6.190 Quiz Review Session

Practice Quiz from Fall 2022 (First Quarter)

Problem 1

Binary Arithmetic

A. What is 0x68 ^ (0x9C | 0x5A)? Provide your result in both unsigned 8-bit binary and unsigned 8-bit hexadecimal.

A. What is 0x68 ^ (0x9C | 0x5A)? Provide your result in both unsigned 8-bit binary and unsigned 8-bit hexadecimal.

Bitwise operators

A. What is 0x68 ^ (0x9C | 0x5A)? Provide your result in both unsigned 8-bit binary and unsigned 8-bit hexadecimal.

- `0x9C              = 0b1001_1100`
- `0x5A              = 0b0101_1010`
- `(0x9C | 0x5A)     = 0b1101_1110 = 0xDE`

- `0xDE              = 0b1101_1110`
- `0x68              = 0b0110_1000`
- `0x68 ^ 0xDE       =` **`0b1011_0110 = 0xB6`**

XOR		
a	b	out
0	0	0
0	1	1
1	0	1
1	1	0

OR		
a	b	out
0	0	0
0	1	1
1	0	1
1	1	1

A. What is 0x68 ^ (0x9C | 0x5A)? Provide your result in both unsigned 8-bit binary and unsigned 8-bit hexadecimal.

9 C

- 0x9C = 0b1001_1100
- 0x5A = 0b0101_1010
- (0x9C | 0x5A) = 0b1101_1110 = 0xDE

- 0xDE = 0b1101_1110
- 0x68 = 0b0110_1000
- 0x68 ^ 0xDE = **0b1011_0110** = **0xB6**

XOR		
a	b	out
0	0	0
0	1	1
1	0	1
1	1	0

OR		
a	b	out
0	0	0
0	1	1
1	0	1
1	1	1

B. What is the result of ((0b001 > 0b101) && 0b100) == 0b001)? Assume all numbers are unsigned. Provide your result in decimal.

B. What is the result of ((0b001 > 0b101) && 0b100) == 0b001)? Assume all numbers are unsigned. Provide your result in decimal.

Logical + Relational operators
(aka not bitwise)

B. What is the result of ((0b001 > 0b101) && 0b100) == 0b001)? Assume all numbers are unsigned. Provide your result in decimal.

- 0b001 = 1
- 0b100 = 4
- 0b101 = 5
- 1 > 5 is False (0)
- (0 && 4) = 0
- (0 == 1) = **0**

C. (4 points): What are 14 and 31 in 8-bit 2’s complement notation? What is –31 in 8-bit 2’s complement notation? Show how to compute 14–31 using 2’s complement addition. What is the result in 8-bit 2’s complement notation?

C. (4 points): What are 14 and 31 in 8-bit 2’s complement notation? What is –31 in 8-bit 2’s complement notation? Show how to compute 14–31 using 2’s complement addition. What is the result in 8-bit 2’s complement notation?

- 14 = **0b0000_1110**
- 31 = **0b0001_1111**
- -31 = **0b1110_0001**

→ **to negate: -A = ~A + 1**

14 // 2 = 7 R 0 LSB
7 // 2 = 3 R 1
3 // 2 = 1 R 1
1 // 2 = 0 R 1

C. (4 points): What are 14 and 31 in 8-bit 2’s complement notation? What is –31 in 8-bit 2’s complement notation? Show how to compute 14–31 using 2’s complement addition. What is the result in 8-bit 2’s complement notation?

- 14 = **0b0000_1110**
- 31 = **0b0001_1111**
- -31 = **0b1110_0001**

- 0b0000_1110
- 0b1110_0001
- **0b1110_1111** = -17

D. How many bits are required to encode decimal values ranging from -128 to 127 in two's complement representation? How many bits are required to encode decimal values ranging from 0 to 127 in unsigned binary representation? Provide your answer in decimal.

D. How many bits are required to encode decimal values ranging from -128 to 127 in two's complement representation? How many bits are required to encode decimal values ranging from 0 to 127 in unsigned binary representation? Provide your answer in decimal.

Two's complement:

- Range of -128 → 127 is **128 + 127 + 1 (we need to include 0) =** 256 values that we need to represent
- 2^(8 bits) = 256 values can be represented using 8 bits
- **Ans: 8 bits**

Unsigned binary representation:

- Range of 0 → 127 is **128 (includes zero)** values that we need to represent
- 2^(7 bits) = 128 values can be represented using 7 bits
- **Ans: 7 bits**

D. How many bits are required to encode decimal values ranging from -128 to 127 in two's complement representation? How many bits are required to encode decimal values ranging from 0 to 127 in unsigned binary representation? Provide your answer in decimal.

- Two's complement range: $[-2^{n-1}, 2^{n-1}-1]$
- Unsigned range: $[0, 2^{n}-1]$
- Where n is the number of bits

Two's complement

- $127 = 2^{n-1}-1$
- $128 = 2^{n-1}$
- $\log_2 128 = n-1$
- $7 = n-1$
- **$8 = n$**

Unsigned

- $127 = 2^{n}-1$
- $128 = 2^{n}$
- $\log_2 128 = n$
- **$7 = n$**

E. (2 points) What is the result of the logical right shift 0b11011010 >> 2 in 2's complement notation? What is the result of the arithmetic right shift 0b11011010 >> 2 in 2's complement notation? Provide your answer in binary

E. (2 points) What is the result of the logical right shift 0b11011010 >> 2 in 2's complement notation? What is the result of the arithmetic right shift 0b11011010 >> 2 in 2's complement notation? Provide your answer in binary

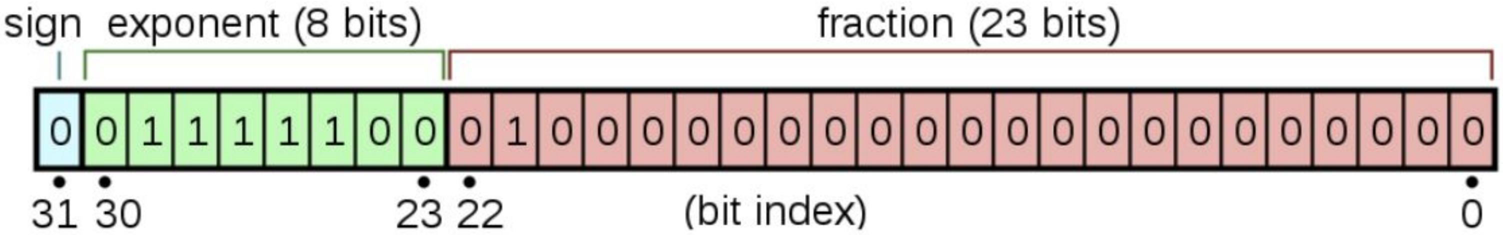
- Logical: right shift in zeros
- Arithmetic: right shift in value of MSB
  - To preserve the sign of the value

E. (2 points) What is the result of the logical right shift 0b11011010 >> 2 in 2's complement notation? What is the result of the arithmetic right shift 0b11011010 >> 2 in 2's complement notation? Provide your answer in binary

- Logical: right shift in zeros
- Arithmetic: right shift in value of MSB
  - To preserve the sign of the value

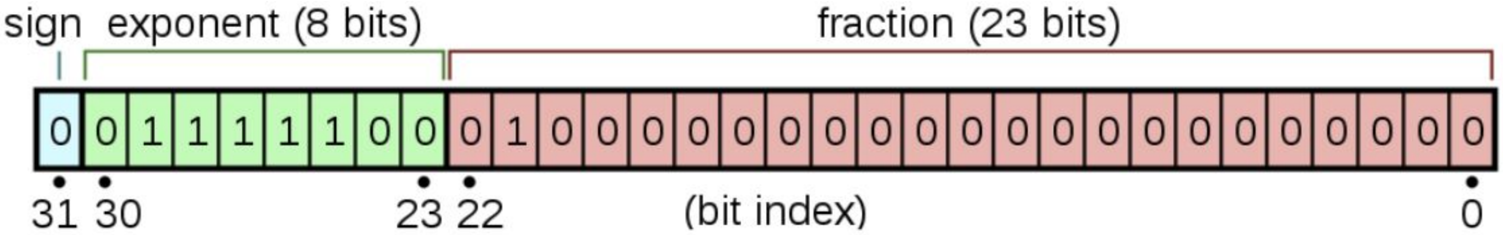
- Logical: `0b1101_1010 >> 2` = `0b0011_0110`
- Arithmetic: `0b1101_1010 >> 2` = `0b1111_0110`

G. What is the decimal equivalent of the 32-bit floating point number 0x41080000? The format of 32-bit floating point encoding is shown below. Show your work



$$\text{Value} = (-1)^{\text{sign}} \cdot 2^{\text{exp}-127} \cdot \left(1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}\right)$$

G. What is the decimal equivalent of the 32-bit floating point number 0x41080000? The format of 32-bit floating point encoding is shown below. Show your work



$$\text{Value} = (-1)^{\text{sign}} \cdot 2^{\text{exp}-127} \cdot \left(1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}\right)$$

0x41080000 = 0100_0001_0000_1000_0000_0000_0000_0000

b_19 → b_(23-4), so i == 4

$(-1)^0 * 2^{130 - 127} * (1 + 2^{-4}) = 2^3 * 1.0625 =$ **8.5**

Problem 2

What If

```c
// Given a string, flip the case of each alphabetical character.
void flip_case(char *x) {
   while (*x != 0) {
       if (is_uppercase(*x)) {
           *x += 'a' - 'A';  // e.g., 'G' becomes 'g'
       } else if (is_lowercase(*x)) {
           *x += 'A' - 'a';  // e.g., 'g' becomes 'G'
       }
       x++; // <- For part A and B
   }
}
```

A. Which candidates are equivalent to x++; in the above program?

Candidate	CIRCLE ONE:
`&x = &x + 1;`	
`*(&x) = x + 1;`	
`x = (char *)((uint32_t)x + 4);`	

```
&x = &x + 1;
```

A. Which candidates are equivalent to x++; in the above program?

Candidate	CIRCLE ONE:
`&x = &x + 1;`	YES NO
`*(&x) = x + 1;`	
`x = (char *)((uint32_t)x + 4);`	

Address of x Increment by 1

```
&x = &x + 1;
```

Address of x

Address of X = Address of X + 1

Doesn't work!
X is a pointer! Incrementing the address of the pointer is not the same thing as incrementing the pointer!

A. Which candidates are equivalent to x++; in the above program?

Candidate	CIRCLE ONE:
`&x = &x + 1;`	YES NO
`*(&x) = x + 1;`	
`x = (char *)((uint32_t)x + 4);`	

```
*(&x) = x + 1;
```

A. Which candidates are equivalent to x++; in the above program?

Candidate	CIRCLE ONE:	
`&x = &x + 1;`	YES	NO
`*(&x) = x + 1;`	YES	NO
`x = (char *)((uint32_t)x + 4);`		

increment x by one

```
*(&x) = x + 1;
```

same thing as x

*(&x) = x

Works just fine!
Obtaining the address of x, and then dereferencing that is just the same thing as writing down x.

A. Which candidates are equivalent to x++; in the above program?

Candidate	CIRCLE ONE:	
`&x = &x + 1;`	YES	NO
`*(&x) = x + 1;`	YES	NO
`x = (char *)((uint32_t)x + 4);`		

```
x = (char *)((uint32_t)x + 4);
```

A. Which candidates are equivalent to x++; in the above program?

Candidate	CIRCLE ONE:	
`&x = &x + 1;`	YES	NO
`*(&x) = x + 1;`	YES	NO
`x = (char *)((uint32_t)x + 4);`	YES	NO

Cast address into 32-bit number

```
x = (char *)((uint32_t)x + 4);
```

Cast back into a char pointer

Add 4 bytes to it

Doesn't work!
The idea was good but the execution wasn't. Chars are 1 byte wide, so we should have added 1 instead of 4.

B. Suppose we are instead to replace x++; in the implementation of flip_case with f(&x); and implement f as follows.

```
void f(__1__ x) {
    __2__++;
}
```

Blank 1:	Blank 2:

Things to note:

- Argument to function f() is a reference to x, since we pass in &x (the address of x).
- X is a char pointer (char *).

B. Suppose we are instead to replace x++; in the implementation of flip_case with f(&x); and implement f as follows.

```
void f(__1__ x) {
    __2__++;
}
```

Blank 1:	Blank 2:
char**	

Make the argument type a reference to a char*!

A pointer to a char pointer!

B. Suppose we are instead to replace x++; in the implementation of flip_case with f(&x); and implement f as follows.

```
void f(__1__ x) {
    __2__++;
}
```

Blank 1:	Blank 2:
char**	(*x)

Make the argument type a reference to a char*!
A pointer to a char pointer!

Increment x, not the pointer to x!
Dereference x before incrementing, but be careful of operator precedence. (++ occurs before *).

C. For each expression, determine the string that the program on the previous page would print if we were to replace ??? with that expression. If the code would not compile, write "WON'T COMPILE". If the exact output cannot be determined, write "CAN'T TELL".

```
int main() {
    char str[] = "ababABAB";
    char *p = str;   // char * that points to str
    char **q = &p;   // char ** that points to p
    flip_case(???); // <- REPLACE HERE
    printf("%s\n", str);
    return 0;
}
```

Here is flip_case again:

```
// Given a string, flip the case of each alphabetical character.
void flip_case(char *x) {
    while (*x != 0) {
        if (is_uppercase(*x)) {
            *x += 'a' - 'A';  // e.g., 'G' becomes 'g'
        } else if (is_lowercase(*x)) {
            *x += 'A' - 'a';  // e.g., 'g' becomes 'G'
        }
        x++; // <- For part A and B
    }
}
```

C. For each expression, determine the string that the program on the previous page would print if we were to replace ??? with that expression. If the code would not compile, write "WON'T COMPILE". If the exact output cannot be determined, write "CAN'T TELL".

Expression	**Answer**
`p`	
`str + 2`	
`(*q) + 8`	
`&(*(&p))[4] - 1`	

C. For each expression, determine the string that the program on the previous page would print if we were to replace ??? with that expression. If the code would not compile, write “WON’T COMPILE”. If the exact output cannot be determined, write “CAN’T TELL”.

Expression	**Answer**
`p`	ABABabab
`str + 2`	
`(*q) + 8`	
`&(*(&p))[4] - 1`	

Passing p should just flip the case of "ababABAB".

C. For each expression, determine the string that the program on the previous page would print if we were to replace ??? with that expression. If the code would not compile, write "WON'T COMPILE". If the exact output cannot be determined, write "CAN'T TELL".

Expression	Answer
`p`	ABABabab
`str + 2`	abABabab
`(*q) + 8`	
`&(*(&p))[4] - 1`	

Passing str+2 should just flip the case of the last 6 chars of str. Offsetting by 2 skips the first 2 chars.

C. For each expression, determine the string that the program on the previous page would print if we were to replace ??? with that expression. If the code would not compile, write "WON'T COMPILE". If the exact output cannot be determined, write "CAN'T TELL".

Expression	Answer
`p`	ABABabab
`str + 2`	abABabab
`(*q) + 8`	ababABAB
`&(*(&p))[4] - 1`	

Passing (*q) + 8 should avoid flipping anything. We've skipped all 8 chars. (*q) is the same as str, and offsetting by 8 moves the str up to the null char.

C. For each expression, determine the string that the program on the previous page would print if we were to replace ??? with that expression. If the code would not compile, write “WON’T COMPILE”. If the exact output cannot be determined, write “CAN’T TELL”.

Expression	**Answer**
`p`	ABABabab
`str + 2`	abABabab
`(*q) + 8`	ababABAB
`&(*(&p))[4] - 1`	abaBabab

***(&p) is just p again. &(p)[4] offsets p by four, and subtracting one brings the offset to 3. This skips the first three characters when flipping.**

Problem 3

C structs

Problem 3. C structs (13 points)

When communicating with our RISC-V microcontroller, we usually did so through the serial monitor embedded in our IDE. A struct called `SerialBuffer` is defined below to represent our controller's serial communication buffer, which is responsible for handling incoming Serial messages.

```
struct SerialBuffer{
  char termChar;          // char to stop reading at
  char charBuffer[64];    // 64 element wide buffer to store characters in
  uint8_t size;           // number of chars stored in buffer
};
```

A. (3 points) First let's model how the buffer receives data. An example of an empty `SerialBuffer`, with the newline as the terminating character, is shown below:

```
struct SerialBuffer buf;
buf.termChar = '\n';
buf.size = 0;
```

When the buffer receives a character, the buffer adds the char to the `charBuffer` array at the smallest available index. It then also increments the active buffer size count by one.

Write a function **`receiveChar`** that takes in a `SerialBuffer` struct (by value) and a character to add to the buffer and returns an updated `SerialBuffer` instance.
Assume the buffer has enough space for an incoming character, `c`.

```
struct SerialBuffer receiveChar(struct SerialBuffer buf, char c){


}
```

```
struct SerialBuffer receiveChar(struct SerialBuffer buf, char c){
```

```
struct SerialBuffer receiveChar(struct SerialBuffer buf, char c){




  // one possible answer
  buf.charBuffer[buf.size++] = c;
  return buf;
```

B. (4 points): Sometimes we want to peek into our serial buffer without removing any characters from the buffer. Write a function called **`peekChars`** that receives **a pointer to a buffer instance** and returns how many characters are in the buffer **up to and including** the first termination character (reflected in `termChar`). Assume the buffer contains at least one terminating character.

```
int peekChars(struct SerialBuffer *buf){








}
```

B. (4 points): Sometimes we want to peek into our serial buffer without removing any characters from the buffer. Write a function called **`peekChars`** that receives **a pointer to a buffer instance** and returns how many characters are in the buffer **up to and including** the first termination character (reflected in `termChar`). Assume the buffer contains at least one terminating character.

```
int peekChars(struct SerialBuffer *buf){

     // one possible answer
     int count = 0;
     while (buf->charBuffer[count] != buf->termChar){
       count++;
     }
     count++;
     return count;

}
```

C. (6 points): Consider the case where we want to read from an active `SerialBuffer` instance.

```
struct SerialBuffer buf;
```

This buffer has already been populated with multiple characters and at least one terminating character.

Create a function, **`readChars`**, that reads characters from the buffer **up to and including the buffer's termination character**. Store this string of characters as a properly terminated C-string in `char *message`, which you can assume will be large enough to store the resulting message.

Note that `readChars` is passed a **pointer** to a `SerialBuffer`.
Be sure to leverage the **`peekChars`** function you just wrote.

```
void readChars(struct SerialBuffer *buf, char *message){
```

Be sure to update the buffer by both updating the `charBuffer` array and the buffer size.
As an example:

```
// up to this point buf has been populated and contains:
// buf.charBuffer = {'h','e','l','l','o','\n','b','y','e','\n', ...}
```

```
printf("%c", buf.termChar); // prints: "\n"
printf("%d", buf.size);     // prints: "10"

char msg[65];               // large enough to store message from buffer
readChars(&buf, msg);       // move chars up to termChar from buffer to msg

printf("%s", msg);          // prints: "hello\n"
printf("%d", buf.size);     // prints: "4"

// now at this point buf.charBuffer = {'b','y','e','\n', ...}
```

```
void readChars(struct SerialBuffer *buf, char *message){

}
```

```
void readChars(struct SerialBuffer *buf, char *message){

 // a possible answer

  int count = peekChars(buf);

  //load bytes onto message
  for (int i = 0; i < count; i++){
    *(message + i) = buf->charBuffer[i];
  }

  //terminate message
  *(message + count) = 0;

  //update buffer
  for (int i = count; i < buf->size; i++){
     buf->charBuffer[i-count] = buf->charBuffer[i];
  }
  buf->size -= count;

}
```

Problem 4

An Average Filter

Fill in the blanks.

```
void find_mean(const float *arr, int n, float *mean); // Defined elsewhere.

void mean_filter(const float *input, int num_elems, int window_size, float *output) {
   for (int i = 0; i+window_size-1 < num_elems; i++) {
      float buffer;
      float *ptr = &buffer;
      find_mean(___(A)___, ___(B)___, ___(C)___);
      *(output+i) = ___(D)___;
   }
}

// For example, if num_elems=4, input={3, 2, 7, 6}, window_size=3, then
// there are two contiguous windows, each with the following arithmetic means:
// output[0] = (3+2+7)/3.0 = 4.0
// output[1] = (2+7+6)/3.0 = 5.0
```

Fill in the blanks.

```
void find_mean(const float *arr, int n, float *mean); // Defined elsewhere.

void mean_filter(const float *input, int num_elems, int window_size, float *output) {
   for (int i = 0; i+window_size-1 < num_elems; i++) {
      float buffer;
      float *ptr = &buffer;
      find_mean(___(A)___, ___(B)___, ___(C)___);
      *(output+i) = ___(D)___;
   }
}

// For example, if num_elems=4, input={3, 2, 7, 6}, window_size=3, then
// there are two contiguous windows, each with the following arithmetic means:
// output[0] = (3+2+7)/3.0 = 4.0
// output[1] = (2+7+6)/3.0 = 5.0
```

find_mean() averages `n` elements in an float array `arr` and returns the mean in float pointer `mean`.

Fill in the blanks.

Fill in the blank (A):
Fill in the blank (B):
Fill in the blank (C):
Circle ALL correct answers (D):

Fill in the blanks.

Fill in the blank (A): `input+i` or `&input[i]`
Fill in the blank (B):
Fill in the blank (C):
Circle ALL correct answers (D):

As we iterate over the elements in the float array `input`, we need to offset the array being passed into `find_mean()`.

Fill in the blanks.

Fill in the blank (A): `input+i` or `&input[i]`
Fill in the blank (B): `window_size`
Fill in the blank (C):
Circle ALL correct answers (D):

We only want to calculate the mean over `window_size` arguments!

Fill in the blanks.

Fill in the blank (A): `input+i` or `&input[i]`
Fill in the blank (B): `window_size`
Fill in the blank (C): `ptr` or `&buffer`
Circle ALL correct answers (D):

There are multiple ways of passing along a reference to the float `buffer` that will contain the result of our averaging.

Fill in the blanks.

Fill in the blank (A): `input+i` or `&input[i]`
Fill in the blank (B): `window_size`
Fill in the blank (C): `ptr` or `&buffer`

Circle ALL correct answers (D):
`buffer` `*buffer` `&buffer`
`ptr` `*ptr` `&ptr`
`ptr[0]` `*ptr[0]` `&ptr[0]`

Multiple ways of updating the array! We just want to store the value of `buffer` in the `output` array. Directly storing `buffer` or dereferencing `ptr` work.

Problem 5

Assembly Language

A. What is hexadecimal encoding of the instruction `srai t3, a2, 6`? You can use the template below to help you with the encoding.

[31:25]	[24:20]	[19:15]	[14:12]	[11:7]	[6:0]
0100000	shamt	rs1	funct3	rd	opcode

SRAI	`srai rd, rs1, shamt`

0100000	shamt	rs1	101	rd	0010011	SRAI

Registers	Symbolic names
x0	zero
x1	ra
x2	sp
x3	gp
x4	tp
x5-x7	t0-t2
x8-x9	s0-s1
x10-x11	a0-a1
x12-x17	a2-a7
x18-x27	s2-s11
x28-x31	t3-t6

A. What is hexadecimal encoding of the instruction `srai t3, a2, 6`? You can use the template below to help you with the encoding.

[31:25]	[24:20]	[19:15]	[14:12]	[11:7]	[6:0]
0100000	shamt	rs1	funct3	rd	opcode

SRAI	`srai rd, rs1, shamt`

0100000	shamt	rs1	101	rd	0010011	SRAI

- shamt = 6 = 0b00110
- rs1 = a2 = x12 = 12 = 0b01100
- rd = t3 = x28 = 28 = 0b11100
- funct3 = 0b101
- Opcode = 0b0010011
- 0100000_00110_01100_101_11100_0010011
- 0x40665E13

Registers	Symbolic names
x0	zero
x1	ra
x2	sp
x3	gp
x4	tp
x5-x7	t0-t2
x8-x9	s0-s1
x10-x11	a0-a1
x12-x17	a2-a7
x18-x27	s2-s11
x28-x31	t3-t6

B. provide the hexadecimal values of the specified registers after each sequence has been executed. Assume that each sequence execution ends when it reaches the `end` label

```
. = 0x20
      li x11, 0x600
      lw x11, 0x0(x11)
      bge x11, x0, L1
      xori x12, x11, 0xA55
      j end

L1:  srli x12, x11, 8
end:

. = 0x600
X:   .word 0xC0C0A0A0
```

B. provide the hexadecimal values of the specified registers after each sequence has been executed. Assume that each sequence execution ends when it reaches the `end` label

- x11 = 0x600
- x11 = 0xC0C0A0A0
- x11 MSB is 1, so negative
  - We don't branch
- x12 = 0xC0C0A0A0 ^ 0xA55
- Don't forget to sign extend
- 0xC0C0A0A0
- 0xFFFFFA55
- 0x3F3f5af5

XOR Truth Table

Input 1	Input 2	Output
0	1	1
0	0	0
1	1	0
1	0	1

```
. = 0x20
      li x11, 0x600
      lw x11, 0x0(x11)
      bge x11, x0, L1
      xori x12, x11, 0xA55
      j end

L1:  srli x12, x11, 8
end:

. = 0x600
X:    .word 0xC0C0A0A0
```

For the RISC-V instruction sequences below, provide the hexadecimal values of the specified registers after each sequence has been executed. Assume that each sequence execution ends when it reaches the `end` label. Also assume that all registers are initialized to 0 before execution of each sequence begins.

The first instruction executed is located at address 0x100

```
. = 0x20
f:
      slli x13, x12, 8
      ret

. = 0x100
      lui x11, 0x3
      lw x12, 0x4(x11)
      jal x1, f
      ori x14, x1, 0xC2
end:

. = 0x3000
.word 0x11112222
.word 0x22224444
.word 0x33336666
```

Value left in x1: 0x____________________

Value left in x11: 0x____________________

Value left in x12: 0x____________________

Value left in x13: 0x____________________

Value left in x14: 0x____________________

For the RISC-V instruction sequences below, provide the hexadecimal values of the specified registers after each sequence has been executed. Assume that each sequence execution ends when it reaches the `end` label. Also assume that all registers are initialized to 0 before execution of each sequence begins.

- Starting at 0x100, `x11` becomes 0x3000 since `lui` shifts the immediate by 12 and then sets the register to that result
- `x12` = 0x22224444 since `lw x12, 0x4(x11)` loads the value at address 0x3004
- `jal x1, f` unconditionally jumps to the `f` label and executes the code there
  - `x1` gets set to the address of the `jal` instruction + 4 = 0x10C
    - Every instruction is 4 bytes
  - `x1` is the `ra` register
- `x13` = 0x22444400
- `ret` makes the program jump back to the address stored in `ra` which is also `x1`
- `x14` = 0xC2 | 0x10C = 0x1CE

The first instruction executed is located at address `0x100`.

```
. = 0x20
f:
      slli x13, x12, 8
      ret

. = 0x100
      lui x11, 0x3
      lw x12, 0x4(x11)
      jal x1, f
      ori x14, x1, 0xC2
end:

. = 0x3000
.word 0x11112222
.word 0x22224444
.word 0x33336666
```

Value left in x1: 0x______10C__________

Value left in x11: 0x______3000__________

Value left in x12: 0x_____22224444_______

Value left in x13: 0x_____22444400_______

Value left in x14: 0x_____000001CE______

Problem 6

Calling Convention

Calling a function

To call fn, use:

- call fn
- jal ra, fn
- jal fn

Two things happen:

- reg[ra] <= reg[pc] + 4
- then, pc becomes the address of fn

To return, use:

- ret
- jalr x0, 0(ra)

Only one thing happens:

- reg[pc] <= reg[ra]

Calling convention

Arguments in a0 - a7 (x10 - x17)

Return values in a0 - a1 (x10 - x11)

Caller-saved registers: a, t, ra

- When you call a function, these registers may lose their original values
- Store them before you call a function *if you will need them later*

Callee-saved registers: s, sp

- When you call a function, these registers will not lose their original values
- To use them, make sure to fulfill your responsibility as a callee too

```
# drawBoard Arguments:
# (1) screen_buffer
# (2) locations: array holding locations of snake segments on board
# (3) num_locations: length of locations array.
# (4) food: location of food
```

```
drawBoard:
      slli a2, a2, 2
      add a2, a2, a1
      mv s0, a0
loop:
      bge a1, a2, end
      mv a0, s0
      lw a1, 0(a1)
      li a2, 1
      call setPixel
      addi a1, a1, 4
      j loop
end:
      mv a0, s0
      mv a1, a3
      li a2, 1
      call setPixel
      ret
```

You decided to write Snake in RISC-V assembly. You implement a `drawBoard` function to render the game board. `drawBoard` uses one helper function, `setPixel`. to set a given pixel to be 0 (off) or 1 (on). It's C function signature is shown below:

```
void setPixel(uint32_t *screen_buffer, uint8_t location, uint8_t val);
```

You can assume that setPixel works as expected and follows calling convention. You do not have access to the assembly implementation of `setPixel`, so you cannot make any further assumptions about its implementation.

Unfortunately, your program does not work, and you suspect that it is due to calling convention. Please add appropriate instructions (**either increment/decrement stack pointer, load word from stack, or save word to stack only**) into the blank spaces on the right to make `drawBoard` follow calling convention. You can assume that `drawBoard` will work as expected once it follows the calling convention.

If the procedure already follows calling convention, write NO INSTRUCTIONS NEEDED. For full credit, you should only save registers that must be saved onto the stack, restore registers that must be restored, and minimize the number of instructions used. You may not need to use all the blank lines.

Calling convention refresher

```
drawBoard:
        slli a2, a2, 2
        add a2, a2, a1
        mv s0, a0
loop:
        bge a1, a2, end
        mv a0, s0
        lw a1, 0(a1)
        li a2, 1
        call setPixel
        addi a1, a1, 4
        j loop
end:
        mv a0, s0
        mv a1, a3
        li a2, 1
        call setPixel
        ret
```

- Since we are calling another procedure, we must store `ra` before the **first** `call` instruction and load it back before we `ret`
  - Only need to store `ra` **once**, no matter how many procedures are called
  - `drawBoard` needs the original `ra` value so `ret` can return to the correct address
- `s` registers are **callee** saved. We must store their values **before** we, as a callee, use them. We then load their original values right before we `ret`.
  - This is why `s` register values persist between procedure calls
- `a` registers are **caller** saved. If we call other procedures and these registers have values we want to use after, we must store them to then load back after.
  - `a` and `t` registers are not guaranteed to stay the same between calls
  - We load them back every time we need that stored value

```
# drawBoard Arguments:
# (1) screen_buffer
# (2) locations: array holding locations of snake segments on board
# (3) num_locations: length of locations array.
# (4) food: location of food
```

```
drawBoard:
        slli a2, a2, 2
        add a2, a2, a1
        mv s0, a0
loop:
        bge a1, a2, end
        mv a0, s0
        lw a1, 0(a1)
        li a2, 1
        call setPixel
        addi a1, a1, 4
        j loop
end:
        mv a0, s0
        mv a1, a3
        li a2, 1
        call setPixel
        ret
```

- Allocate enough space on the stack
- Store `ra` because we call other procedures
- Store `a3` because we use its value here also after a `call setPixel`
- Store `s0` because we will be using it (overwriting it with our own value)
- Store `a2` since we care about its value **after** we set it with `slli` and `add`
    - And we need it for our branch condition after a potential `call setPixel` from looping

```
drawBoard:

        ___addi sp, sp, -20_______
        ____sw ra, 0(sp)__________
        _____sw a3, 8(sp)_________
        ______sw s0, 12(sp)_______
        __________________________
        __________________________

        slli a2, a2, 2
        add a2, a2, a1
        mv s0, a0

        ______sw a2, 4(sp)________
        __________________________
        __________________________
```

```
# drawBoard Arguments:
# (1) screen_buffer
# (2) locations: array holding locations of snake segments on board
# (3) num_locations: length of locations array.
# (4) food: location of food
```

```
drawBoard:
        slli a2, a2, 2
        add a2, a2, a1
        mv s0, a0
loop:
        bge a1, a2, end
        mv a0, s0
        lw a1, 0(a1)
        li a2, 1
        call setPixel
        addi a1, a1, 4
        j loop
end:
        mv a0, s0
        mv a1, a3
        li a2, 1
        call setPixel
        ret
```

- We store `a1` because we will be using this same value to branch
- Load `a1` and `a2` since their values could have changed with `call setPixel`
  - And we use them for our branch instruction

```
loop:
        bge a1, a2, end

        ______________________

        _______sw a1, 16(sp)_______

        ______________________

        ______________________

        ______________________

        ______________________

        mv a0, s0
        lw a1, 0(a1)
        li a2, 1
        call setPixel

        _______lw a1, 16(sp)_______

        _______lw a2, 4(sp)________

        addi a1, a1, 4
        j loop
```

```
# drawBoard Arguments:
# (1) screen_buffer
# (2) locations: array holding locations of snake segments on board
# (3) num_locations: length of locations array.
# (4) food: location of food
```

```
drawBoard:
        slli a2, a2, 2
        add a2, a2, a1
        mv s0, a0
loop:
        bge a1, a2, end
        mv a0, s0
        lw a1, 0(a1)
        li a2, 1
        call setPixel
        addi a1, a1, 4
        j loop
end:
        mv a0, s0
        mv a1, a3
        li a2, 1
        call setPixel
        ret
```

- We load `a3` since we want to use its original value
- At the end, we load back `s0` and `ra` to get their original values
  - `ra` is used to return to the proper address after the procedure is done
  - `s0` needs to keep its original value after we use it
- Don't forget to increment `sp` since we are no longer use that stack space
- `a1`, `a2`, and `a3` are never guaranteed to be the values they started as, so we don't need to load them

```
end:
    _____lw a3, 8(sp)__________
    mv a0, s0
    mv a1, a3
    li a2, 1
    call setPixel

    ___________________________

    ___________________________

    ___________________________

    _____lw s0, 12(sp)_________

    _____lw ra, 0(sp)__________

    _____addi sp, sp, 20_______

    ret
```

Problem 7

Stack Detective

Problem 7. Stack Detective (14 points)

Consider the following C function which takes an array of unsigned 32 bit integers `a` of length `b` and computes their product. We don't have a multiply instruction in our RV32I system, so we use the `mult` procedure (which you used in class, provided in appendix for reference) in order to actually do the multiplication:

```
int arrayProd(uint32_t* a, uint32_t b){
   // uint32_t *a: pointer to array
   // uint32_t b: length of array
   if (b == 1) {
       return a[0];
   }else {
       // multiply both numbers:
       return mult(arrayProd(a+1, b-1), a[0]);
   }
}
```

The equivalent assembly procedure for this function is below:

```
arrayProd:
   lw t0, 0(a0)
   li a3, 1
   beq a1, a3, end
   addi sp, sp, -8
   sw ra, 0(sp)
   sw t0, 4(sp)
   # SAMPLE POINT - prints ra, sp, a0, a1, a part of the stack
   addi a0, a0, 4
   addi a1, a1, -1
   jal arrayProd
   lw a1, 4(sp)
   jal mult  # returns product of a0 and a1 (see appendix)
   ______________
   lw ra, 0(sp)
   addi sp, sp, 8
end:
   mv a0, t0
   ret
```

Note the sample point line above. When this line is encountered, the four registers `ra`, `sp`, `a0`, and `a1` are printed as well as a region of the stack.

A. What line of assembly should be substituted into the blank line in the `arrayProd` procedure above?

```
arrayProd:
   lw t0, 0(a0)
   li a3, 1
   beq a1, a3, end
   addi sp, sp, -8
   sw ra, 0(sp)
   sw t0, 4(sp)
   # SAMPLE POINT - prints ra, sp, a0, a1, a part of the stack
   addi a0, a0, 4
   addi a1, a1, -1
   jal arrayProd
   lw a1, 4(sp)
   jal mult  # returns product of a0 and a1 (see appendix)
   ______________
   lw ra, 0(sp)
   addi sp, sp, 8
end:
   mv a0, t0
   ret
```

A. What line of assembly should be substituted into the blank line in the `arrayProd` procedure above?

```
arrayProd:
    lw t0, 0(a0)       t0 = a[0]
    li a3, 1           a3 = 1
    beq a1, a3, end    If b == 1 go to end
    addi sp, sp, -8
    sw ra, 0(sp)       save ra, t0 == a[0] to stack
    sw t0, 4(sp)
    # SAMPLE POINT - prints ra, sp, a0, a1, a part of the stack
    addi a0, a0, 4     a0 = a +1, go to the next address in the int array a. An int is 4 bytes so add 4
    addi a1, a1, -1    a1 = b - 1
    jal arrayProd      Recursive call, inputs a+1 and b-1
    lw a1, 4(sp)       a1 = t0
    jal mult  # returns product of a0 and a1 (see appendix)   a0 = whatever arrayProd returns
    ______________
    lw ra, 0(sp)       reload ra and reset stack pointer
    addi sp, sp, 8
end:
    mv a0, t0          Result is in t0, put it in return register… how did the result get into t0?
    ret
```

A. What line of assembly should be substituted into the blank line in the `arrayProd` procedure above?

Ans: mv t0, a0

- Our answer from the `mult` procedure call is in a0.
- t0 is not guaranteed to be known after the call
- Before we `ret,` we move t0 into a0
  - So we must ensure t0 is also the value we are returning

B. A user creates an array and passes it and its length into `arrayProd`. Immediately prior to and immediately after the procedure call, a sample of the ra, sp, a0, and a1 is collected as well as a region of the stack.

B. A user creates an array and passes it and its length into `arrayProd`. Immediately prior to and immediately after the procedure call, a sample of the ra, sp, a0, and a1 is collected as well as a region of the stack.

```
#1   sp   =0x00080280 ra   =0x00000000
     a0   =0x00004000 a1   =0x00000005
Address: Data:
0x80258: 0x000ff3af
0x8025c: 0x00000018
0x80260: 0x0000035c
0x80264: 0x00000011
0x80268: 0x00000008
0x8026c: 0x00000001
0x80270: 0x0000035c
0x80274: 0x00000011
0x80278: 0x00000808
0x8027c: 0x0000a321
0x80280: 0x00000781
```

Before call to arrayProd

```
#2   sp   =0x00080278 ra   =0x00000204
     a0   =0x00004000 a1   =0x00000005
Address: Data:
0x80258: 0x000ff3af
0x8025c: 0x00000018
0x80260: 0x0000035c
0x80264: 0x00000011
0x80268: 0x00000008
0x8026c: 0x00000001
0x80270: 0x0000035c
0x80274: 0x00000011
0x80278: 0x00000204
0x8027c: 0x00000001
0x80280: 0x00000781
```

```
#3   sp   =0x00080270 ra   =0x0000025C
     a0   =0x00004004 a1   =0x00000004
Address: Data:
0x80258: 0x000ff3af
0x8025c: 0x00000018
0x80260: 0x0000035c
0x80264: 0x00000011
0x80268: 0x00000008
0x8026c: 0x00000001
0x80270: 0x0000025c
0x80274: 0x00000003
0x80278: 0x00000204
0x8027c: 0x00000001
0x80280: 0x00000781
```

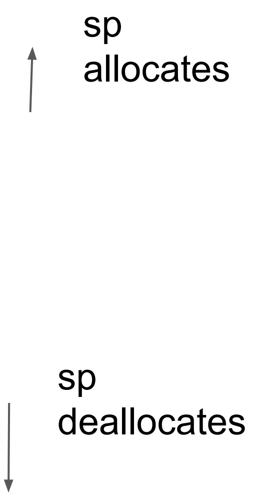
```
#4   sp   =0x00080268 ra   =0x0000025C
     a0   =0x00004008 a1   =0x00000003
Address: Data:
0x80258: 0x000ff3af
0x8025c: 0x00000018
0x80260: 0x0000035c
0x80264: 0x00000011
0x80268: 0x0000025c
0x8026c: 0x00000005
0x80270: 0x0000025c
0x80274: 0x00000003
0x80278: 0x00000204
0x8027c: 0x00000001
0x80280: 0x00000781
```

```
#5   sp   =0x00080260 ra   =0x0000025C
     a0   =0x0000400C a1   =0x00000002
Address: Data:
0x80258: 0x000ff3af
0x8025c: 0x00000018
0x80260: 0x0000025c
0x80264: 0x00000008
0x80268: 0x0000025c
0x8026c: 0x00000005
0x80270: 0x0000025c
0x80274: 0x00000003
0x80278: 0x00000204
0x8027c: 0x00000001
0x80280: 0x00000781
```

```
#6   sp   =0x00080280 ra   =0x00000204
     a0   =0x000000F0 a1   =0x00000000
Address: Data:
0x80258: 0x000ff3af
0x8025c: 0x00000018
0x80260: 0x0000025c
0x80264: 0x00000008
0x80268: 0x0000025c
0x8026c: 0x00000005
0x80270: 0x0000025c
0x80274: 0x00000003
0x80278: 0x00000204
0x8027c: 0x00000001
0x80280: 0x00000781
```

A. What line of assembly should be substituted into the blank line in the `arrayProd` procedure above?

```
arrayProd:
   lw t0, 0(a0)
   li a3, 1
   beq a1, a3, end
   addi sp, sp, -8
   sw ra, 0(sp)
   sw t0, 4(sp)
   # SAMPLE POINT – prints ra, sp, a0, a1, a part of the stack
   addi a0, a0, 4
   addi a1, a1, -1
   jal arrayProd
   lw a1, 4(sp)
   jal mult  # returns product of a0 and a1 (see appendix)
   ______________
   lw ra, 0(sp)
   addi sp, sp, 8
end:
   mv a0, t0
   ret
```



B1) What is the hexadecimal address of the instruction that originally calls `arrayProd`?

B1) What is the hexadecimal address of the instruction that originally calls `arrayProd`?

We know when arrayProd is initially called, ra == call instruction + 4

```
arrayProd:
    lw t0, 0(a0)
    li a3, 1
    beq a1, a3, end
    addi sp, sp, -8
    sw ra, 0(sp)
    sw t0, 4(sp)
    # SAMPLE POINT - prints ra, sp, a0, a1, a part of the stack
    addi a0, a0, 4
    addi a1, a1, -1
    jal arrayProd
    lw a1, 4(sp)
    jal mult  # returns product of a0 and a1 (see appendix)
    ______________
    lw ra, 0(sp)
    addi sp, sp, 8
end:
    mv a0, t0
    ret
```

On the initial run ra would be saved at the address the stack pointer is at

And a snapshot would be taken

This means we should look at snapshot 2!

B1) What is the hexadecimal address of the instruction that originally calls `arrayProd`?

- We save ra to the stack multiple times, the first ra is the initial call arrayProd instruction + 4

```
#1  sp  =0x00080280 ra  =0x00000000
    a0  =0x00004000 a1  =0x00000005
Address: Data:
0x80258: 0x000ff3af
0x8025c: 0x00000018
0x80260: 0x0000035c
0x80264: 0x00000011
0x80268: 0x00000008
0x8026c: 0x00000001
0x80270: 0x0000035c
0x80274: 0x00000011
0x80278: 0x00000808
0x8027c: 0x0000a321
0x80280: 0x00000781
```

Before call to arrayProd

```
#2  sp  =0x00080278 ra  =0x00000204
    a0  =0x00004000 a1  =0x00000005
Address: Data:
0x80258: 0x000ff3af
0x8025c: 0x00000018
0x80260: 0x0000035c
0x80264: 0x00000011
0x80268: 0x00000008
0x8026c: 0x00000001
0x80270: 0x0000035c
0x80274: 0x00000011
0x80278: 0x00000204
0x8027c: 0x00000001
0x80280: 0x00000781
```

```
#3  sp  =0x00080270 ra  =0x0000025C
    a0  =0x00004004 a1  =0x00000004
Address: Data:
0x80258: 0x000ff3af
0x8025c: 0x00000018
0x80260: 0x0000035c
0x80264: 0x00000011
0x80268: 0x00000008
0x8026c: 0x00000001
0x80270: 0x0000025c
0x80274: 0x00000003
0x80278: 0x00000204
0x8027c: 0x00000001
0x80280: 0x00000781
```

```
sw ra, 0(sp)
sw t0, 4(sp)
# SAMPLE POINT -
addi a0, a0, 4
addi a1, a1, -1
jal arrayProd
```

After the initial function call we save the first ra to the stack before the 2nd snapshot is taken before we enter jal arrayProd

B1) What is the hexadecimal address of the instruction that originally calls `arrayProd`?

- ra is 0x00000204
- Therefore, the address of the original call is ra - 4
  - **Ans: 0x00000200**

```
#2  sp  =0x00080278 ra  =0x00000204
    a0  =0x00004000 a1  =0x00000005
Address: Data:
0x80258: 0x000ff3af
0x8025c: 0x00000018
0x80260: 0x0000035c
0x80264: 0x00000011
0x80268: 0x00000008
0x8026c: 0x00000001
0x80270: 0x0000035c
0x80274: 0x00000011
0x80278: 0x00000204
0x8027c: 0x00000001
0x80280: 0x00000781
```

B2) What is the hexadecimal address of the instruction that is responsible for the recursive calls to `arrayProd`?

B2) What is the hexadecimal address of the instruction that is responsible for the recursive calls to `arrayProd`?

```
arrayProd:
   lw t0, 0(a0)
   li a3, 1
   beq a1, a3, end
   addi sp, sp, -8
   sw ra, 0(sp)
   sw t0, 4(sp)
   # SAMPLE POINT - prints ra, sp, a0, a1, a part of the stack
   addi a0, a0, 4
   addi a1, a1, -1
   jal arrayProd
   lw a1, 4(sp)
   jal mult  # returns product of a0 and a1 (see appendix)
   ______________
   lw ra, 0(sp)
   addi sp, sp, 8
end:
   mv a0, t0
   ret
```

B2) What is the hexadecimal address of the instruction that is responsible for the recursive calls to `arrayProd`?

```
arrayProd:
   lw t0, 0(a0)
   li a3, 1
   beq a1, a3, end
   addi sp, sp, -8
   sw ra, 0(sp)
   sw t0, 4(sp)
   # SAMPLE POINT - prints ra, sp, a0, a1, a part of the stack
   addi a0, a0, 4
   addi a1, a1, -1
   jal arrayProd
   lw a1, 4(sp)
   jal mult  # returns product of a0 and a1 (see appendix)
   ______________
   lw ra, 0(sp)
   addi sp, sp, 8
end:
   mv a0, t0
   ret
```

that address is saved over and over 2 instructions apart

After the initial call, jal instruction calls arrayProd multiple times, and ra is set to the same address (jal instruction address + 4) every time

B2) What is the hexadecimal address of the instruction that is responsible for the recursive calls to `arrayProd`?

- ra is 0x0000025C (not initial ra), it is repeated throughout the stack 2 instructions apart
- The recursive call will be ra - 4
  - **Ans: 0x00000258**

```
#4   sp  =0x00080268 ra  =0x0000025C
     a0  =0x00004008 a1  =0x00000003
Address: Data:
0x80258: 0x000ff3af
0x8025c: 0x00000018
0x80260: 0x0000035c
0x80264: 0x00000011
0x80268: 0x0000025c
0x8026c: 0x00000005
0x80270: 0x0000025c
0x80274: 0x00000003
0x80278: 0x00000204
0x8027c: 0x00000001
0x80280: 0x00000781
```

```
#5   sp  =0x00080260 ra  =0x0000025C
     a0  =0x0000400C a1  =0x00000002
Address: Data:
0x80258: 0x000ff3af
0x8025c: 0x00000018
0x80260: 0x0000025c
0x80264: 0x00000008
0x80268: 0x0000025c
0x8026c: 0x00000005
0x80270: 0x0000025c
0x80274: 0x00000003
0x80278: 0x00000204
0x8027c: 0x00000001
0x80280: 0x00000781
```

```
#6   sp  =0x00080280 ra  =0x00000204
     a0  =0x000000F0 a1  =0x00000000
Address: Data:
0x80258: 0x000ff3af
0x8025c: 0x00000018
0x80260: 0x0000025c
0x80264: 0x00000008
0x80268: 0x0000025c
0x8026c: 0x00000005
0x80270: 0x0000025c
0x80274: 0x00000003
0x80278: 0x00000204
0x8027c: 0x00000001
0x80280: 0x00000781
```

B3) What is the hexadecimal address of the array `a` provided to the initial call of `arrayProd`?

B3) What is the hexadecimal address of the array `a` provided to the initial call of `arrayProd`?

```
arrayProd:
    lw t0, 0(a0)
    li a3, 1
    beq a1, a3, end
    addi sp, sp, -8
    sw ra, 0(sp)
    sw t0, 4(sp)
    # SAMPLE POINT - prints ra, sp, a0, a1, a part of the stack
    addi a0, a0, 4
    addi a1, a1, -1
    jal arrayProd
    lw a1, 4(sp)
    jal mult  # returns product of a0 and a1 (see appendix)
    ______________
    lw ra, 0(sp)
    addi sp, sp, 8
end:
    mv a0, t0
    ret
```

a0 is passed into the initial call of arrayProd
It is address of array a

Before arrayProd is called a0 needs to be loaded with the address of array a → let's look at snapshot 1

B3) What is the hexadecimal address of the array `a` provided to the initial call of `arrayProd`?

- Look at arguments that `arrayProd` takes in
  - `uint32_t* a` in a0
  - `uint32_t b` in a1
- `a` is the pointer of the array provided to `arrayProd`
  - A pointer is a variable that stores the address of something in memory
- Snapshot 1 shows us the value of a0 right before we first call `arrayProd`

```
#1   sp   =0x000802[illegible]0 ra   =0x00000000
     a0   =0x00004000 a1   =0x00000005
Address: Data:
0x80258: 0x000ff3af
0x8025c: 0x00000018
0x80260: 0x0000035c
0x80264: 0x00000011
0x80268: 0x00000008
0x8026c: 0x00000001
0x80270: 0x0000035c
0x80274: 0x00000011
0x80278: 0x00000808
0x8027c: 0x0000a321
0x80280: 0x00000781
```

B4) Specify a C array below that is identical to the one the user must have handed into `arrayProd`.

B4) Specify a C array below that is identical to the one the user must have handed into `arrayProd`.

```
arrayProd:
   lw t0, 0(a0)
   li a3, 1
   beq a1, a3, end
   addi sp, sp, -8
   sw ra, 0(sp)
   sw t0, 4(sp)
   # SAMPLE POINT - prints ra, sp, a0, a1, a part of the stack
   addi a0, a0, 4
   addi a1, a1, -1
   jal arrayProd
   lw a1, 4(sp)
   jal mult  # returns product of a0 and a1 (see appendix)
   ______________
   lw ra, 0(sp)
   addi sp, sp, 8
end:
   mv a0, t0
   ret
```

Collects a[0]

Saves it to the stack after ra

a-1 is sent as an argument to the recursive call

B4) Specify a C array below that is identical to the one the user must have handed into `arrayProd`.

```
arrayProd:
    lw t0, 0(a0)
    li a3, 1
    beq a1, a3, end
    addi sp, sp, -8
    sw ra, 0(sp)
    sw t0, 4(sp)
    # SAMPLE POINT - prints ra, sp, a0, a1, a part of the stack
    addi a0, a0, 4
    addi a1, a1, -1
    jal arrayProd
    lw a1, 4(sp)
    jal mult  # returns product of a0 and a1 (see appendix)
    ______________
    lw ra, 0(sp)
    addi sp, sp, 8
end:
    mv a0, t0
    ret
```

Collects a[0]

UNTIL we hit the base case of b == 1, we don't save the final value in the a array

Saves it to the stack after ra

a-1 is sent as an argument to the recursive call

B4) Specify a C array below that is identical to the one the user must have handed into `arrayProd`.

We have → [1, 3, 5, 8, ?]

b == 5 so we need 5 elements

```
#1  sp  =0x00080280 ra  =0x00000000
    a0  =0x00004000 a1  =0x00000005
Address: Data:
0x80258: 0x000ff3af
0x8025c: 0x00000018
0x80260: 0x0000035c
0x80264: 0x00000011
0x80268: 0x00000008
0x8026c: 0x00000001
0x80270: 0x0000035c
0x80274: 0x00000011
0x80278: 0x00000808
0x8027c: 0x0000a321
0x80280: 0x00000781
```

Before call to arrayProd

```
#2  sp  =0x00080278 ra  =0x00000204
    a0  =0x00004000 a1  =0x00000005
Address: Data:
0x80258: 0x000ff3af
0x8025c: 0x00000018
0x80260: 0x0000035c
0x80264: 0x00000011
0x80268: 0x00000008
0x8026c: 0x00000001
0x80270: 0x0000035c
0x80274: 0x00000011
0x80278: 0x00000204
0x8027c: 0x00000001
0x80280: 0x00000781
```

```
#3  sp  =0x00080270 ra  =0x0000025C
    a0  =0x00004004 a1  =0x00000004
Address: Data:
0x80258: 0x000ff3af
0x8025c: 0x00000018
0x80260: 0x0000035c
0x80264: 0x00000011
0x80268: 0x00000008
0x8026c: 0x00000001
0x80270: 0x0000025c
0x80274: 0x00000003
0x80278: 0x00000204
0x8027c: 0x00000001
0x80280: 0x00000781
```

```
#4  sp  =0x00080268 ra  =0x0000025C
    a0  =0x00004008 a1  =0x00000003
Address: Data:
0x80258: 0x000ff3af
0x8025c: 0x00000018
0x80260: 0x0000035c
0x80264: 0x00000011
0x80268: 0x0000025c
0x8026c: 0x00000005
0x80270: 0x0000025c
0x80274: 0x00000003
0x80278: 0x00000204
0x8027c: 0x00000001
0x80280: 0x00000781
```

```
#5  sp  =0x00080260 ra  =0x0000025C
    a0  =0x0000400C a1  =0x00000002
Address: Data:
0x80258: 0x000ff3af
0x8025c: 0x00000018
0x80260: 0x0000025c
0x80264: 0x00000008
0x80268: 0x0000025c
0x8026c: 0x00000005
0x80270: 0x0000025c
0x80274: 0x00000003
0x80278: 0x00000204
0x8027c: 0x00000001
0x80280: 0x00000781
```

```
#6  sp  =0x00080280 ra  =0x00000204
    a0  =0x000000F0 a1  =0x00000000
Address: Data:
0x80258: 0x000ff3af
0x8025c: 0x00000018
0x80260: 0x0000025c
0x80264: 0x00000008
0x80268: 0x0000025c
0x8026c: 0x00000005
0x80270: 0x0000025c
0x80274: 0x00000003
0x80278: 0x00000204
0x8027c: 0x00000001
0x80280: 0x00000781
```

B4) Specify a C array below that is identical to the one the user must have handed into `arrayProd`.

In the last snapshot sp is back in the original position.

At the end of the arrayProd function we have:

```
    addi sp, sp, 8
end:
    mv a0, t0
    ret
```

We reset the stack pointer and then move the answer into a0.

If the stack pointer is in its original position we have completed the function. If the function is complete the **final product is in a0**

```
#6  sp  =0x00080280 ra  =0x00000204
    a0  =0x000000F0 a1  =0x00000000
Address: Data:
0x80258: 0x000ff3af
0x8025c: 0x00000018
0x80260: 0x0000025c
0x80264: 0x00000008
0x80268: 0x0000025c
0x8026c: 0x00000005
0x80270: 0x0000025c
0x80274: 0x00000003
0x80278: 0x00000204
0x8027c: 0x00000001
0x80280: 0x00000781
```

B4) Specify a C array below that is identical to the one the user must have handed into `arrayProd`.

In the last snapshot sp is back in the original position.

At the end of the arrayProd function we have:

```
    addi sp, sp, 8
end:
    mv a0, t0
    ret
```

We reset the stack pointer and then move the answer into a0.

If the stack pointer is in its original position we have completed the function. If the function is complete the **final product is in a0**

```
#6  sp  =0x00080280 ra  =0x00000204
    a0  =0x000000F0 a1  =0x00000000
Address: Data:
0x80258: 0x000ff3af
0x8025c: 0x00000018
0x80260: 0x0000025c
0x80264: 0x00000008
0x80268: 0x0000025c
0x8026c: 0x00000005
0x80270: 0x0000025c
0x80274: 0x00000003
0x80278: 0x00000204
0x8027c: 0x00000001
0x80280: 0x00000781
```

- 0x0F0 = 240 = 1*3*5*8*???
    - ??? = 2
- The array is `{ 1, 3, 5, 8, 2 }`

B4) Specify a C array below that is identical to the one the user must have handed into `arrayProd`.

- Let's look at the last snapshot for values added in the stack
  - `{ 1, 3, 5, 8 }` coupled with `ra`'s
- Notice that the value in `a0` is the result of multiplying each element in the array
- From the first snapshot, `a1` was 5
  - `a1` corresponds to the length of the array
- 0x0F0 = 240 = 1*3*5*8*???
  - ??? = 2
- The array is `{ 1, 3, 5, 8, 2 }`

```
#6  sp  =0x00080280 ra  =0x00000204
    a0  =0x000000F0 a1  =0x00000000
Address: Data:
0x80258: 0x000ff3af
0x8025c: 0x00000018
0x80260: 0x0000025c
0x80264: 0x00000008
0x80268: 0x0000025c
0x8026c: 0x00000005
0x80270: 0x0000025c
0x80274: 0x00000003
0x80278: 0x00000204
0x8027c: 0x00000001
0x80280: 0x00000781
```