

1 Instructions

This assignment includes exercises covering assembly language procedures, discussed in Ch. 2, and the IEEE 754 floating point standard, which is discussed in the Ch. 3 lecture notes. First, read §2 which discusses how to define and use one- and two-dimensional arrays (1D and 2D arrays) as parameters and local variables in function. Then complete Ch. 2 Exercises 1 and 2, which are assembly language functions involving arrays, if statements, and loops. Next, read §6 and §7 on IEEE 754 rounding modes and converting numbers among different bases using Wolfram Alpha. Finally, complete Ch. 3 Exercises 3–5 in §5.

You may work in pairs with a partner on this assignment if you wish or you may work alone. If you work with a partner, only submit one PDF file with both of your names in the document; you will each earn the same number of points. Your PDF file must be uploaded to Blackboard by the assignment deadline. Section 8 describes what to submit and by when.

2 1D Arrays as Parameters and Local Variables

We discussed earlier how to define and allocate a global 1D array variable in the data section. What about defining array variables as local variables and passing them as function arguments?

First, to define an array local variable in a function, the array must be allocated in the stack frame of the function. For example,

```
void main()
{
    char name[255];
    ...
    print_string(name);
}
```

To allocate *name* requires *main()* to allocate 255 bytes in its stack frame for *name*. However, to maintain word-alignment, the size of data allocated in the stack frame should be a power of 4 (or divisible by 4) and 255 is not a power of 4, but 256 is. Consequently, *main()* will allocate *name* using 256 bytes, with the address of *name* being 0(\$sp),

```
main:
    addi    $sp, $sp, -256        # Allocate 256 bytes for 255-char array name
    ...
    move    $a0, $sp             # Place the address of name ($sp) into $a0 prior to the funct call
    jal     print_string         # Call print_string() with $a0 containing the address of name
    ...
    addi    $sp, $sp, 256        # Deallocate stack frame
    addi    $v0, $zero, SYS_EXIT  # $v0 = SysExit service code
    syscall                               # SysExit()
```

Note that in C, the name of an array variable, i.e., *name*, is equivalent to the address of the variable in memory. Notice the move instruction in *main()*: to pass *name* as the argument to *print_string* requires us to load the address of *name* into the \$a0 register, and 0(\$sp) is the address of *name*. In C, when an array variable is passed as a function argument, we do not pass a copy of the entire array; rather, to save both time and memory, we simply pass the address of the array. Since addresses are 4-bytes in our MIPS32 system, we only need to send 4 bytes to the callee when passing an array that consists of one million elements!

See *print_string()* below. Notice in *main()* that the array of characters is called *name* while in *print_string()*, the array is called *string*. In high-level programming languages like C and Java, the names of the arguments in the function call do not need to be the same as the names of the parameters in the function header. Note, also that *name* and *string* both refer to the same 1D array of characters allocated in *main*'s stack frame. That is, if we were to modify the contents of *string* in *print_string()* we would really be modifying the block of characters of *name* in *main*'s stack frame, which means upon returning to *main()*, the contents of *name* will have been changed. This may or may not be desirable.

```
void print_string(char string[])
{
    for (int i = 0; string[i] != '\0'; ++i) { -- Loop over the chars of string at indices 0, 1, 2, ...
        SysPrintChar(string[i]);           -- stop looping and printing when the null char is reached
    }
}
```

Note that on entry to `print_string()`, `string` will contain in `$a0` the address of `name` allocated in `main`'s stack frame. What we have in `string` is what C programmers call a **pointer**, making `string` a **pointer variable**. A pointer variable is just like an integer variable, except the integer stored in a pointer variable is **always the memory address** of where some data are located. Since `string` contains the address of the character array `name` found in `main`'s stack frame, we can mentally picture `string` pointing to `name`. Let us complete the code for `print_string()` so you will see how to use `string`. First, rewrite `print_string()` changing the **for** loop into a **while** loop,

```
void print_string(char string[])
{
    int i = 0;
    while (string[i] != '\0') {
        SysPrintChar(string[i])
        ++i;
    }
}
```

Next, translate the code using a **while** loop into code using an **if** statement and **goto**,

```
void print_string(char string[])
{
    int i = 0;                                // i is the index in string of the char to be printed
begin_loop:
    if (string[i] == '\0') goto end_loop;      // Drop out of the loop when the null char is reached
    SysPrintChar(string[i]);                   // Print the char at string[i]
    ++i;                                       // Update i to be the index of the next char in string
    goto begin_loop;                          // Continue looping...
end_loop:                                     // We come here when the null char is reached
}
```

Finally, this code maps directly onto the assembly language code. `print_string()` is a leaf procedure so we do not need to save `$ra` (syscalls are not function calls, they do not change `$ra`). Note that we *do* need to save `$a0` in the stack frame because it contains parameter `string` and when performing the syscall, we need to place the ASCII value of the character to be printed into `$a0`. I am also intentionally not optimizing this code so you can better see how local variables are allocated and accessed.

```
print_string:
# int i = 0;
    addi    $sp, $sp, -8                    # Allocate 2 words in stack frame: i at 0($sp), $a0 at 4($sp)
    sw      $a0, 4($sp)                    # Save $a0 (param string) in stack frame
    sw      $zero, 0($sp)                  # Initialize i in stack frame to 0

# if (string[i] == '\0') goto end_loop
    lw      $t0, 0($s0)                    # $t0 = i
    lw      $t1, 4($sp)                    # $t1 = string
    add     $t0, $t1, $t0                  # $t0 = string + i = &string[i]
    lbu     $a0, 0($t0)                    # $a0[7:0] = string[i] (note: lbu loads a byte/char)
    beq     $a0, $zero, end_loop           # if (string[i] == '\0') goto end_loop
```

```

# SysPrintChar(string[i])
    addi    $v0, $zero, SYS_PRINT_CHAR    # $v0 = SysPrintChar service code
    syscall                                # SysPrintChar(string[i])

# ++i;
    lw      $t0, 0($sp)                   # $t0 = i
    addi    $t0, $t0, 1                   # $t0 = i + 1
    sw      $t0, 0($sp)                   # ++i

# goto begin_loop
    j       begin_loop                   # Continue looping

end_loop:
    addi    $sp, $sp, 8                   # Deallocate stack frame
    jr      $ra                           # Return

```

It is important that you study this code, especially the instructions in `print_string()` that involve the local variable `i` and parameter `string`. Note in particular how the address of `string[index]` is computed in the `add $t0, $t1, $t0` instruction. Since `string` is the address of `name` in `main`'s stack frame, `string + i` is the address of `string[i]` which is really the address of `name[i]`.

One final note on 1D array addressing. The example used an array of characters, where the size of each array element was 1, since a character in our MIPS system consumes 1 byte of memory. If we are working with an array of integers, where each int occupies 4 bytes of memory, then the general formula for calculating the address of a 1D array element is: $\&a[i] = a + i \cdot \text{sizeof}(\text{type})$, where `type` is the data type of each element of the array and the C `sizeof` operator evaluates to the number of bytes allocated for `type`. For a character array, `sizeof(char)` is 1, but for an integer array, `sizeof(int)` is 4. Hence, if `a` is an array of `ints`, to find the address of the element at index 5, the formula is: $\&a[5] = a + 5 \cdot 4 = a + 20$, and in general, it would be $\&a[i] = a + 4i$.

3 2D Arrays as Parameters and Local Variables

In programming, we mentally picture a two-dimensional array as a table consisting of rows and columns of elements. However, in a computer system, the memory address space is one-dimensional (or linear), i.e., memory can be pictured as just a very big 1D array. So how do we store a 2D data structure in 1D space? There are two common methods. The one described here is the most common method, as most popular programming languages do it this way.

The idea is to recognize that each row of a 2D array (table) is simply a 1D array of elements, e.g., consider this 2D array: `int a[2, 3]`, consisting of two rows and three columns. The method stores the elements for row 0 in contiguous memory locations, then it stores the elements for row 1 in memory following the elements of row 0, for example,

```

+-----+      +---+
| a[1][2] | 0x7FFF_EA14 |
+-----+      |
| a[1][1] | 0x7FFF_EA10 | Row 1
+-----+      |
| a[1][0] | 0x7FFF_EA0C |
+-----+      +---+
| a[0][2] | 0x7FFF_EA08 |
+-----+      |
| a[0][1] | 0x7FFF_EA04 | Row 0
+-----+      |
| a[0][0] | 0x7FFF_EA00 = a |
+-----+      +---+

```

Storing the array elements by rows in this manner is referred to as **row-major order**. The other method is to store all of the elements for column 0 first, then the elements for column 1, then the elements for column 2, and so on. That method is called **column-major order**. The majority of programming languages I know of use row-major order. The only language I know that uses column-major order is FORTRAN.

Note that the name of the array is a and the arbitrarily-chosen address of a is `0x7FFF_EA00` (remember, the name of an array variable is equivalent to the address of the array). In a manner similar to calculating the address of a 1D array element, to calculate the address of a 2D array element, we need to know three things: the address of the array, the size in bytes of each element of the array, and additionally, we must know the number of columns in the 2D array (call this value num_cols). Then, the general formula is: $\&a[r][c] = a + r \cdot num_cols \cdot sizeof(type) + c \cdot sizeof(type)$ which can be simplified to $\&a[r][c] = a + (r \cdot num_cols + c) \cdot sizeof(type)$.

For example, try this out: what is the address of $a[1][1]$? a is `0x7FFF_EA00`, $r = 1$, $c = 1$, $sizeof(int) = 4$, and $num_cols = 3$, so we have $\&a[1][1] = a + (1 \cdot 3 + 1) \cdot 4 = 0x7FFF_EA00 + (4) \cdot 4 = 0x7FFF_EA00 + 0x10 = 0x7FFF_EA10$.

4 Chapter 2 Exercises

1. (8 pts) **MIPS32 Assembly Language Programming - *strlen()* Function (modify *h4-1.s*)**. In §2.8 of the Ch. 2 lecture notes, we discussed how to write and use assembly language functions. We have also previously discussed how C-strings are stored in memory as a one-dimensional array of characters with a null character `'\0'` (with ASCII value 0) following the last character of the string. For this exercise, you will write the assembly language instructions to implement the function `int strlen(char string[])` which computes and returns the length of the C-string *string*. Note, the length of a C-string is the number of characters in the string, excluding the null character, e.g., if *string* is "Wilma" then what is stored in six consecutive bytes of memory for *string* is 87, 105, 108, 110, 103, 0, which are the decimal ASCII values for 'W', 'i', 'l', 'm', 'a', and `'\0'`, respectively. Therefore, *strlen()* must return 5. Here is the pseudocode for *strlen()*,

```
function strlen(char string[])
    local int index ← 0
    while string[index] ≠ '\0' do -- walk thru string starting at the beginning looking for the null char
        ++index
    end while
    return index -- when we drop out of the while loop, index is the index of the null char within string
end function strlen -- index also happens to be a count of the number of non-null chars that were encountered
```

Complete the following exercises:

- a. First, rewrite this function replacing the **while** loop with equivalent code using an **if** statement and a **goto**. Insert the neatly-typed revised function into your solution document.
- b. Translate the code using the **if** statement and **goto** into MIPS assembly language. Insert the assembly language code into *h4-1.s* following the definition of *main()*. Test your function by running *main()* with several test cases.

Miscellaneous notes, hints, and requirements:

1. Study the assembly language source code files listed in the course notes and posted on the course website. Format your code in a similar manner, i.e., an assembly language line containing an instruction consists of four columns of mostly-optional stuff: column 1 is aligned with the left margin and is reserved for an optional label; column 2 is indented from column 1 by around 4-8 spaces and is reserved for instruction mnemonics; column 3 is indented from column 2 and reserved for optional operands; column four is indented from column 3 and is reserved for optional comments. How many spaces or tabs you indent is up to you, the important thing is to line things up in columns and be consistent.
2. See how I commented *main()* in *h4-1.s*. In *strlen()*, write a meaningful comment in column 4 of each line of code containing an instruction. This may seem like busy work, but it can be very helpful when your code does not work and you are trying to figure out why. Trust me on this one.
3. Note that *strlen()* is a **leaf procedure** (because it does not call other functions) and as long as there are enough registers to store all of the local variables, a leaf procedure does not really need to create a stack frame because it does not need save `$ra` or any of the `$ax` argument registers. However, the primary objective of this exercise is for

you to demonstrate that you know how to write a function that creates a stack frame, allocates local variables within the stack frame, accesses local variables in the stack frame, and destroys the stack frame before returning. Consequently, your solution **shall** create a stack frame for *strlen()* and **shall** allocate local variable *index* in the stack frame at 0(\$sp). Whenever the code must read or write *index*, write the lw and sw instructions to read and write *index* from/to memory, i.e., do not optimize the code by simply storing *index* in a register and always accessing it from there. Study how I wrote the code for *main()* which is also unoptimized, e.g., on line 103 I wrote the return value from *strlen()*—which is in \$v0—into the stack frame word allocated for *len*. Then, on line 107, I wrote an instruction which loads the value of *len* from the stack frame into \$a0. If I were optimizing the code, I could have just written `move $a0, $v0` on line 107, rather than accessing memory, but again, the primary objective of this exercise is for you to prove to the grader that you know how to create and use a stack frame.

4. Because *SysReadStr* always writes a newline character into the string buffer following the last typed character (well, it always does this when the size of the buffer is large enough to store the newline character), then the smallest string we can enter would be to simply hit the Enter key when the prompt is displayed. This will cause *string* to be set to "\n", for which *strlen()* will return 1. Using *SysReadStr* the way we are using it, it is impossible to type the empty string "" for testing. However, testing the empty string can be done this way: set a breakpoint in the debugger on the line containing the `jal strlen` instruction, (as long as you have not modified *main()*, this will be line 125; in the debugger, the instruction will be at 0x0040_0028). Run the program and when prompted for the string, just hit the Enter key. In the *Data Segment* view of the MARS debugger window, click on the button (next to the green arrows) that allows us to display different regions of memory; select *current \$sp* from the drop-down list box. Select Hexadecimal Addresses and Hexadecimal Values. If the ASCII check box is selected, unselect it. Look at the stack diagram for *main()*—found in the header comment block for *main()* starting on line 84—and note that the address of *string* is 0x7fff_efa8. In the *Data Segment* view, locate the word stored in memory at that address, and you should see 0x0000000a as the value of this word. Place the cursor inside the GUI box for the word and double-click. Edit the value of this word to be 0x00000000 and press Enter to save the changes. We have now changed the *string* stored in memory from "\n" to "". Now click the Run button to allow *strlen()* to be called and you should see 0 for the output. The empty string "" has length 0.
 5. Update the header comment block of *h4-1.s* with your name and email address, if you have a partner, his/her name and email address, and your lecture time: 10:45am or 2:00pm.
2. (27 pts) MIPS32 Assembly Language Programming - *write_grid()* Function (modify *h4-2.s*). **Encryption** is the process of converting information from one form to another form so as to make the original information unintelligible. For example, we may wish to encrypt a **plaintext** message "Gunther is loose again. Release the hounds." into the encrypted **ciphertext** message "Giedulnsno.t.tohhsReeeerlhaoigaasasn".

A multitude of encryption algorithms have been devised over the last couple of millennia. Some are better than others; the strength of an encryption algorithm *E* is a measure of how difficult it is to decrypt a message encrypted using *E* without knowing how *E* works or what encryption key(s) *E* uses. That is, if Bob uses secret encryption algorithm *E* with secret key *K* to encrypt a plaintext message *plain* forming an encrypted ciphertext message *cipher*, and if Alice does not know how *E* works or she does not know *K*, then if *E* is a strong encryption method, it will be extremely difficult—if not impossible—for Alice to decrypt *cipher* so she can read *plain*.

Strong encryption algorithms are very complex and very difficult to design. One of the best and well-known strong encryption algorithms is the **Advanced Encryption Standard** (AES) endorsed by the U.S. National Institute of Standards and Technology (NIST) and the National Security Agency (NSA). Weak or lousy encryption algorithms, on the other hand, are quite easy to devise. For example, I will describe a method that we will implement in MIPS assembly language.

When the program starts, it will display a prompt requesting the user to enter the plaintext message *plain*, with $1 \leq \text{strlen}(\text{plain}) \leq 80$. Next, it writes the individual characters of *plain* into a 2D array of characters named *grid*, containing *NUM_ROWS* = 8 rows and *NUM_COLS* = 10 columns.

For example,

```
Plaintext (80 or fewer chars)? Gunther is loose again. Release the hounds.
```

When the plaintext message is entered and read using the *SysReadStr* system call, the trailing newline character that is generated when the user presses the Enter key is placed in the string buffer following the final period (and following that byte in memory will be the null character '\0'). Excluding the newline character, in this example, the length of *plain* is 43. We write the individual characters of *plain* into *grid* in **row-major order** (row-major order means we fill up an entire row *r* before moving to row *r*+1).

G	u	n	t	h	e	r		i	s
	l	o	o	s	e		a	g	a
i	n	.		R	e	l	e	a	s
e		t	h	e		h	o	u	n
d	s	.							

where represents the space character. Note that unused elements of *grid* are written with spaces. Next, to create the ciphertext message *cipher* we traverse the elements of *grid* in **column-major order**, appending each **non-space** character to *cipher* (column-major order means we append all of the characters in an entire column *c* before moving to column *c*+1). In this example, *cipher* will become "Giedulnsno.t.tohhsReeeerlhaeoigaussn". Finally, the program shall output *cipher*,

```
Plaintext (80 or fewer chars)? Gunther is loose again. Release the hounds.
Ciphertext: Giedulnsno.t.tohhsReeeerlhaeoigaussn
```

The equivalent C-like pseudocode for the program is shown below.

```
-- Note: Define these constants using the .eqv directive and use the constants in the assembly language code, e.g.,
-- in at least one instruction, you will need to load the value of NUM_COLS into a register. To do this, write:
-- addi $t0, $zero, NUM_COLS rather than addi $v0, $zero, 10. Remember, the entire reason we define constants
-- is to enhance the readability of the code, and using well-named constants does that.
global constant BUF_LEN      ← 84 -- Size of the array string allocated in main().
global constant SYS_RS_MAX   ← 82 -- Maximum number of chars to read when calling SysReadStr.
global constant NUM_COLS     ← 10 -- Number of columns in the 2D grid.
global constant NUM_ROWS     ← 8  -- Number of rows in the 2D grid.

function main()
    local char string[BUF_LEN], grid[NUM_ROWS][NUM_COLS] -- local vars are allocated in stack frame
    SysPrintStr("Plaintext (80 or fewer chars)? ") -- Display the prompt for the plaintext message.
    SysReadStr(string, SYS_RS_MAX) -- Read the plaintext message into string.
    write_grid(grid, string) -- Write plain into grid in row-major order.
    read_grid(grid, string) -- Read ciphertext msg by traversing grid in column-major order
    SysPrintStr("Ciphertext: ")
    SysPrintStr(string) -- Display the ciphertext message.
    SysExit() -- Terminate the program.
end function main
```

```

function read_grid(char grid[], char cipher[]) -- grid is a 2D array of chars; cipher is a 1D array of chars
    local int col, index ← 0, row
    for col ← 0 to NUM_COLS - 1 do           -- Because the for column loop is the outer loop and the for row loop
        for row ← 0 to NUM_ROWS - 1 do       -- is the inner loop, we traverse grid in column-major order.
            if grid[row][col] ≠ ' ' then      -- We do not append space chars to cipher.
                cipher[index] ← grid[row][col] -- Append the char at grid[row][col] to cipher.
                ++index                       -- Update index to point to the next free element in cipher.
            end if
        end for
    end for
    cipher[index] ← '\0' -- We must append the required null char following the last char of cipher
end function read_grid

function strlen(char string[]) -- Copy-and-paste your strlen() function of Exercise 1 into h4-2.s here.
    ...
end function strlen

function write_grid(grid, plain)
    local int col, index ← 0, len ← strlen(plain) - 1, row -- subtract 1 from strlen() to skip newline char at the end
    for row ← 0 to NUM_ROWS do           -- Because the for row loop is the outer loop and the for column loop
        for col ← 0 to NUM_COLS do       -- is the inner loop, we traverse grid in row-major order.
            if index < len then          -- Does index still refer to a valid char in plain?
                grid[row][col] ← plain[index] -- Yes, write the char to grid.
                ++index                     -- Update index to refer to the next char in plain.
            else
                grid[row][col] ← ' '       -- Else, no, we have copied all of the chars from plain into grid
            end if                         -- so fill remaining unused grid elements with spaces
        end for
    end for
end function write_grid

```

You do not need to write the entire program as I have provided a template in *h4-2.s* containing the completed code for *main()* and *read_grid()*. You will insert the code for the *strlen()* function you wrote in Exercise 1 into *h4-2.s* so it may be called from *write_grid()*, leaving *write_grid()* as the only function remaining to be implemented by you.

Complete the following exercises.

- First, rewrite the pseudocode for *write_grid()*, replacing the **for** loops with equivalent code using **while** loops. Insert the neatly-typed revised function into your solution document and identify it as Version 2 of *write_grid()*.
- Next, rewrite Version 2, replacing the **while** loops with equivalent code using **if** statements and **goto**'s. Insert the neatly-typed revised function into your solution document and identify it as Version 3.
- Next, rewrite Version 3, replacing the structured **if-else** statement with unstructured code that uses an **if** statement (not an if-else statement) and **goto**'s. Insert the neatly-typed revised function into your solution document and identify it as Version 4.
- Version 4 is now in a format that maps directly onto the assembly language implementation. Translate Version 4 to MIPS assembly language by writing the instructions for *write_grid()* in *h4-2.s*.

- e. Testing. To ensure your program works correctly, you must test it using several test cases. I will give you two, but you should test your program on other test cases as well.

Test Case 1

Input (*plaintext*): Gunther is loose again. Release the hounds.

Expected output (*ciphertext*): Giedulnsno.t.tohhsReeeerlhaeoigaasasn

Test Case 2

Input (*plaintext*): Now is the time for all good men to come to the aid of their country.

Expected output (*ciphertext*): Naedotlntcwilooomtfuiegotnsotthtfocehrtodoeyhrmai.emeir

Miscellaneous notes, hints, and requirements (see the same Miscellaneous section described in Exercise 1):

1. Neatly format your typed code.
2. Write a meaningful and helpful comment in column 4 for each instruction of *write_grid()*.
3. Study *read_grid()*. The code for *write_grid()* is somewhat similar.
4. Since the primary objective of this exercise is for the student to demonstrate that he/she knows how to create a stack frame, how to allocate local variables in the stack frame, how to access local variables in the stack frame during the execution of the function, and how to destroy the stack frame before returning from the function, *write_grid()* **shall** create and use a stack frame. Furthermore, even though we have plenty of free registers for use in *write_grid()*, you **shall not** optimize the code in a way that does not allocate or access the local variables in the stack frame (by keeping the values in registers during the function). See *read_grid()* for an example of how the instructor did this.
4. Update the header comment block of *h4-1.s* with your name and email address, if you have a partner, his/her name and email address, and your lecture time: 10:45am or 2:00pm.

5 Chapter 3 Exercises

Please read §6–§7 before completing these exercises. For each of these exercises, in addition to providing what you believe is the correct answer, you must also show your work and/or explain how your answer was obtained for full credit.

3. (5 pts) What would be the IEEE 754 single precision floating point representation of $n = -34543210.012345987654321 \times 10^{12}$? For explanation, I want you to document the steps you perform, in this order: (1) What is n in decimal *fixed point* form (*ddd.ddddd*); (2) What is n in binary *fixed point* form (*bbb.bbbb*), storing the first 25 bits following the binary point; (3) What is the normalized binary number, written in the form $1.bbbbb...bbb \times 2^e$, storing 25 bits following the binary point? (4) What are the 23 mantissa bits, after the bits in bit positions -24, -25, ... are eliminated using the round to nearest, ties to even mode; exclude the 1. part; (5) What is the biased exponent in decimal and in binary? (6) Write the 32-bits of the number in the order: *s e m*; and (7) Write the final answer as an 8-hexdigit number.
4. (5 pts) What decimal floating point number does this big-endian IEEE 754 single precision number represent: $n = 0xF4E3_C2D1$? For explanation, I want you to document the steps you perform, in this order: (1) What is n in binary; (2) What is the value of the sign bit; What does this value signify about the final number; (3) What are the binary and decimal values of the biased exponent; (4) What is the binary value of the mantissa, with the 1. part preceding the binary point? (5) What is the decimal value of the unbiased exponent; (6) What is the decimal value of the mantissa, with the leading 1. part? (7) What is the final decimal real number, written in the form $[-]d.ddddddddd \times 10^e$ where d represents a decimal digit 0-9 and there is an optional leading negative sign. Write exactly 15 digits after the decimal point (even if they are 0's) and round the final 15th digit up or down as required based on the value of the 16th digit (16th digit < 5 round down; otherwise, round up).
5. (5 pts) What would be the IEEE 754 *double* precision floating point representation of $1.827509156530856712385673895965827169405837361 \times 10^{-15}$. For explanation, I want you to document the steps you perform, in this order: (1) What is n in decimal *fixed point* form (*ddd.ddddd*); (2) What is n in binary *fixed point* form (*bbb.bbbb*), storing the

first 110 bits following the binary point); (3) What is the normalized binary number, written in the form $1.bbbbb\dots bbb \times 2^e$, storing 54 bits following the binary point) (4) What are the 52 mantissa bits, after the bits in bit positions -53, -54, ... are eliminated using the round to nearest, ties to even mode; exclude the 1. part; (5) What is the biased exponent in decimal and in binary? (6) Write the 64-bits of the number in the order: $s e m$; and (7) Write the final answer as a 16-hexdigit number.

6 IEEE 754 Rounding Modes

Review IEEE 754 rounding modes discussed in §3.6 of the Chapter 3 lecture notes. Here is an algorithm I hope will help you learn how round to nearest, ties to even rounding mode works.

-- When converting a number in decimal to binary floating point representation, after the number has been converted
 -- to binary normalized form, the number will be something like $1.mmm\dots m \times 2^e$, where the 25 m bits following the
 -- binary point are the input to this function. $m_{1:-25}$ are the 25-bits mantissa bit following the binary point (the first
 -- bit following the binary point is in bit position -1, the second bit following the binary point is in bit position -2, ...,
 -- and the twenty-fifth bit is in bit position -25. The return value $n_{1:-23}$ is the 23-bit mantissa which has been rounded
 -- using the round to nearest, ties to even rounding mode.

```
function round_mantissa (input:  $m_{1:-25}$ ) returns  $n_{1:-23}$  -- 25 mantissa bits are input, only 23 are stored
  if  $m_{24:-25} = 00_2$  or  $m_{24:-25} = 01_2$  then -- round down to nearest number (this is equivalent to truncation)
     $n_{1:-23} \leftarrow m_{1:-23}$  -- the rounded mantissa is just the 23 bits following the binary pt in  $m$ 
  elseif  $m_{24:-25} = 11_2$  then -- round up to nearest number (adding 1 to  $m_{1:-23}$  does the job)
     $n_{1:-23} \leftarrow m_{1:-23} + 1$ 
  elseif  $m_{23} = 0$  then -- else,  $m_{24:-25}$  is  $10_2$  so we have a tie;  $m_{23}$  being 0 means  $m_{1:-23}$  is even
     $n_{1:-23} \leftarrow m_{1:-23}$  -- since  $m_{1:-23}$  is even we round to even (which is equivalent to truncation)
  else -- else,  $m_{24:-25}$  is  $10_2$  so we have a tie;  $m_{23}$  being 1 means  $m_{1:-23}$  is odd
     $n_{1:-23} \leftarrow m_{1:-23} + 1$  -- round up to nearest even number (adding 1 to  $m_{1:-23}$  does the job)
  end if
  return  $n_{1:-23}$ 
end function
```

For example, consider $m_{1:-25} = 10101010101010101010101010101$ where bits $m_{24:-25}$ are underlined and bits $m_{22:-23}$ are in bold. The first conditional expression checks to see if $m_{24:-25}$ is 00_2 or 01_2 and since $m_{24:-25}$ is 01_2 , the conditional expression is true. Therefore, we eliminate bits in m following bit position -23 by rounding down to the nearest number, which is equivalent to truncating $m_{24:-25}$ so $n_{1:-23} \leftarrow m_{1:-23} = 10101010101010101010101010101$ (notice $m_{22:-23}$ remain unchanged).

Another example, consider $m_{1:-25} = 10101010101010101010101010111$. The first conditional expression checks to see if $m_{24:-25}$ is 00_2 or 01_2 and is false; the second conditional expression checks to see if $m_{24:-25}$ is 11_2 , which it is. Therefore, we eliminate bits in m following bit position -23 by rounding up to the nearest number, which is equivalent to adding 1 to $m_{24:-25}$ so $n_{1:-23} \leftarrow m_{1:-23} + 1 = 10101010101010101010101010101 + 1 = 10101010101010101010101010110$ (notice $m_{22:-23}$ changed from 01_2 to 10_2).

A tie occurs when $m_{24:-25} = 10_2$ and to break the tie, we must round m to the nearest even number. To tell if $m_{1:-23}$ is even or odd, it is sufficient to determine if $m_{23} = 0$ or 1, respectively. (Remember, an even binary integer is divisible by 2 and a binary integer that is divisible by 2 will have a 0 in the lsb. If the integer is odd, the lsb will be 1.) When $m_{24:-25} = 10_2$ and m_{23} is 0 (meaning $m_{1:-23}$ is even), we break the tie by rounding to the nearest even number via truncation, i.e., $n_{1:-23} \leftarrow m_{1:-23}$ will make $n_{1:-23}$ an even number. On the other hand, when $m_{24:-25} = 10_2$ and m_{23} is 1 (meaning $m_{1:-23}$ is odd), and because we round the to nearest even number, $n_{1:-23} \leftarrow m_{1:-23} + 1$ will make $n_{1:-23}$ an even number.

For example, suppose $m_{1:-23}$ is $10101010101010101010101010110$. Since $m_{24:-25} = 10_2$, we have a tie (m could be rounded up or rounded down, and either way, the introduced roundoff error would be the same). We next examine m_{23} and see that it is 1, indicating that $m_{1:-23}$ is odd. Therefore, we break the tie by rounding $m_{1:-23}$ to the nearest even number which is

Conversely, suppose $m_{\mathbf{1:-23}}$ is 10101010101010101010**10**10. Since $m_{\mathbf{24:-25}} = 10_2$ we have a tie. We next examine $m_{\mathbf{23}}$ and see that it is 0 indicating that $m_{\mathbf{1:-23}}$ is even. Therefore, we break the tie by rounding $m_{\mathbf{1:-23}}$ to the nearest even number which is accomplished by truncating bits $m_{\mathbf{24:-25}}$ resulting in $n_{\mathbf{1:-23}} \leftarrow m_{\mathbf{1:-23}} = 10101010101010101010**10**$.

You may use any calculator you wish to perform the conversions of numbers between bases, including doing it manually if you are so inclined (practice with small numbers so you can quickly answer these types of questions on the exams). I like to use Wolfram Alpha¹. In Alpha, to convert a real, decimal number to binary, e.g., 137.123, type the command **137.123 to binary** (**137.123 to base 2** performs the same operation). Alpha will respond with the binary representation in either fixed or exponential notation (in the *Result* field). Clicking the *More Digits* button will display more bits after the binary point.

To convert a binary number to decimal, e.g., 10010101.100101_2 , type the command **10010101.100101_2 to decimal** (the `_2` part is required to specify a base other than 10). Incidentally, if you want to know what 10010101.100101_2 is in base 73, try **10010101.100101_2 to base 73** (the first few digits are 2.3421460355512369...).

Create a word processing document or text file and neatly type your solutions to Exercises 1.a, 2.a–2.c, and 3–5 in this document. Make sure to put the names of both partners in the document if you worked with a partner. Convert this document into a PDF file name **cse230-f16-h04-asurite.pdf** or **cse230-f16-h04-asurite1-asurite2.pdf**, where *asurite* is the user name you use to log into My ASU and Blackboard. If you work with a partner, put both user names in the file name. Next, create an empty folder named **cse230-f16-h04-asurite** or **cse230-f16-h04-asurite1-asurite2**. Copy your PDF document, *h4-1.s*, and *h4-2.s* into this folder; there should only be three files in the folder. Then compress the folder creating a **.zip** archive named either **cse230-h04-asurite.zip** or **cse230-h04-asurite1-asurite2.zip**. Upload the zip archive to Blackboard using the homework submission link by the deadline, which is **4:00am Wed 26 Oct**. Consult the online syllabus for the late and academic integrity policies.

(c) Kevin R. Burger :: Computer Science & Engineering :: Arizona State University