# Midterm Solutions

CSEE W3827 - Fundamentals of Computer Systems — Mar. 27, 2018
Spring 2017 — Prof. Rubenstein

This midterm contains 9 pages: a question 0 and 3 other questions, and totals 110 points. To get full credit you must answer all questions. **NO BOOKS, NOTES OR ELECTRONIC DEVICES PERMITTED!** The time allowed is 75 minutes.

Please answer all questions in the blue book, using a **separate** page for each question. **Show all work! We are not just looking for the right answer, but also how you reached the right answer. Right answers without work get no credit!!!**

YOU MAY KEEP YOUR COPY OF THE QUESTIONS.
Some advice:

- Be sure to leave some time to work on each problem. The right answer to each problem does not require a very long answer.
- Be sure to start every problem. And take some time to think about how to set the problem up before you start writing.

0. (5 pts) Do the following in the blue book:
   (a) CLEARLY write your name **and UNI** on the front cover.
   (b) start each numbered question's solution on a new page. So question 2 should start on a new page, question 3 on a new page, etc.
   (c) label solutions (e.g., 2a, 2b, 2c or 2a, b, c)

1. (35 pts) Professor Rubenstein wants to start a gaming company called FunForTwoHands: automation for games involving 2 people and fun things they can do together, with one hand each.

   Are you thinking what I'm thinking? Yes - the initial product will implement the 2-player hand game of Rock-Paper-Scissors, helping to identify the winner of each round.

   The game of Rock-Paper-Scissors is played by two players (player 0 and player 1), where each round, the two players each simultaneously select one of the three options: Rock, Paper or Scissors. If both players choose the same object, the players tie. Otherwise, the winner is chosen as follows:

   - Paper covers Rock (player choosing paper beats player choosing rock)
   - Scissors cuts paper (player choosing scissors beats player choosing paper)
   - Rock smashes scissors (player choosing rock beats player choosing scissors)

   To automate the process, player 0's choice is described by the two-bit input $AB$, and player 1's choice by $CD$, where the 2-bit input indicates the following choice:

| 2-bit input | | choice indicated |
|---|---|---|
| 0 | 0 | Rock |
| 0 | 1 | Paper |
| 1 | 0 | Scissors |
| 1 | 1 | (Unused) |

   Given inputs $AB$ and $CD$ that select one of the three 2-bit options above (i.e., you **should** assume that $AB = 11$ or $CD = 11$ will never be provided as an input), 2 outputs can be generated:

   - $T$: outputs 1 when there is a tie, 0 otherwise.
   - $W$: identifies the winner when the selections do not result in a tie (i.e., equals 0 when player 0 wins, 1 when player 1 wins).

   For instance, if $ABCD = 0001$, then $T = 0$ and $W = 1$ since player 0 chose rock, and player 1 chose paper, and paper covers rock, hence player 1 wins.

   Note that in the event of a tie, the $W$ output is meaningless, since there is no winner.

   As the circuit designer for Prof. Rubenstein, you need to provide **simplified SoP algebraic equations** for $T$ and $W$ as a function of the 4 inputs, $A, B, C, D$. Make sure your equations will produce circuits that, considered individually, are as simple as possible (in SoP form).

**Answer:** We "don't care" what $T$ is when either $AB = 11$ or $CD = 11$. Otherwise, $T$ only equals 1 when $AB = CD$. We "don't care" what $W$ equals when $AB = 11$ or $CD = 11$. We also "don't care" what it equals when there is a tie (i.e., when $T = 1$). Otherwise, $W$'s solution follows game rules.

| $AB$ | $CD$ | $T$ | $W$ |
|---|---|---|---|
| 00 | 00 | 1 | X |
| 00 | 01 | 0 | 1 |
| 00 | 10 | 0 | 0 |
| 00 | 11 | X | X |
| 01 | 00 | 0 | 0 |
| 01 | 01 | 1 | X |
| 01 | 10 | 0 | 1 |
| 01 | 11 | X | X |
| 10 | 00 | 0 | 1 |
| 10 | 01 | 0 | 0 |
| 10 | 10 | 1 | X |
| 10 | 11 | X | X |
| 11 | 00 | X | X |
| 11 | 01 | X | X |
| 11 | 10 | X | X |
| 11 | 11 | X | X |

$T$:

| $A$ \ $D$ | | | | |
|---|---|---|---|---|
| | 1 | 0 | X | 0 |
| | 0 | 1 | X | 0 |
| | X | X | X | X |
| | 0 | 0 | X | 1 |

B (rows), $C$ (columns)

$T = \bar{A}\bar{B}\bar{C}\bar{D} + AC + BD$

$W$:

| $A$ \ $D$ | | | | |
|---|---|---|---|---|
| | X | 1 | X | 0 |
| | 0 | X | X | 1 |
| | X | X | X | X |
| | 1 | 0 | X | X |

B (rows), $C$ (columns)

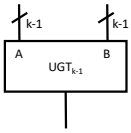
$W = A\bar{D} + BC + \bar{A}D = A \oplus D + BC$

Note: you did not need to use the XOR.

2. (35 pts) A $k$-UnsignedGreaterThan circuit ($\text{UGT}_k$ for short) takes two $k$-bit inputs, $B = B_{k-1}B_{k-2}\cdots B_1B_0$ and $A = A_{k-1}A_{k-2}\cdots A_1A_0$, and returns 1 (TRUE) when $B > A$ where both $B$ and $A$ represent *unsigned $k$-bit binary values*. The circuit can be built recursively via a combination of circuitry and $\text{UGT}_{k-1}$ circuits.

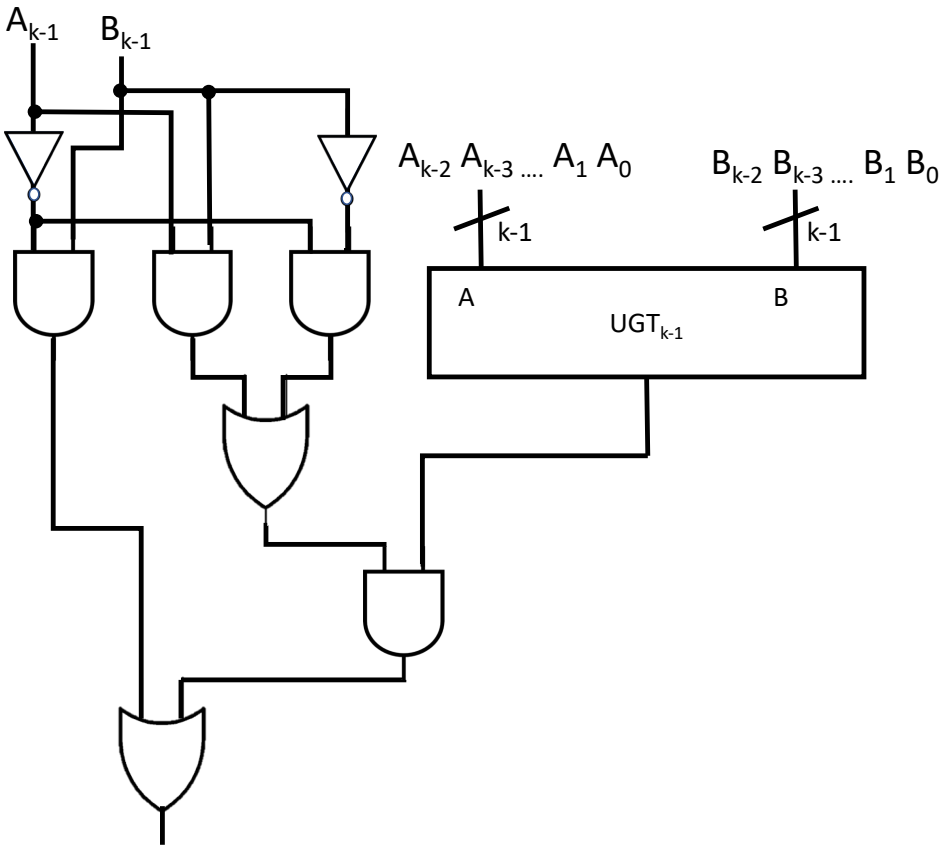
   (a) (5 pts) Show a simplified design for the base case: a $\text{UGT}_1$ circuit, built using only AND, OR, and NOT gates, that returns TRUE when $B_0 > A_0$ (note, we are requiring strictly greater here).

   **Answer:** The circuit must return 1 when and only when $B_0 = 1$ and $A_0 = 0$, hence the circuit is simply $B_0\overline{A_0}$.




   (b) (8 pts) Now show how to build a $\text{UGT}_k$ circuit for $k > 1$ recursively by using one or more $\text{UGT}_{k-1}$ circuits and additional basic gates (i.e., AND, OR, NOT gates). Make sure your design clearly indicates which inputs feed in where. You should represent a $\text{UGT}_{k-1}$ circuit as shown in the figure above when using it within your $\text{UGT}_k$ circuit.

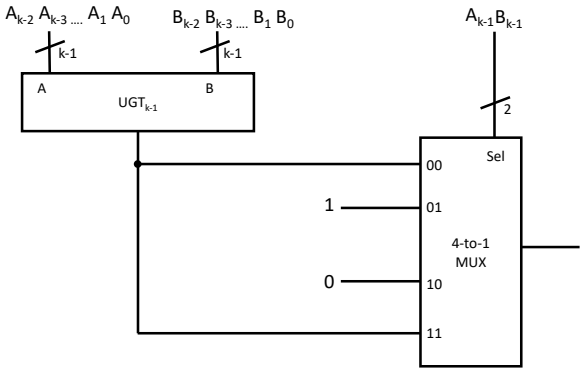
   **Answer:** When $B_{k-1} = 1$ and $A_{k-1} = 0$, then $B$ will be larger, so the circuit should return 1. When $B_{k-1} = 0$ and $A_{k-1} = 1$, the circuit should return 0. When $B_{k-1} = A_{k-1}$, the result should defer to the result of the $\text{UGT}_{k-1}$ circuit. Hence the output is $B_{k-1}\overline{A_{k-1}}+(B_{k-1}A_{k-1}+\overline{B_{k-1}}\cdot\overline{A_{k-1}})R_{k-1}$ where $R_{k-1}$ is the result of the $\text{UGT}_{k-1}$.



(c) (7 pts) Now show how to build a general $\text{UGT}_k$ recursively with one or more $\text{UGT}_{k-1}$ circuits and 4-to-1 MUXes. Remember to clearly indicate how the inputs are feeding into the MUXes and the $\text{UGT}_{k-1}$ circuits.

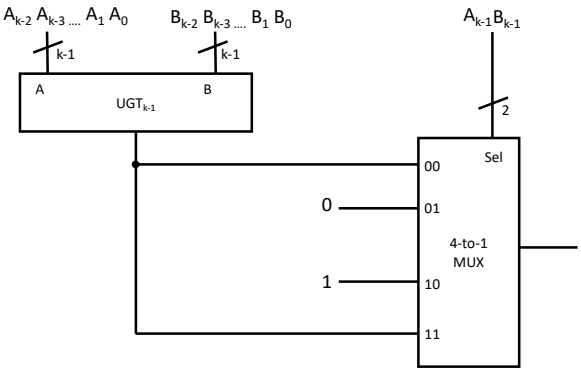
**Answer:** We only require a single MUX and a single $\text{UGT}_{k-1}$. We feed in $A_{k-1}B_{k-1}$ in as the selector, and feed $A_{k-2}A_{k-3}\cdots A_0$ and $B_{k-2}B_{k-3}\cdots B_0$ into the $\text{UGT}_{k-1}$. The outputs from the MUX correspond to the following combinations of $A_{k-1}B_{k-1}$:

| $A_{k-1}$ | $B_{k-1}$ | MUX should output |
|---|---|---|
| 0 | 0 | result from $\text{UGT}_{k-1}$ |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | result from $\text{UGT}_{k-1}$ |




(d) (8 pts) Suppose instead we wish to build a circuit that treats $k$-bit inputs $A$ and $B$ as signed 2s complement values. Show how to modify the design in part 2c for this purpose. Make sure you clearly indicate for which $i$ the $\text{UGT}_i$ can stay the same and for which it must be modified (as well as showing the modification).

**Answer:** First, let us consider when $A_{k-1}$ and $B_{k-1}$ differ. In this case, whichever is the 0 corresponds to the larger number, since it is positive, and a 1 would imply a negative number. When $A_{k-1} = B_{k-1} = 0$, then $A_{k-1}A_{k-2}\cdots A_0$ and $B_{k-1}B_{k-2}\cdots B_0$ are positive values and which is larger is returned by the $\text{UGT}_{k-1}$ circuit as in the previous part. When $A_{k-1} = B_{k-1} = 1$, then both are negative numbers. Which is larger is again returned by the $\text{UGT}_{k-1}$ as in the previous part. Specifically, the magnitude is given by starting at 0 and walking counter-clockwise around the wheel until reaching a bit-sequence matching the corresponding $k-1$-bit combination. Note that the smaller the value in unsigned, the farther the walk, i.e., the larger the magnitude, hence the value, which is the negation of the magnitide, is smaller. In other words, we rotate the least distance for all 1's, and the most distance for all 0's.



(e) (7 pts) Returning back to part (2b), suppose the circuit is represented in sum-of-products form (without further simplification). What circuit depth is achievable for the above design, where each AND, NOT, or OR gate used in series increases depth, and where AND and OR gates can take at most 2 inputs? Explain (in one or two sentences) why it has the depth you are claiming.

(Note: if you are finding part (e) difficult, realize it's only 7 points. Perhaps just move onto the next question, and come back after that one. Partial credit can also be received for answers that are approximately right, i.e., having the right intuition).

**Answer:** The $\text{UGT}_1$ has depth 2 (a NOT followed by an AND). Each additional recursion adds an extra OR gate for the $B_{k-1}A_{k-1}$ term, and working through the part in parentheses ANDing with a sum-of-product $R_{k-1}$ yields an additional OR gate and two additional terms into the product. Hence, at depth $i$, we require $2(i-1)$ OR gates that includes $2i$-way products. This puts the depth at $1 + \log_2(2i-2) + \log_2(2i)$, 1 to produce complements, $\log_2(2i-2)$ to perform the $2i-2$-input OR, and $log_2(2i)$ to perform the $2i$-input AND. This can be simplified to $3 + \log_2(i-1) + \log_2(i)$.

3. (35 pts) This is a MIPS coding problem. The function performed by the code are explained in great detail. However, the solution does not require a detailed understanding of what the code is doing. i.e., without even a detailed understanding, the problem can still be solved.

## What the code is supposed to do

For those interested in the details, the following describes a simple recursive process for multiplying two unsigned integers using only basic MIPS operations (i.e., SHIFT, AND and ADD).

(i) If a1 is odd (lowest-order bit is 1), then set s0 = a0, otherwise set s0=0.

(ii) shift-left a0 by 1 bit

(iii) shift-right a1 by 1 bit

(iv) Add s0 to the recursive call with the new a0, a1.

(v) repeat until a1 = 0.

The following shows an example with $a0 = 11$ and $a1 = 9$, where the lowest-order bit of $a1$ is shown underlined and the recursion is expanded out one more iteration in each subsequent line.

$$
\begin{array}{lllll}
1011 \cdot 100\underline{1} & = 1011 & +(10110 \cdot 10\underline{0}) & & \\
& = 1011 & +0 & +(101100 \cdot 1\underline{0}) & \\
& = 1011 & +0 & +0 & +(1011000 \cdot \underline{1}) \\
& = 1011 & +0 & +0 & +1011000 \quad +(10110000 \cdot \underline{0}) \\
& = 1011 & +0 & +0 & +1011000 \quad +0 \\
& = 1011 + 1011000 & & & \\
& = 1100011 = 99 & & &
\end{array}
$$

## The problem

The following page presents C code that implements the above procedure, using variables whose names correspond to MIPS registers. (i.e., $a0, a1, v0, s0, t0$). A **main** function loads in values for $a0$ and $a1$, and then calls a the recursive function **MIPSMult** to produce the result in $v0$. The program ends with a print statement that includes $a0, a1$ and $v0$.

On the page thereafter is MIPS code which is basically copied from the C-code. However, the code isn't running correctly, even for small values of $a0$ and $a1$. What changes should be made to the code? Give specific MIPS instructions and where they need to be added.

Here are some hints:

- The problem **does not** require modification to the code in the sections labeled "Input Info" and "Output Info" (i.e., lines those sections that basically use syscall). However, since those sections set and read values of $a0, a1, v0$, it may relate to those registers' values.
- The procedure may produce overflow for unsigned int inputs whose values are greater than $2^{16}$. You may assume the inputs are always smaller (i.e., not looking for a fix to the overflow problem).
- The code as-is returns the correct answer when $a1 = 0$ initially, but never terminates (is caught in some circular loop) for any other value of $a1$.

(This page may be detached from the rest of the exam)

```
#include <stdlib.h>
#include <stdio.h>




int MIPSMult(int a0, int a1){
    unsigned long s0;
    unsigned long v0;
    unsigned long t0;
    if (a1==0){
        v0 = 0;
        return v0;
    }
    s0 = 0;
    t0 = a1 & 1;
    if (t0 != 0){
        s0 = a0;
    }
    a0 = a0 << 1;
    a1 = a1 >> 1;
    v0 = MIPSMult(a0,a1);
    v0 = v0 + s0;
    return v0;
}


int  main(){
//#####################################
// Input info
    unsigned long a0,a1,v0;
    printf("Enter First Value: ");
    scanf("%lu", &a0);
    printf("Enter Second Value: ");
    scanf("%lu", &a1);
// End Input Info
//#####################################

    v0 = MIPSMult(a0,a1);

//#####################################
// Output Info and terminate
    printf("%lu x %lu = %lu\n", a0, a1, v0);
    return 1;
// End Output Info
//#####################################

}
```

(Intentionally left blank)

(This page may be detached from the rest of the exam)

```
1   .data
2
3   EnterFirstValue:    .asciiz "Enter first value: "
4   EnterSecondValue:    .asciiz "Enter second value: "
5
6   .text
7
8   MIPSMult:
9           bne $a1, $zero, NotBaseCase   ## C Code line 12
10          move $v0, $zero               ## C Code line 13
11          jr  $ra                       ## C Code line 13 cont'd
12  NotBaseCase:
13          move $s0, $zero               ## C Code line 16
14          and $t0, $a1, 1               ## C Code line 17
15          beq $t0, $zero, SkipSets0     ## C Code line 18
16          move $s0, $a0                 ## C Code line 19
17  SkipSets0:
18          sll $a0, $a0, 1               ## C Code line 21
19          srl $a1, $a1, 1               ## C Code line 22
20          jal MIPSMult                  ## C Code line 23
21          add $v0, $v0, $s0             ## C Code line 24 & 25
22          jr $ra                        ## C Code line 26
23
24
25  main:
26  ########### Input Info: this snippet correctly
27  ########### implements C code lines 31-36
28          la $a0, EnterFirstValue       ## C code line 33
29          li $v0, 4
30          syscall
31          li $v0, 5                     ## C code line 34
32          syscall
33          move $t0, $v0                 ## result should go in $a0
34                                        ## but syscall will use $a0
35                                        ## so keep in $t0 for now
36          la $a0, EnterSecondValue      ## C code line 35
37          li $v0, 4
38          syscall
39          li $v0, 5                     ## C code line 36
40          syscall
41          move $a1, $v0                 ## Move values in $v0 and
42          move $a0, $t0                 ## $t0 respectively
43                                        ## into $a1 and $a0
44
45  ############# End Input Info
46
47          jal MIPSMult                  ## C Code  line 40
48
49
50
```

(This page may be detached from the rest of the exam)

```
## This page intentionally left blank
```

(This page may be detached from the rest of the exam)

```
51 ############# Output Info: this snippet correctly
52 ############# implements C code lines 44 & 45
53
54         move $s0, $v0      ## Saving $v0 (in $s0) since
55                            ## needed for syscall
56         li $v0, 1          ## C code line 44, first %lu
57         syscall
58         li $a0, 'x'        ## C code line 44, print "x"
59         li $v0, 11
60         syscall
61         move $a0, $a1      ## move $a1 into $a0 for
62                            ## printing (syscall reqm't)
63         li $v0,1           ## C code line 44 second %lu
64         syscall
65         li $a0, '='        ## C code line 44, print "="
66         li $v0, 11
67         syscall
68         move $a0, $s0      ## solution, initially in $v0,
69                            ## now in $so, move into
70                            ## $a0 for printing (syscall reqm't)
71         li $v0, 1          ## C code line 44, third %lu
72         syscall
73         li $a0, '\n'       ## C code line 44, print "\n" (newline)
74         li $v0, 11
75         syscall
76         li $v0, 10         ## Exit program (similar to C code line 45)
77         syscall
78
79 ############## End Output Info
80
81 ############## THIS IS THE LAST PAGE OF THE EXAM
```

**Answer:** Whereas C will automatically scope variables, MIPS does not, and it is the programmer's responsibility to save values in registers that might be overwritten by procedure calls. This is most commonly done (and for recursive calls as we have here, it really needs to be done) via the stack.

Following proper convention (you didn't have to do this for full credit so there are other approaches that worked), the recursive **MIPSMult** procedure uses register $s0. This is a permanent register and so it is the called function's obligation to preserve its value at the start of the procedure if it is being overwritten, which indeed it is. Also, the $ra register must be preserved, as the **jal** instruction that performs a recursive call to MIPSMult will overwrite the prior value. It is especially important for the first call of **MIPSMult** to preserve this value, as the value in $ra holds where in the **main function** to return. So their temp values don't need to be preserved by **MIPSMult**. However, the values do need to be preserved by the **main** function, since the values in $a0 and $a1 are printed after the call from **main** to **MIPSMult**, and **MIPSMult** will be changing these values. If you did all your stack pushing and popping with **MIPSMult** including $a0 and $a1, this is fine. However, $ra and $s0 must be pushed in **MIPSMult** due to the recursive nature of the procedure.

Revised code with the stack instructions to save the register values follows. The added code is distinguished by comments with more than two #s.

(Solution Code)

```
1  .data
2
3  EnterFirstValue:     .asciiz  "Enter first value: "
4  EnterSecondValue:     .asciiz  "Enter second value: "
5
6  .text
7
8  MIPSMult:
9          bne $a1, $zero, NotBaseCase   ## C Code line 12
10         move $v0, $zero               ## C Code line 13
11         jr  $ra                       ## C Code line 13 cont'd
12 NotBaseCase:
13         sw $s0, -4($sp)              ######### Going to modify perm. $s0 and
14         sw $ra, -8 ($sp)             ######### recurse, so save $s0 and
15         addi $sp, $sp, -8            ######### $ra onto stack
16         move $s0, $zero               ## C Code line 16
17         and $t0, $a1, 1               ## C Code line 17
18         beq $t0, $zero, SkipSets0     ## C Code line 18
19         move $s0, $a0                 ## C Code line 19
20 SkipSets0:
21         sll $a0, $a0, 1               ## C Code line 21
22         srl $a1, $a1, 1               ## C Code line 22
23         jal MIPSMult                  ## C Code line 23
24         add $v0, $v0, $s0             ## C Code line 24 & 25
25         addi $sp, $sp, 8             ######### pop values prior to
26         lw $s0, -4($sp)              ######### return
27         lw $ra, -8($sp)              #########
28         jr $ra                        ## C Code line 26
29
30
31 main:
32 ########### Input Info: this snippet correctly
33 ########### implements C code lines 31-36
34         la $a0, EnterFirstValue       ## C code line 33
35         li $v0, 4
36         syscall
37         li $v0, 5                     ## C code line 34
38         syscall
39         move $t0, $v0                 ## result should go in $a0
40                                       ## but syscall will use $a0
41                                       ## so keep in $t0 for now
42         la $a0, EnterSecondValue      ## C code line 35
43         li $v0, 4
44         syscall
45         li $v0, 5                     ## C code line 36
46         syscall
47         move $a1, $v0                 ## Move values in $v0 and
48         move $a0, $t0                 ## $t0 respectively
49                                       ## into $a1 and $a0
50
51 ############## End Input Info
52         sw $a0, -4($sp)    ######### $a0 and $a1 are temp values
53         sw $a1, -8($sp)    ######### that may get overwritten by
54         addi $sp, $sp, -8  ######### MIPSMult procedure
55                            ######### so save on stack
56
57         jal MIPSMult                  ## C Code  line 40
58
59         addi $sp, $sp, 8   ######### Recover $a0 and $a1
60         lw $a0, -4($sp)    ######### after MIPSMult Call
61         lw $a1, -8($sp)    #########
62
63
```

(Solution Code)

```
63
64 ############ Output Info: this snippet correctly
65 ############ implements C code lines 44 & 45
66
67         move $s0, $v0      ## Saving $v0 (in $s0) since
68                            ## needed for syscall
69         li $v0, 1          ## C code line 44, first %lu
70         syscall
71         li $a0, 'x'        ## C code line 44, print "x"
72         li $v0, 11
73         syscall
74         move $a0, $a1      ## move $a1 into $a0 for
75                            ## printing (syscall reqm't)
76         li $v0,1           ## C code line 44 second %lu
77         syscall
78         li $a0, '='        ## C code line 44, print "="
79         li $v0, 11
80         syscall
81         move $a0, $s0      ## solution, initially in $v0,
82                            ## now in $so, move into
83                            ## $a0 for printing (syscall reqm't)
84         li $v0, 1          ## C code line 44, third %lu
85         syscall
86         li $a0, '\n'       ## C code line 44, print "\n" (newline)
87         li $v0, 11
88         syscall
89         li $v0, 10         ## Exit program (similar to C code line 45)
90         syscall
91
92 ############## End Output Info
93
94 ############## THIS IS THE LAST PAGE OF THE EXAM
```