`x4804`	**`x480A`**	points to "x2"
`x4805`	**`x0888`**	09:00 | 13:00 | 17:00
`x4806`	**`xFFFF`**	-1 (ends list)
`x4807`	**`x0078`**	'x'
`x4808`	**`x0031`**	'1'
`x4809`	**`x0000`**	NUL
`x480A`	**`x0078`**	'x'
`x480B`	**`x0032`**	'2'
`x480C`	**`x0000`**	NUL



```
     |  MONDAY  | TUESDAY  |WEDNESDAY| THURSDAY|  FRIDAY
0600 |          |          |         |         |
0700 |          |          |         |         |
0800 |          |          |         |         |
0900 |    x2    |    x2    |    x2   |         |
1000 |          |          |         |         |
1100 |          |          |         |         |
1200 |          |          |         |         |
1300 |   one    |          |   one   |         |   one
1400 |          |          |         |         |
1500 |    x1    |          |    x1   |         |    x1
1600 |          |          |         |         |
1700 |          |    2     |         |         |
1800 |          |          |         |         |
1900 |          |          |         |         |
2000 |          |          |         |         |
```

Specifics

- Your code must be written in LC-3 assembly language and must be contained in a single file called **`mp3.asm`** in the **`mp/mp3`** subdirectory of your repository. We **will not grade** any other files.
- Your program must start at x3000, and must call a subroutine called MP3 between translating and printing the schedule. The subroutine takes no inputs (it uses the schedule and extra event lists from memory). As an output, the subroutine must return R0=0 for failure and R0=1 for success. For simplicity, all registers are caller-saved. Side effects are specified below. If the MP3 subroutine returns success, your program must print the schedule (as produced by the subroutine). Your subroutine must be able to execute without execution of your main program (for testing).
- Your MP3 subroutine must attempt to find a compatible combination of times in the schedule at x3800 in memory for all events in the extra event list starting at x4800 in memory.
    - Each event in the extra event list consists of a bit vector of days of the week (Monday = 16, Tuesday = 8, Wednesday = 4, Thursday = 2, Friday = 1), a pointer to a string, and a bit vector of possible hours (06:00 = 1, … , 20:00 = x4000).
    - A day bit vector of -1 ends the event list (the final entry is not considered an event, and no string nor bit vector possible hours are included after the -1).
    - You may assume that all extra event string pointers point to valid and unique ASCII strings. The assumption of uniqueness makes it slightly easier to remove conflicts that are partially written into the schedule, as otherwise an event's name may match that of another event. Be sure that you do not create tests that break this assumption (and thus spend time 'debugging' your code for no reason).
    - You may assume that all bit vectors of days are valid combinations representing (possibly empty) subsets of weekdays (Monday through Friday).

    - You may not make any assumptions about bit vectors of hours; these can be anything. In particular, your MP3 subroutine must ignore the meaningless bit (bit 15).
    - Your program must make use of a stack with base equal to x8000 and a DFS to explore the possible combinations of hours. Earlier hours must be considered first. For example, if an event can occur at 07:00 or at 08:00, and both hours are compatible with all other events, 07:00 must be chosen for that event.
    - The DFS must explore events in the order in which they appear in the event list. The first extra event must be added to the stack first, then the second, and so forth.
    - If a compatible combination of hours for events in the extra event list is found, your subroutine must return R0=1, which should cause your program to print the schedule with the extra events included at the chosen time for each event on the days on which the event occurs. Your subroutine must leave the schedule filled with the correct values so that the MP2 PRINT_SCHEDULE subroutine prints the correct resulting schedule.
    - If no compatible combination exists for all events in the extra event list, your subroutine must print the message, "Could not fit all events into schedule.\n" and return R0=0, which should cause your program to terminate without printing a schedule. Note that, in this case, your MP3 subroutine may leave the schedule in an unusable and meaningless state—there is no need to clean it up.
- Your code must not access the contents of memory locations other than those required for this MP (the event list, schedule, extra event list, and stack) or declared within your program (using .FILL or .BLKW), and must not modify any location in the event list nor in the extra event list.
- Your code must be well-commented, and must include a table describing how registers are used within each part of the code. Follow the style of examples provided to you in class and in the textbook.
- Do not leave any additional code in your program when you submit it for grading.

Testing

We strongly suggest that you make use of pencil and paper when developing your code. You are likely to run short on registers and need to use some LC-3 registers for multiple purposes. Be sure to keep track of the meaning of each register for each part of the code (write the meanings into comments!).

Be sure that you remove any conflicting events correctly. In MP2, when you found a conflict, you could just leave any instances of the conflicting event (on non-conflicting days earlier in the week) in the schedule. For this MP, you must remove such instances. Similarly, when you remove an event from the stack, your program must remove all instances of that event from the hour slot in which it was previously scheduled.

You may want to work through a small example on paper and think about when and why your code should push a new event structure onto the stack as well as when and why your code should pop an event structure from the stack.

Several tests have been provided to you along with samples of original schedule data and extra events. The tests provided are likely to help you find bugs, but remember that **testing your program is your responsibility**. Duplicating the results provided with the given tests does not guarantee that your program contains no errors.

Grading Rubric

Functionality (70%)

- 15% - program handles extra event lists with no schedule conflicts (in which all extra events can occur at their earliest possible hour)
- 20% - program handles extra event lists with no backtracking (in which all extra events can occur at the earlier possible hour that does not conflict with the starting schedule, nor with earlier extra events)
- 30% - program handles backtracking correctly (extra events lists that require popping events from stack to explore alternative hours for events already on the stack)
- 5% - program correctly identifies cases in which no compatible combination of extra events exists, and outputs the correct message in such cases.

Comments, Clarity, and Write-up (30%)

- 5% - a paragraph appears at the top of the program explaining what it does (this is given to you; you just need to document your work)
- 5% - event structure used on the stack is defined clearly in the code
- 10% - register use is well-documented in the code (register tables and/or additional comments as necessary)
- 10% - code is clear and well-commented

Note that some categories in the rubric may depend on other categories and/or criteria, and that your subroutine must implement most but not all of the functionality. If your division of work does not match the specification, you will lose points. Also, for example, if your code does not assemble, you will receive no functionality points. Similarly, if MP2 code is not working, you are likely to lose lots of points. If you violate constraints (for example, changing the extra event list data, or not using a stack), the penalty will be substantial (half or all of the points). Finally, you will also be penalized heavily if your code executes data or modifies itself (do not write self-modifying code).

Sharing MP2 Solutions

To help you in testing your code, you may make use of another student's MP2 solution as part of your MP3, provided that you **strictly obey the following**:

- You may not obtain another student's MP2 solution until **after class on Tuesday 13 October**. Violation of this rule is an academic integrity violation, will result in BOTH students receiving 0 for MP2, and may have additional consequences.
- You must clearly mark the other student's code in your own MP2, and must include the students name in comments indicating that you are using their code. **Failure to mark their code appropriately will result in your receiving a 0 for MP3.**
- As you should know already, you may not share any additional code beyond the solution to MP2.

Please also note that if the other student's code has bugs that lead to your introducing bugs into your MP3 code, you may lose points as a result. We in no way guarantee the accuracy of any student's MP2 solution.