CSE 30 Notes

Computer Organization and Systems Programming

Spring 2021

Taught by Professor Brian Chin

Table of Contents

1 Number Systems

In this section, we will briefly discuss the different number systems used, with emphasis placed on binary numbers.

1.1 Common Number Bases

There are four common number bases:

Base	Symbols	Name	Examples
2	0, 1	Binary	0b101101111
8	0-7	Octal	0321
10	0-9	Decimal	1234
16	0-9, a-f (10-16)	Hexadecimal	0xfedba123

There are a few things we should point out. In particular, when written out:

- **Binary** numbers always start with `0b`.
- **Octal** numbers always start with `0`.
- **Hexadecimal** numbers always start with `0x`. The letters can be uppercase or lowercase.

Sometimes, the base will be written as a subscript directly after the number. For example, we can write `0b10101` like 10101_2.

1.2 Positional Encoding and Significance

Each place (digit) represents some power of the base. Consider a number with n digits laid out like so[1]:

$$d_{n-1}d_{n-2}\dots d_3d_2d_1d_0$$

Where d_i is a digit in a specific place. Using a polynomial, we can represent this number like so:

$$\begin{aligned}\text{Number} &= \sum_{i=0}^{n-1} d_i b^i \\ &= d_{n-1}b^{n-1} + d_{n-2}b^{n-2} + \cdots + d_2b^2 + d_1b^1 + d_0b^0\end{aligned}$$

Here, d is the digit and b is the base.

We call the left-most digit, or the $(n-1)$th digit, the *most significant digit* since this digit contributes a lot to the number as a whole. We call the right-most digit, or the 0th place, the *least significant digit* since this digit contributes very little to the number as a whole. For instance, consider the decimal number `12345`, which has $n = 5$ digits. We can write this number out using the polynomial representation:

$$\sum_{i=0}^{5-1} a_i b^i = 1\cdot 10^4 + 2\cdot 10^3 + 3\cdot 10^2 + 4\cdot 10^1 + 5\cdot 10^0 = 10000 + 2000 + 300 + 40 + 5 = 12345$$

- Here, we know that $1\cdot 10^4 = 10000$, which contributes the most to the overall number. Thus, 1 is the most significant digit.
- We also know that $5\cdot 10^0 = 5$ contributes the least to the overall number. Thus, 5 is the least significant digit.

This same exact concept can be applied to numbers of different bases (binary, octal, hexadecimal, and other bases). For instance, if we consider the number `0b10101` (which has $n = 5$ digits), we know that[2]:

$$\sum_{i=0}^{5-1} a_i b^i = 1\cdot 2^4 + 0\cdot 2^3 + 1\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 = 10000 + 0 + 100 + 0 + 1 = 0b10101$$

[1] For example, if we have the number `123`, then this number has 3 digits, so $n = 3$. This means that $d_{n-1} = d_{3-1} = d_2 = 1$, $d_{n-2} = d_{2-1} = d_1 = 2$, and so on.

[2] Here, we have performed binary addition, which will be discussed later on.

1.3 Converting Between Bases

Often times, we'll need to convert between different bases. Here, we'll summarize how to do this.

1.3.1 Binary and Hexadecimal

To convert between binary and hexadecimal numbers, we should note that every four-bit sequence (i.e. 4 binary digits) corresponds to one hexadecimal number. The following table gives a better idea of how this works[3]:

Binary	Hexadecimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A (10)
1011	B (11)
1100	C (12)
1101	D (13)
1110	E (14)
1111	F (15)

The conversion between binary and hexadecimal works like so:

- To convert from binary to hexadecimal, begin by dividing up the bits into chunks of four, from right to left. If the left-most chunk does not have 4 bits, you can pad that chunk with leading zeroes until you have 4 bits. Then, you can make use of the above table.

  For instance, to convert `0xb491` to binary, note that:

  ```
  b       4       9       1
  1011    0100    1001    0001    ->      0b1011010010010001
  ```

- To convert from hexadecimal to binary, simply translate each individual hexadecimal digit to its corresponding binary number.

  For instance, to convert `0b1111011001` to hexadecimal:

  ```
  0b1111011001
  -> 11 1101 1001          (Break into chunks of four)
  -> 0011 1101 1001        (Pad the left-most chunk if needed)
  -> 3    d    9           (Convert to hexadecimal digits)
  -> 0x3d9                 (Combine)
  ```

[3]Note that this table is basically counting from 0 to 15 in both binary and hexadecimal. In this sense, you should not have to memorize this table.

1.3.2 Binary and Octal

The idea is exactly the same as when converting between binary and hexadecimal. The only difference is that we are dealing with every three-bit sequence (i.e. 3 binary digits) as opposed to a four-bit sequence. In other words, every three-bit sequence corresponds to one octal number. The following table gives a better idea of how this works[4]:

Binary	Hexadecimal
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Aside from this difference, the process of converting between binary and octal is basically the same as with binary and hexadecimal.

1.3.3 Hexadecimal and Octal

To convert between hexadecimal and octal, we can simply take the immediate step of finding the binary representation of a number and *then* converting that binary representation to the other base.

For instance, suppose we have the octal number `0763` (or 763_8). The binary representation of this number is:

```
111 110 011
```

Regrouping these bits gives us[5]:

```
0001 1111 0011
```

And then converting this to hexadecimal gives us `0x1f3`.

1.3.4 Converting to Decimal

Converting to decimal is very straightforward. Consider a number with n digits that is in base b. Then, the most significant digit is the $(n-1)$th digit (the left-most digit) and the least significant digit is the 0th digit (the right-most digit). For every digit d_i (where i is the location of the digit relative to the least significant digit), we have:

$$\begin{aligned}\text{Number} &= \sum_{i=0}^{n-1} d_i b^i \\ &= d_{n-1}b^{n-1} + d_{n-2}b^{n-2} + \cdots + d_2 b^2 + d_1 b^1 + d_0 b^0\end{aligned}$$

You can use this same formula to find the decimal form of any given base.

For instance, if we have the hexadecimal number `0xb491`, we know that the base is 16 and the most significant digit is b, which is located at position 3 ($n = 4$ and $4 - 1 = 3$). Then:

$$11 \cdot 16^3 + 4 \cdot 16^2 + 9 \cdot 16^1 + 1 \cdot 16^0 = 46225$$

So, the hexadecimal number `0xb491` is equivalent to `46225` in decimal.

[4] Same idea as with the previous footnote. The only difference is that we're counting from 0 to 7 in both binary and octal.
[5] Note that we padded three zeroes in the left-most chunk.

1.3.5 Converting from Decimal

To do this, we make use of an algorithm.

- Given `number` and base `b`.
- Start with the least-significant digit (`n = 0`).
- Repeat until `number == 0`:
  - Divide `number` by `b` to produce a quotient and remainder.
  - Place the remainder in the b^n position.
  - Let `number = quotient`.
  - Increment `n` by one.

For instance, let's suppose we have 72_{10} and we wanted to convert this number to base 7. Using this algorithm, we have:

- $n = 0$ with `number = 72`: $\frac{72}{7} = 10$ with remainder **2** (least-significant digit).
- $n = 1$ with `number = 10`: $\frac{10}{7} = 1$ with remainder **3**.
- $n = 2$ with `number = 1`: $\frac{1}{7} = 0$ with remainder **1** (most-significant digit).

So, it follows that $71_{10} = 132_7$.

1.4 Signed vs. Unsigned Binary Integers

An *unsigned* integer is simply an integer that is strictly non-negative. A *signed* integer is simply an integer that can be negative, positive, or zero.

1.4.1 Signed Magnitude and One's Complement

Signed magnitude simply treats the high-order bit as a sign bit. That is, instead of including the most significant bit (sign bit) as part of the number, we only use the most-significant bit to determine if the number is positive (`0xxx...`) or negative (`1xxx...`).

Computing a decimal value for an n-bit signed magnitude sequence is simple: simply calculate the value of the digits d_0 through d_{n-2} using the method discussed in the first page and then change the sign of this number based on the most significant bit.

To negate the value, we simply take the *one's complement*, or simply flip the bits such that all `0`s become `1`s and all `1`s becomes `0`s.

However, this suffers from one major drawback: there are two representations of zero. For instance, for 4-bit binary, `0b1000` represents `-0` while `0b0000` represents `+0`.

1.4.2 Two's Complement

We can solve the issue presented by one's complement by using the two's complement encoding. Here, the most significant bit (the sign bit) will determine the sign of the number while also being included in the computation of the number itself.

If the sign bit is `0`, then we can compute the number like so:

$$d_{n-2} \cdot 2^{n-2} + \cdots + d_2 \cdot 2^2 + d_1 \cdot 2^1 + d_0 \cdot 2^0$$

If the sign bit is `1`, then we must also include the sign bit in the computation, which can be done like so:

$$-d_{n-1} \cdot 2^{n-1} + d_{n-2} \cdot 2^{n-2} + \cdots + d_2 \cdot 2^2 + d_1 \cdot 2^1 + d_0 \cdot 2^0$$

It follows that the range is simply $-2^{n-1} \leq x \leq 2^{n-1} - 1$.

To negate the value, we simply take the *one's complement* like usual, but then also add one to the resulting value. In other words:

- Flip all bits.
- Add one to result.

For instance, to negate `0111` (`7`), flip all bits (`1000`) and then add one (`1001`). Thus, the negation of `0111` is `1001` (`-7`).

1.5 Binary Arithmetic

It should be noted that the paper-and-pencil method that we learned can easily be applied to binary arithmetic. Thus, we will not cover this in great detail.

1.5.1 Binary Addition

This is exactly the same as doing regular addition, except that you are restricted to `0` and `1` instead of what we're used to with the decimal system.

1.5.2 Binary Subtraction

Same deal. However, we should note that two's complement can be used to make things easier. In particular, it should be noted that:

$$A - B = A + (-B)$$

So, we can easily use two's complement to negate B, thus allowing us to add A and $-B$ without having to deal with actual subtraction.

1.5.3 Binary Multiplication and Division

Same deal. It should be noted that, with binary division, the fractional part will always be truncated.

1.5.4 Multiplying and Dividing by Powers of 2

Consider the following table.

	$a_5 \cdot 2^5$	$a_4 \cdot 2^4$	$a_3 \cdot 2^3$	$a_2 \cdot 2^2$	$a_1 \cdot 2^1$	$a_0 \cdot 2^0$
10			1	0	1	0
20		1	0	1	0	0
40	1	0	1	0	0	0
5				1	0	1

To multiply a number by a power of two, we can simply shift the bits one to the left. For division, we can shift the bits one to the right. In C or C++, this can be represented like so:

```
int a = 10;

int b = a << 1; // 10 * 2^1 = 20
int c = a >> 1; // 10 / 2^1 = 5

int d = a << 2; // 10 * 2^2 = 40
int e = a >> 2; // 10 / 2^2 = 2
```

1.6 Overflow

When dealing with a fixed number of bits, we need to be aware of limitations. In particular, if we have a fixed number of bits, we cannot store some numbers.

Detecting overflow is relatively straightforward.

- Add the two numbers.
- If the sign bits are different, we cannot have overflow. In other words, $A + (-B)$ or $(-A) + B$ will not produce an overflow.
- Two positive numbers must sum to a positive number.
- Two negative numbers must sum to a negative number.

For instance, `1000 - 0001 = 1000 + 1111` will result in an overflow, as will `0110 + 0111`.

1.7 Fixed Point Numbers

This is relatively straightforward. Basically, a fixed point number is a number with a *fixed* number of digits after the decimal point.

We can only represent a fixed point number of the form $\frac{x}{2^k}$ for $x, k \in \mathbb{Z}$. Otherwise, we will have repeating bits.

We can express fixed point numbers in polynomial form, similar to how we did it with integers, like so:

$$d_{n-1}b^{n-1} + d_{n-2}b^{n-2} + \cdots + d_1b^1 + d_0b^0 + d_{-1}b^{-1} + d_{-2}b^{-2} + \ldots$$

For instance, `10/4 = 2.5` can be represented by the binary number 10.1 (you would have to do long division to get this result).

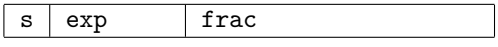
1.8 Floating Point Numbers

We use floating point numbers to express numbers that cannot be expressed using fixed point numbers.

1.8.1 Representation

Generally speaking, we can represent floating point numbers in one of two ways:

- Numerically:
$$(-1)^S M \cdot 2^E$$
  - S represents the sign bit. This tells you if the number is negative (`1`) or positive (`0`).
  - M represents the mantissa. This is the fractional part of the number (after normalizing).
  - E represents the exponent.
- Encoding:



  - The `s` field is the sign bit.
  - The `exp` field encodes E, but is not equal to E.
    - We introduce a concept known as a biased exponent. The formula is:
$$\textbf{exp} = E + \textbf{bias}$$
  - The `frac` field encodes M, but is not equal to M.

1.8.2 Converting Decimal to Floating Point

To convert from decimal to floating point, we do the following:

(1) Convert the integer and fraction separately.

- Convert the integer to a decimal using the algorithm mentioned in page 4.
- For the fractional part, multiply by 2 and use the digit to the left of the decimal as the binary digit.

(2) Express this number as a binary fixed point.

(3) Normalize. Here, we define a normalized number to be one with a leading non-zero digit.

1.8.3 Converting Floating Point to Decimal

To convert from floating point to decimal, we do the following:

(1) Decompose the bits so that you can easily tell what the sign field, exponent field, and mantissa are.

- Since you are given the exponent field, convert this to the actual exponent by using the formula:

$$\textbf{exp} = E + \textbf{bias}$$

(2) Write out the number in binary scientific notation form:

$$1.\text{mmmm} \cdot 2^E$$

Where `mmmm` is the mantissa field.

(3) Expand out the number that you found in the previous step. Then, convert it to decimal. Be sure to account for the sign bit.

1.8.4 Examples

For instance, let's suppose we have a 9-bit floating point format, with an exponent bias of 7 and an exponent field width of 4 bits. Furthermore, let's suppose we are given `+0.3125` and want to convert this to its equivalent hexadecimal floating point value.

(1) We want to convert the integer and fraction separately.

- The integer value is `0`, which is `0` in binary. We do not need to do anything with this.
- The fractional value is `0.3125`. To convert this to binary:
  - $0.3125 \cdot 2 = \mathbf{0}.625$
  - $0.625 \cdot 2 = \mathbf{1}.25$
  - $0.25 \cdot 2 = \mathbf{0}.50$
  - $0.50 \cdot 2 = \mathbf{1}.00$

  So, we have `.0101` as our decimal[6].

(2) Our binary fixed point is `0 + 0.0101 = 0.0101`.

(3) Normalizing this, we have $1.01 \cdot 2^{-2} = 1.0100 \cdot 2^{-2}$.

- Since the bias is 7, we know that `exp = E + bias = -2 + 7 = 5`. In binary, this is `0101`.
- The mantissa (`frac`) is `0100` (we omit the leading `1`).
- The sign is `0` (the given number is positive).

[6] When compared to how you would use the algorithm, this is the main difference. In the main algorithm, after we found the digits, we went from bottom-to-top (132 instead of 231 in the example). Here, we're going from top-to-bottom (0101 instead of 1010)

(4) Encoding this, we have:

s	exp	frac
0	0101	0100

So, our final binary representation is[7]:

```
0000 0101 0100
```

And thus, our hexadecimal representation is:

```
0x054
```

For instance, let's suppose we have a 10-bit floating point format, with an exponent bias of 7 and an exponent field of 4 bits rather than 3. What is `0x33C` in decimal?

(1) We want to begin by converting `0x33C` to binary. This is simply:

```
11 0011 1100
```

Decomposing this, we know that:

- `1` is the sign bit. This means that the number is negative.
- `1001` is the exponent field. We know that $\mathbf{exp} = 1001_2 = 9_{10}$. Since the bias is 7, we know that the exponent is:
$$\mathbf{exp} = 9 = E + 7 \iff E = 2$$
So, the exponent is 2.
- `11100` is the mantissa.

(2) Now, we want to write this number out into scientific notation form. This would look like:

$$1.11100 \cdot 2^2 = 111.1$$

(3) Converting this to decimal, we have:

$$2^2 + 2^1 + 2^0 + 2^{-1} = 7.5$$

Since the sign bit was `1`, this number is negative. So, our final answer is:

$$-7.5$$

1.8.5 The Denormal Case

A denormal number is one where the exponent field is all zeroes. There is no implied leading one. Despite the exponent field being all zeroes, you can assume that the exponent E will be the lowest possible E (in other words, an exponent field containing all zeroes is the same as the lowest possible exponent field).

[7] Here, we padded the 0s to the left of the sign bit

2 The C Programming Language

In this section, we will briefly discuss the C programming language. Heavier emphasis will be placed on pointers, memory management, strings, `structs`, and more.

2.1 Basic Data Types

C has several basic data types.

- `int`.
- `char`.
- `float`.
- `short`.
- `long`.

By default, these are signed. For example, the following line of code defines a *signed* integer:

```
int x = -5;
```

If you want the unsigned type, simply prefix the type with `unsigned`. For example:

```
unsigned long y = 2;
```

2.2 Basic Data Object in Memory

Consider the following code:

```
int num;
num = 20;
```

Here, we define a variable `num` with value `20`, which is `4` bytes. This will be located in a particular memory address, which will be assigned by the compiler.

Every variable generally has the following attributes:

- The name/identifier - *how will you refer to this variable?*
- The value - *What is the value of this variable? In other words, what value is this variable holding?*
- The address - *Where in memory is this value located?*
- The size - *What is the size of this value? Or, rather, what's the maximum size that this variable can hold?*
- The data type - *What is the variable is supposed to be?*
- The lifetime - *How long is this variable supposed to last?*
- The scope - *Where was this variable defined?*

2.3 Strings

C does not have a dedicated variable type for strings. Instead, we can think of strings as an **array of characters** with a null character[8] at the end. The null character is simply a backslash followed by a zero: `\0`.

Let's consider the string `Hello`:

Index	0	1	2	3	4	5
char	'H'	'e'	'l'	'l'	'o'	\0

We should note the following:

- A valid string must have a null character at the end. If it doesn't have a null character, it's not a valid string.
- You must allocate one extra byte in an array for the null character.
- Strings do not always have a newline, so don't depend on a newline being right before the null-terminating character.
- Strings are *not* objects. They have no additional embedded information, like string length or methods.
- If you must calculate the length of a string, use the C string library `strlen` function to calculate the string length. In this case, the null character is not counted.[9]

Consider the string declarations:

```
char mess1[] = "Hello World";
char mess2[] = {'H', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd', '\0'};
char mess3[] = {'H', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd'};
```

`mess1` is an array that has enough space to hold the contents of the string and the null character (here, an implict null character is added at the end). Thus, `mess1` is a valid string. `mess2` is exactly the same as `mess1`, but more explicit. In memory, `mess1` or `mess2` would be stored like so[10]:

Value	**Byte Address**
...	...
00110011	0x0100000d
01000101	0x0100000c
00000000 (\0)	0x0100000b
'd'	0x0100000a
'l'	0x01000009
'r'	0x01000008
'o'	0x01000007
'w'	0x01000006
' '	0x01000005
'o'	0x01000004
'l'	0x01000003
'l'	0x01000002
'e'	0x01000001
'H'	0x01000000

On the other hand, `mess3` is an invalid string since there is not a null-terminating charaacter at the end.

[8] A null character, also known as the null-terminating character, marks the *end* of a string.

[9] Keep in mind that `strlen` is $O(n)$ because it must scan the entire string. Thus, if you need the length of the string at a later point, you should save the value.

[10] The byte addresses will be determined by the compiler. These are simply examples.

2.4 Pointers

A pointer is a variable that contains the address of a variable. In other words, a pointer stores a reference to something. A pointee is the thing that the pointer points to. The address of a variable is simply a number that indicates where the data is stored in memory (we say that this address represents the place in memory where the data lives).

A pointer can only point to one type and can represent a basic or derived type such as an `int`, `char`, a `struct`, another pointer, and more.

2.4.1 Pointer Declaration

To declare a pointer, simply put a star (`*`) directly before the name of the variable or directly after the data type. For example, the following are all valid[11]:

```
int *x;
int* x;
```

Here, `x` is a pointer to an integer. `x` is not an integer.

When we declare a pointer like what we did above, we note that `x` doesn't actually point to anything yet. We can either:

- Make it point to something that already exists.
- Or, allocate room in memory for something new that it will point to.

2.4.2 Addresses

To get the address of a variable, simply put an ampersand (`&`) before the name of a variable. For example:

```
int y = 3;
int* x = &y;
```

Here, `&y` gives you the address of `y`.[12]

2.4.3 Dereference/Indirection Operator

We can use `*` to declare a pointer. We can also use `*` to *dereference* a pointer. When we dereference a pointer, we are *retriving* the value from the memory address that is pointed by the pointer.

To dereference a pointer, we simply put the `*` *before* the pointer variable name. For instance:

```
int x = 10;         // Declare variable "x" that holds value "10"
int* y = &x;        // Declare int pointer "y" that holds address to x
int z = *y;         // Declare "z" to be the value "y" is pointing to (dereferencing)
printf("%i\n", x);  // Prints: 10
printf("%p\n", y);  // Prints: 0x7ffdd1cd2578 (some memory address)
printf("%i\n", z);  // Prints: 10
*y = 5;             // Set the value at address 0x7ffdd1cd2578 to "5"
printf("%i\n", x);  // Prints: 5
printf("%p\n", y);  // Prints: 0x7ffdd1cd2578 (same memory address)
printf("%i", z);    // Prints: 10
```

As noted in the example, we can also use the dereference operator to:

- Change the value located at whatever address the pointer has to a different value.
- Set a new variable with the value located at whatever address the pointer is pointing to.

[11] In my opinion, declaring a pointer as `int* x;` (as opposed to `int *x;`) makes it more clear what `x` is.
[12] `&y` is **not** an address. `&y` gives you the address to `y`.

2.5 Arrays and Pointer Arithmetic

In C, an array is simply a contiguous (adjacent) region of memory.

Declaring an array is as simple as:

```
int arr[5];                   // Declare int array of size 5. If no initial values
                              // are given, you must explicitly state the size.
int arr[] = {1, 2, 3, 4, 5};  // Declare int array of size 5 (implicitly) with
                              // initial values.
int arr[5] = {1, 2, 3, 4, 5}; // Declare int array of size 5 (explicitly) with
                              // initial values.
int arr[5] = {1, 2, 3};       // Declare int array of size 5 (explicitly) with
                              // some initial values.
```

A pointer can *point* to an array. In fact, when we define an array, we are given the pointer to the first element. For instance:

```
int arr[3];                 // "arr" is a pointer to the first element in this array.
```

So, we say that `arr` is a pointer to the first element. Then:

- `arr[0]` is the same as `*arr`
- `arr[2]` is the same as `*(arr + 2)`

What we did in the second bullet point is known as *pointer arithmetic*. Here's a more concrete example that showcases pointer arithmetic:

```
// Declare an array of 10 int elements.
// Suppose the first element was located at address 100.
int array[10];
// Iterate through every element.
// Set every element to its index.
for (int i = 0; i < 10; i++) {
    // The following two lines are equivalent.
    *(array + i) = i;
    array[i] = i;
}
// Define a pointer to array.
// ptr is now pointing to the first element (array[0])
int* ptr = array;
// Increment pointer. Now, ptr is pointing to the second element.
ptr++;
// Set the second element to 100. Same as: array[1] = 100;
*ptr = 100;
// Decrement pointer. Now, ptr is pointing to the first element.
ptr--;
// Set the first element to 50. Same as: array[0] = 50;
*ptr = 50;
```

A few notes to consider:

- `*(array + 1)` is *not* the same as `*array + 1`. The former is equivalent to `array[1]` while the latter is equivalent to `array[0] + 1`.
- When doing pointer arithmetic, C will automatically handle the byte size. For instance, when you do `*(array + 1)` where `array` is an `int` array, C will know to add 4 to the pointer (since an `int` has 4 bytes).

2.5.1 Legality of Pointer Operators

The following are some common pointer arithmetic operators, along with whether said operators are legal.

Operator	Legal?
`++`, `--`, `[]`, `==`, `!=`	Yes
Pointers + Integer	Yes
Pointer - Integer	Yes
Integer - Pointer	No (See 1)
Pointer - Pointer	Yes (See 2)
Pointer + Pointer	No (See 3)
Pointer × Pointer	No (See 4)

(1) You would have to negate the pointers; pointers are always positive.

(2) This returns an integer representing the *distance* in elements between pointers.

(3) This is not guaranteed to be an address.

(4) This is simply repeated addition.

2.6 Functions

When we think of a C function, we think of a function that looks something like this:

```
void foo(int x, int y) {
    int tmp;
    tmp = x;
    x = y;
    y = tmp;
}
```

We note a few things.

- Parameters are always passed by value.
- A function will always have a return type, a name, parameter list, and an implementation.

2.6.1 Header vs. Implementation Files

When writing a C function, you want to put the function header, or function declaration, in a `.h` file (the header file). You put the actual function implementation in the `.c` file. For example:

```
// Function headers (.h file).
void foo(int x, int y);
void bar(int, char);

// Implementation (.c file).
void foo(int x, int y) {
    int z = x + y;
}

void bar(int x, char z) {
    int y = z + x;
}
```

2.6.2 Mimic Pass-by-Reference

We can use pointers to mimic pass-by-reference. For instance, the `swap` function swaps `x` and `y`:

```
void swap(int* x, int* y) {
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}
```

To use this function, we pass the addresses of the two variables we want to swap. For example:

```
int x = 5;
int y = 10;
swap(&x, &y);
// x is 10, y is 5.
```

This is still pass-by-value, but we are passing the address instead of the actual value. Thus, it might be more accurate to call this *pass-by-pointer*.

2.7 Types of Memory

There are two main types of memory that we are concerned with:

- Local variables (and parameters) go on the *stack*.
- Dynamically allocated memory goes on the *heap*.

2.8 Allocating Memory at Runtime

We can allocate memory using `malloc`, `realloc`, or `calloc`. These will go on the *heap*.

2.8.1 Allocate Memory: `malloc`

We can allocate memory using `malloc`. `malloc` has the following function declaration:

```
// malloc initializes the allocate memory with garbage values.
// - The first argument is the amount of memory to allocate, in bytes.
void* malloc(size_t size);
```

Here, `void*` is the return type. `void*` is a pointer to anything (so you can cast this pointer to anything). If this is `NULL`, then an error occurred. Otherwise, this returns the *base address* of the allocated heap memory.

From the `man` page:

> The `malloc()` function allocates `size` bytes and returns a pointer to the allocated memory. The memory is not initialized. If size is 0, then `malloc()` returns either `NULL`, or a unique pointer value that can later be successfully passed to `free()`.

2.8.2 Allocate Memory and Zero Out: `calloc`

We can also allocate memory using `calloc`[13]:

```
// calloc initializes the allocated memory with 0 value.
// - The first argument is the number of blocks to be allocated.
// - The second argument is the size of each block.
void* calloc(size_t nmemb, size_t memsize);
```

[13] `calloc(5, sizeof(type))` is approximately equivalent to `malloc(5 * sizeof(type))`, but the former zeroes out all blocks while the latter keeps the garbage values.

From the `man` page:

> The `calloc()` function allocates memory for an array of `nmemb` elements of `size` bytes each and returns a pointer to the allocated memory. The memory is set to zero. If `nmemb` or `size` is 0, then `calloc()` returns either `NULL`, or a unique pointer value that can later be successfully passed to `free()`.

2.8.3 Reallocate Memory: `realloc`

If we need to, we can resize a block of memory using the `realloc` function:

```
// realloc resizes the memory block pointed to by ptr that
// was previously allocated with a call to malloc or calloc.
// - The first argument is the pointer to the memory block that
//   you want to reallocate.
// - The second argument is the new size for the memory block,
//   in bytes. If this is zero and ptr points to an existing
//   block of memory, then this will deallocate ptr.
void* realloc(void* ptr, size_t size);
```

From the `man` page:

> The `realloc()` function changes the size of the memory block pointed to by `ptr` to `size` bytes. The contents will be unchanged in the range from the start of the region up to the minimum of the old and new sizes. If the new `size` is larger than the old size, the added memory will not be initialized. If `ptr` is `NULL`, then the call is equivalent to `malloc(size)`, for all values of `size`; if `size` is equal to zero, and `ptr` is not `NULL`, then the call is equivalent to `free(ptr)`. Unless `ptr` is `NULL`, it must have been returned by an earlier call to `malloc()`, `calloc()` or `realloc()`. If the area pointed to was moved, a `free(ptr)` is done.

2.8.4 Free Dynamically Allocated Memory: `free`

Of course, when you allocate memory, you must free it (or else it's a memory leak). For this, there's a `free` function. `free` has the following function declaration:

```
void free(void* ptr);
```

From the `man` page:

> The `free()` function frees the memory space pointed to by `ptr`, which must have been returned by a previous call to `malloc()`, `calloc()` or `realloc()`. Otherwise, or if `free(ptr)` has already been called before, undefined behavior occurs. If `ptr` is NULL, no operation is performed.

2.8.5 Basic Example

Consider the following code:

```
// Allocate an int.
int* num = malloc(sizeof(int));
// Free that int.
free(num);
// Allocate an int array of size 10.
int* arr = malloc(sizeof(int) * 10);
arr[5] = 10;
// Free that array.
free(arr);
```

All we did was allocate some memory, free it, and then allocate more memory. Simple as that.

2.9 Structures

A structure, or `struct`, is a data structure composed of simple types. It's simple to classes in Java and C++, but there are no methods or inheritance.

2.9.1 General Syntax

The general syntax for a `struct` declaration is:

```
struct struct_name {
    type member1;
    // ...
    type memberN
};
```

2.9.2 Using an Alias: `typedef`

We can also define an *alias* for the `struct`. The syntax for this is as follows:

```
typedef struct struct_name {
    type member1;
    // ...
    type memberN
} struct_alias;
```

We can also define an alias like so:

```
typedef struct struct_name struct_alias;
struct struct_name {
    type member1;
    // ...
    type memberN
};
```

2.9.3 Defining a Structure

The following three declarations are equal.

- The normal way.

```
struct Point {
    int x_coord;
    int y_coord;
};

struct Point point;
```

- Using an alias.

```
struct Point {
    int x_coord;
    int y_coord;
} p_t;

p_t point;
```

- The alternative alias.

```
typedef struct Point p_t;
struct Point {
    int x_coord;
    int y_coord;
};

p_t point;
```

2.9.4 Allocating Memory for a Structure

Suppose we have the above `Point` struct. Then, allocating memory is as simple as:

```
// Using the full name.
struct Point* pt = malloc(sizeof(Point));
// Using type alias
p_t* pt = malloc(sizeof(p_t));
```

The compiler will determine the size of the given `struct`.

2.9.5 Accessing Members of a Structure

Suppose we wanted to access the `x_coord` member. There are two ways.

- If you are given the structure itself, use the `.` (dot) operator. For example:

```
struct Point pt;
pt.x_coord = 5;
pt.y_coord = 3;
int sum = pt.x_coord + pt.y_coord;
```

- If you are given a pointer to the structure, use the `->` (arrow) operator. For example:

```
struct Point* pt = malloc(sizeof(Point));
pt->x_coord = 5;
pt->y_coord = 3;
int sum = pt->x_coord + pt->y_coord;
```

  It should be noted that the following two lines are equivalent:

```
pt->x_coord = 5;
(*pt).x_coord = 5;
```

2.10 Dangling Pointers vs. Memory Leaks

Here, we will briefly discuss the difference between a dangling pointer and a memory leak, as well as common examples of such.

2.10.1 Dangling Pointers

A **dangling pointer** is a pointer that points to a memory location that no longer exists. In other words, it's when you free an area of memory but still hvae a pointer to the freed memory.

Here are two common examples of dangling pointers.

1. Returning a local address from a function.

```
char* function() {
    // Note that this is a local declaration!
    char str[10];
    strcpy(str, "Hello!");
    return str;
}

int main() {
    char* hello_string = function();
    // Returned pointer points to "str" which has gone out of scope.
    // This is a dangling pointer because "str" no longer exists in
    // this scope.
}
```

2. Accessing a memory location that has already been freed.

```
int* c = malloc(sizeof(int));
free(c);
*c = 3;
// Another example of a dangling pointer because, here, we are writing
// to a freed location. In other words, this memory location no longer
// exists.
```

2.10.2 Memory Leaks

A memory leak is memory in heap that can no longer be accessed. In other words, it's when you lose the pointer but still have the memory allocated.

Here are two common examples of memory leaks.

1. Allocating memory but not keeping a pointer to it.

```
void function() {
    int* int_array = malloc(5 * sizeof(int));
}

// After "function" is done executing, int_array will still be initialized
// but there won't be a pointer to it.
```

2. Allocating memory but overwriting it.

```
int main() {
    int* array1 = malloc(10 * sizeof(int));
    int* array2 = malloc(10 * sizeof(int));
```

```
    array1 = array2;
}

// Here, we created two integer arrays. But, we assigned "array2" to
// "array1." Now, we lost all references to the original contents of
// "array1," which is a memory leak.
```

3 32-Bit ARM Assembly (Part I)

In this section, we will discuss ARM assembly instructions.

3.1 Arithmetic Instructions

We begin by talking about arithmetic instructions.

3.1.1 Format

Generally speaking, we can write an instruction using the format:

```
opcode  DESTINATION, OPERAND1, OPERAND2
```

Where `opcode` is the instruction by name, `DESTINATION` is always the left-most register (right after the opcode), `OPERAND1` is the first operand, and `OPERAND2` is the second operand.

So, for example, suppose we have the following assembly instruction:

```
add     r3, r2, r1
```

The destination register is `r3`; this is where the result goes. Assuming that `r3` corresponds to `c`, `r2` corresponds to `b`, and `r1` corresponds to `a`, this assembly code is equivalent to the C code:

```
c = b + a;
```

We can also introduce *immediates* (which are otherwise known as *constants*). These are prefixed with a pound symbol (`#`) and will always be the **last** operand. That is:

```
add     r3, r2, #5    // Valid.
add     r3, #5, r2    // Invalid.
```

In the first line above, we are using the constant `5` (notice how it is prefixed by a pound symbol). In C, assuming `r3` corresponds to variable `c` and `r2` corresponds to variable `b`, this corresponds to:

```
c = b + 5;
```

We can also use binary and hexadecimal numbers in immediates. For example:

```
add     r3, r2, #0xf
```

3.1.2 Addition and Subtraction of Integers

To add two integers, we use the following instruction:

```
add     DESTINATION, OPERAND1, OPERAND2
```

For example, let's suppose that the C variable `c` corresponds to register `r3`, `b` corresponds to register `r2`, and `a` corresponds to register `r1`. Then, the following C code can be translated to its assembly counterpart:

```
c = b + a;
add     r3, r2, r1

c = a + b;
add     r3, r1, r2

a = b + c;
add     r1, r2, r3

a = a + a;
add     r1, r1, r1

a = b + 2;
add     r1, r2, #2
```

To subtract two integers, we use the following instruction:

```
sub     DESTINATION, OPERAND1, OPERAND2
```

Using the same example as above, the following C code can be translated to its assembly counterpart:

```
c = b - a;
sub     r3, r2, r1

c = a - b;
sub     r3, r1, r2

a = b - c;
sub     r1, r2, r3

a = a - a;
sub     r1, r1, r1

a = b - 2;
sub     r1, r2, #2
```

We should note that, if we need to translate some C code which involves the use of orders of operations (for example, the problem `f = (g + h) - (i + j)`), we will need to make use of an intermediate temporary register. The general strategy is that we should prioritize operations that have higher precedence (for example, for the example problem, we would make a temporary register for `i + j` or `g + h` and then subtract the two registers that we used to get the final result).

3.1.3 Multiplication of Integers

To multiply two integers, we use the following instruction:

```
mul     DESTINATION, OPERAND1, OPERAND2
```

For example, let's suppose that the C variable `c` corresponds to register `r3`, `b` corresponds to register `r2`, and `a` corresponds to register `r1`. Then, the following C code can be translated to its assembly counterpart:

```
a = b * c;
mul     r1, r2, r3

a = b * 5;
mul     r1, r2, #5

b = b * c;
mul     r2, r2, r3
```

3.1.4 Logical/Bitwise Operators

There are four basic logical operators:

- `AND`
- `OR`
- `XOR`
- `BIC` (Bit Clear)

A truth table can better demonstrate how each operation works.

A	B	A AND B	A OR B	A XOR B	A BIC B
0	0	0	0	0	0
0	1	0	1	1	0
1	0	0	1	1	1
1	1	1	1	0	0

Just like with addition and subtraction, the format for these instructions is exactly the same. So, here are some direct translations:

Operator	Example (ARM)	Equivalent To (C)
Bitwise AND	`and r0, r1, r2`	`r0 = r1 & r2`
Bitwise OR	`orr r3, r4, r5`	`r3 = r4 | r5`
Bitwise XOR	`eor r0, r1, r2`	`r0 = r1 ^ r2`
Bitwise Clear BIC	`bic r3, r4, r5`	`r3 = r4 & (!r5)`

Some remarks:

- Notice how the `XOR` operation is actually denoted as `EOR` in assembly!
- We can use these bitwise operators (especially `AND`) to create a bitmask.

3.1.5 Shift Instructions

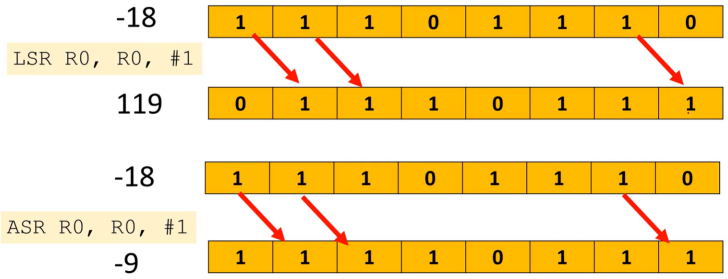
Assume ARM registers `r0`, `r1` are associated with C variables `a` and `b`, respectively. Then:

Operator	Example (ARM)	Equivalent To (C)
<<	`lsl r0,r1,#4`	`a = b << 4`
>> (Zero Extend)	`lsr r0, r1, #8`	(Maybe) `a = b >> 8`
>> (Sign Extend)	`asr r0, r1, #7`	(Maybe) `a = b >> 7`

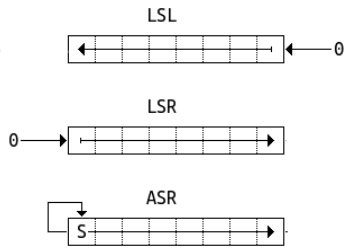
The difference between `ASR` and `LSR` are as follows:

- `LSR` is **l**ogical **s**hift **r**ight. It will shift and the top bits become 0. This is equivalent to dividing an unsigned integer by a power of 2.
- `ASR` is **a**rithmetical **s**hift **r**ight. It will shift the number, taking into account if the number is positive or negative, essentially preserving the highest bit. This has the effect of dividing a signed integer by a power of 2.

A visual diagram[14] is shown below (courtesy of Professor Chin from his review video):



Another visual diagram is shown below:



[14] Note that we are using 8-bits for illustration purposes.

3.2 Assignment Instructions

Now, we will briefly discuss assignment instructions.

3.2.1 Assigning Registers or Immediates

We can assign registers or intermediates to another register like so:

Example (in ARM)	Equivalent to (in C)
`mov r0, r1`	`a = b;`
`mov r0, #10`	`a = 10;`

Where ARM registers `r0`, `r1` are associated with C variables `a` and `b`, respectively.

3.2.2 Move Complement Instruction

The `mvn` instruction moves the one's complement of the operand into the register. You can also think of it as it taking the value of the operand and then performing a bitwise logical NOT operation on the value.

Example (in ARM)	Equivalent to (in C)
`mvn r0, #0`	`a = -1;`
`mvn r0, r2`	`a = ~b`

Where ARM registers `r0`, `r1` are associated with C variables `a` and `b`, respectively.

3.3 Control Transfer

We will discuss control transfer ARM assembly instructions, which include the following:

- Jumping to another instruction.
- Conditional jump.
- Jump to a subroutine.

3.3.1 Conditional Branch

To perform a conditional branch, we must do the following:

- First, set the condition bits (`N`, `Z`, `V`, or `C`) in the program status register.
- Then, check on these condition bits to branch conditionally.

Here, the condition bits mean the following:

- `N`: Negative condition code flag (`< 0`).
- `Z`: Zero condition code flag (`== 0`).
- `C`: Carry condition code flag.
- `V`: Overflow condition code flag.

These condition codes are defined like so[15]:

Suffix	**Flags**	**Meaning**
`eq`	`Z` set.	Equal.
`ne`	`Z` cleared.	Not Equal.
`le`	`z` set; or `n` and `v` different.	Less Than or Equal.
`lt`	`n` and `v` different.	Less Than.
`ge`	`n` and `v` same.	Greater Than or Equal.
`gt`	`z` cleared; or `n` and `v` same.	Greater Than.

[15] You do not need to memorize these.

3.3.2 Comparison Instructions

To set the condition bits mentioned in the previous subsection, we make use of the `cmp` instruction. Essentially, the `cmp` instruction compares and sets condition bits. This subtracts a register or an immediate value from a register value and updates the condition code. Unlike `sub`, this does not store the results anywhere.

Consider the following examples:

```
cmp     r3, #0   // Set Z flag if r3 == 0, or namely r3 - 0 == 0.
cmp     r3, r4   // Set Z flag if r3 == r4, or namely r3 - r4 == 0.
```

Obviously, all flags are set as result of this operation, not just `Z`. Conditional branches are often preceded by `cmp`.

3.3.3 Conditional Branches

ARM has variants of the branch instruction that only goes to the label if a certain condition is *true*. Consider the following table, which contains a subset of all possible conditional branches, where `label` is some label:

Instruction	Example	Meaning
`b`	`b label`	Unconditionally goto label.
`beq`	`beq label`	Branch to Label if equal.
`bne`	`bne label`	Branch to Label if not equal.
`ble`	`ble label`	Branch to Label if less than or equal.
`blt`	`blt label`	Branch to Label if less than.
`bge`	`bge label`	Branch to Label if greater than or equal.
`bgt`	`bgt label`	Branch to Label if greater than.

These conditions are true or false based upon the fields in the program status register. As a final remark, we can use these branches to mimic the behavior of `||` and `&&` when doing comparisons.

3.3.4 Various Examples

Assume `X`, `Y`, and `Z` and integers in registers `r0`, `r1`, and `r2`, respectively. Observe the following C code.

```
if (X == 0) {
    X = Y + Z;
}
```

What is the equivalent ARM assembly code?

- Generally, when we want to use labels, we want to negate the condition so we can tell the instruction routine to go to a different place in our code. This makes it significantly easier to manage various conditional statements.

```
    cmp     r0, #0       // if (X == 0) {
    bne     false_label  //      .
    add     r0, r1, r2   //      X = Y + Z;
false_label:             // }
    ...
```

Assume `X`, `Y`, and `Z` and integers in registers `r0`, `r1`, and `r2`, respectively. Observe the following C code.

```
if (X == 0) {
    X = Y + Z;
}
else {
    X = Y - Z;
}
```

What is the equivalent ARM assembly code?

- Same idea as the previous example.

```
    cmp     r0,#0       // if (X == 0) {
    bne     else_stat   //      .
    add     r0,r1,r2    //      X = Y + Z;
    b       end_if      // }
else_stat:              // else {
    sub     r0,r1,r2    //      X = Y - Z;
end_if:                 // }
    ...
```

Assume `X`, `Y`, and `Z` and integers in registers `r0`, `r1`, and `r2`, respectively. Observe the following ARM code.

```
    mov     r2,#0
label:
    add     r2,r2,r1
    sub     r0,r0,#1
    cmp     r0,#0
    bne     label
```

What is this equvalent to in C?

- Annotating this code, we have the following:

```
    mov     r2,#0       // int Z = 0;
label:                  // while (true) {
    add     r2,r2,r1    //      Z = Z + 1;
    sub     r0,r0,#1    //      X = X - 1;
    cmp     r0,#0       //      if (X == 0)
    bne     label       //          continue;
                        // }
```

  Or, using a `do...while` loop:

```
int Z = 0;
do {
    Z = Z + 1;
    X = X - 1;
} while (X != 0);
```

4 32-Bit ARM Assembly (Part 2)

In this section, we will continue to discuss ARM assembly, but we will talk more about functions and data transfer.

4.1 Data Transfer

We will briefly review data transfer instructions, which interact with memory and involve:

- Loading a word from memory into a register.
- Storing the contents of a register into a memory word.

4.1.1 Single Register Data Transfer

We make use of the following instructions:

Load Instruction	Store Instruction	Type	Bits
`ldr`	`str`	Word	32
`ldrb`	`strb`	Byte	8
`ldrh`	`strh`	Halfword	16
`ldrsb`		Signed Byte Load	8
`ldrsh`		Signed Halfword Load	16

A few remarks:

- The `ldrsh` instruction loads a half-word and sign-extends it to the 32-bits of the register.
- The `ldrsb` instruction loads a byte and sign-extends it to the 32-bits of the the register.
- The `ldrh` instruction loads a half-word and zero-extends the loaded 16-bits to 32-bits.
- The `ldrb` instruction loads a byte and zero-extends the loaded 8-bits to 32-bits.

The syntax for all of these instructions are as follows:

```
ldr     Rd, <Address>
str     Rd, <Address>
```

Where `Rd` is a register.

4.1.2 Addressing Modes

There are two ways to specify the address in ARM. We call these *addressing modes*.

1. Basic register addressing.
   - Register holds the 32-bit memory address.
   - Also known as the base address.
2. Base displacement addressing mode.
   - An effective address is calculated using the formula:

     ```
     Effective Address = Base Address + Offset
     ```

   - The base address is in a register as before.
   - The offset can be specified in different ways.

4.1.3 Basic Register Addressing Examples

Suppose variable `X` is in register `r1` and variable `Y` is in register `r2`. Given the following ARM instruction:

```
ldr     r2, [r1]
```

What is the C equivalent?

- The answer is `Y = *X;`.

Suppose you have the variable `X` in memory at the address contained in the register `r1`, and `r2` contains the address of `Y`. Given the following C code:

```
Y = &X;
```

How would you write this in ARM?

- The answer is `str r1, [r2]`.

Suppose you have variable `X` in memory at the address contained in `r1`, and `r2` contains the address of `Y`. Given the following C code:

```
Y = *X;
```

How would you write this in ARM?

- The answer is:

```
ldr     r10, [r1]
ldr     r9, [r10]
str     r9, [r2]
```

4.1.4 Base Displacement Addressing Mode

To specify a memory address to copy from, we need to specify two things.

- A register which contains a pointer to memory.
- A numerical offset (in bytes).

The effective memory address is the sum of these two values. For example, given `[r0, #8]`:

- This specifies the memory address pointed to by the value in `r0`, plus 8 bytes.
- `r0` is generally called the base address.
- `#8` is generally called the offset.

This is particularly useful for accessing arrays, as we will see.

4.1.5 Base Displacement Addressing Examples

Given the C code:

```
X = arr[4];
```

Where ***arr*** *is an* ***int*** *array, the base address of* ***arr*** *is in register* ***r1*** *and* ***X*** *is in* ***r2****. What is the assembly equivalent?*

- The answer is `ldr r2, [r1, 16]`. Recall that an integer is 4 bytes. To access the element at index 4, we need to multiply the size of an integer by the index, so `4 * 4 = 16`.

Given the following C code:

```
g = h + A[8];
```

Where `g` is in `r1`, `h` is in `r2`, and the base address of `A` is in `r3`. What is the ARM equivalent?

- The answer is:

```
ldr     r0, [r3, #32]       // r0 = A[4]; (sizeof(int) * 8 = 4 * 8 = 32)
add     r1, r2, r0          // g = h + r0;
```

4.2 Function Calls

Here, we will talk about function calls.

Recall that a **caller** is a function that calls another function; a **callee** is a function that was called. The currently-executing function is a callee, but not a caller.

4.2.1 Register Usage

There are 16 registers that we will work with.

- Registers `r0` through `r3`: Arguments into function. Result(s) from function otherwise corruptible. Additional parameters are passed on stack.
- Registers `r4` through `r11`: Register variables must be preserved. In other words, the value in one of these registers must be the same before and after execution of the function.
  - `r9`: Known as the **stack base** (`sb`).
  - `r10`: Known as the **stack limit** (`sl`).
  - `r11`: Known as the **frame pointer** (`fp`).
- Register `r12`: **Scratch register**, which is corruptible.
- Register `r13`: **Stack pointer** (`sp`).
- Register `r14`: **Link register** (`lr`). This is a register which holds the address to return when a function call completes.[16].
- Register `r15`: **Program counter**. This is a register that indicates where a computer is in its program sequence. Usually, the PC is incremented after fetching an instruction, and holds the memory address of ("points to") the next instruction that would be executed.

[16] `r14` can be used as a temporary once value stacked.

4.2.2 Nested Procedures

Consider the C function:

```
int someFunc(int j, int k) {
    int i, m;
    i = mult(j, k);
    m = mult(i, i) + j;
    return i + m;
}
```

There are a few things to know here.

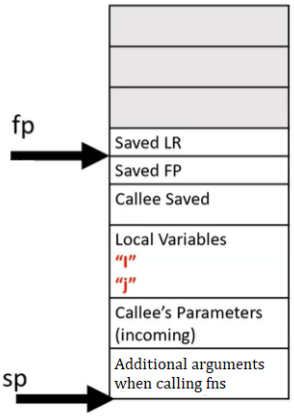
- `r0` will always be the first parameter.
- `r1` will always be the second parameter.
- When a function returns a value, it will always be in `r0`.

In the code shown above, `r0` is `j` as `r1` is `k`. But, once you call the `mult` function, `r0` will be the result of the `mult` function. However, it's not hard to tell that the original parameter value `j` will be overwritten. Thus, you need to save `j` to a local variable, or preserved register (`r4` through `r10`) or you will lose the value.

In general, if you have a parameter value (`r0` through `r3`) that you need to use at some point in the function, save it in a preserved register.

4.2.3 Storing Local Variables on the Stack

When saving things, we need to save it to the stack. Recall that the stack looks something like:



There are a few things to note here.

- If there is a 5th, 6th, or more parameters, it will be put above `Saved LR` (above the stack frame). We can load these parameters using:

```
ldr     Rd, [fp, #4]     // 4 is the location that this
                         // parameter is stored at.
```

- When saving things, we will use the stack frame. The stack frame is bounded by the frame pointer (`fp`) and the stack pointer (`sp`). The frame pointer is like an anchor; it is the beginning of the stack frame. When you call a function, it generates a new stack frame.

- When you get to a function, you should save the link register (`lr`) and the frame pointer (`fp`). In particular, we want to change the frame pointer since the frame pointer was pointing to the frame pointer of the previous function (the one that just called said function).
  - You also want to save any potential preserved register (`r4` through `10`) that you might use.
  - Then, you want to save any local variables.

Generally speaking, for a function, we might have ARM code that looks something like this:

```
    .equ    FP_OFFSET, 12           // Offset to set fp to base of saved regs.
                                    // (# of Saved Regs - 1) * 4
    .equ    LOCAL_VAR_SPACE, 8      // Total number of local variable bytes to
                                    // allocate
    .equ    i_OFFSET, -16           // Local var i offset from fp.
    .equ    j_OFFSET, -20           // Local var j offset from fp.
    .equ    PARAM_SPACE, 16         // Total number of formal parameter bytes to
                                    // allocate.
    .equ    PARAM1_OFFSET0, -24     // 1st format param offset from fp.
    .equ    PARAM2_OFFSET1, -28     // 2nd formal param offset from fp.

fnEntry:
    push    {r4-r5, fp, lr}         // Saving just 4 registers.
    add     fp, sp, #FP_OFFSET      // (# of saved registers - 1) * 4
    sub     sp, sp, #(LOCAL_VAR_SPACE + PARAM_SPACE)
    mov     r4, #10
    str     r4, [fp, #i_OFFSET]
    mov     r4, #20
    str     r4, [fp, #j_OFFSET]

    ldr     r0, [fp, #i_OFFSET]     // i = i + j;
    ldr     r4, [fp, #j_OFFSET]     // .
    add     r0, r0, r4              // .

    sub     sp, fp, #FP_OFFSET
    pop     {r4, fp, lr}
    bx      lr
```

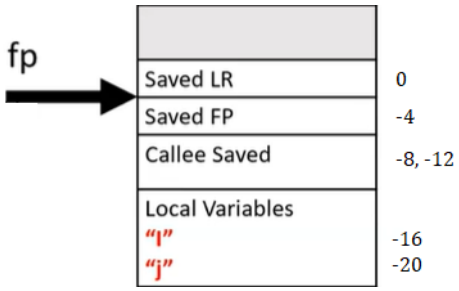
A few things to consider. Here, `fp` means frame pointer and `sp` means stack pointer. We are saving the `fp` (frame pointer) of the previous function. The linked register only needs to be saved if it will be altered in this function (say, if we call another function in this function).

- `FP_OFFSET` refers to the number of registers that we are saving. The number of registers that we are saving can be found in the `push` instruction. This does not allocate any bytes yet. The formula for this variable is:

$$(\text{Number of Saved Registers } - 1) \cdot 4$$

- `LOCAL_VAR_SPACE` are the total number of local variable bytes to allocate. Since we have `i` and `j` (`r4` and `r5`), we need to allocate 8 bytes.
- `i_OFFSET` refers to the local variable `i` offset from `fp`. `j_OFFSET` refers to the local variable `j` offset from `fp`. We got these values because:
  - `Saved LR` is at offset `0`.
  - `Saved FP` is at offset `-4`.

- `Callee Saved`[17] is at offset `-8` and `-12` (one for each local variable, if any). For every register you push (using the `push` instruction), you should make a note here.
- `Local Variable i` is at offset `-16` and `j` is at `-20`.



- `PARAM_SPACE` refers to the number of formal parameter bytes to allocate. This is where you can store the incoming parameters. This refers to `r0-r3`.
- `PARAM1_OFFSET0` and `PARAM2_OFFSET1` both follow from the local variable offsets. You use these if you want to save the parameters.

4.2.4 Accessing Local Variables

Consider the C function:

```
int num(int a, int b) {
    int c;
    int d;
    c = a + b;
    d = a - b;
}
```

And the preamble:

```
.equ    NUL, 0
.equ    FP_OFFSET, 4
.equ    LOCAL_VAR_SPACE, 8
.equ    C_OFFSET, -8
.equ    D_OFFSET, -12
.equ    PARAM_SPACE, 8
.equ    A_OFFSET, -16
.equ    B_OFFSET, -20
```

Here, we demonstrate how local variables can be accessed:

```
push    {fp, lr}
add     fp, sp, #FP_OFFSET
sub     sp, sp, #20
str     r0, [fp, #A_OFFSET]         // c = a + b;
str     r1, [fp, #B_OFFSET]         // .
ldr     r2, [fp, #A_OFFSET]         // .
```

[17] There is a pretty good explanation on StackOverflow. It reads: ***Callee-saved registers*** *(AKA non-volatile registers, or call-preserved) are used to hold long-lived values that should be preserved across calls. When the caller makes a procedure call, it can expect that those registers will hold the same value after the callee returns, making it the responsibility of the callee to save them and restore them before returning to the caller. Or to not touch them. By contrast,* ***Caller-saved registers*** *(AKA volatile registers, or call-clobbered) are used to hold temporary quantities that need not be preserved across calls. For that reason, it is the caller's responsibility to push these registers onto the stack or copy them somewhere else if it wants to restore this value after a procedure call.*

```
    ldr     r3, [fp, #B_OFFSET]         // .
    add     r3, r2, r3                  // .
    str     r3, [fp, #C_OFFSET]         // .
    ldr     r2, [fp, #A_OFFSET]         // d = a - b;
    ldr     r3, [fp, #B_OFFSET]         // .
    sub     r3, r2, r3                  // .
    str     r3, [fp, #D_OFFSET]         // .
    nop                                 // .
    mov     r0, r3                      // return d;
    sub     sp, fp, #FP_OFFSET          // .
    pop     {fp, lr}                    // .
    bx      lr                          // .
```

4.3 Recursion

Now, we will talk about recursion.

4.3.1 Recursion in C and ARM: Example

Consider the factorial function:

```
fact:
    push    {r4, r5, fp, lr}
    cmp     r0, #1              // if (n > 1) {
    ble     ret_one             //      .
    mov     r4, r0              //      r4 = r0;
    sub     r0, r0, #1          //      r0 = r0 - 1;
    bl      fact                //      r0 = r4 * fact(r0);
    mul     r0, r0, r4          //      .
    b       end                 // }
ret_one:                        // else {
    mov     r0, #1              //      r0 = 1;
end:                            // }
    pop     {r4, r5, fp, lr}
    bx      lr                  // return r0;
main:
    mov     r0, #1
    bl      fact
    mov     r2, r0
```

Remarks:

- If we didn't save the callee-saved registers, we would have gotten into trouble. This is because we kept clobbering `r4`. This was because every call to `fact` needs its own value of `r4`, or more precisely its own stack frame.
- Additionally, if we didn't save the `lr` (link register), we would end up with an infinite loop since every call to `fact` clobbers the old version of `lr`.

5 Digital Logic

Now, we will briefly go over digital logic. In particular, we will review combinational logic, timing diagrams, machine code, and more.

5.1 Logic Gates

There are a few logic gates to remember.

5.1.1 AND Gates

An AND Gate is a basic logic gate which provides `1` once all of the inputs to the AND Gate are `1`. If any of the input is `0`, than its output is `0`. It is also referred to as arithmetic multiply operation. It can be represented like so:

Here are some common representations of `AND`.

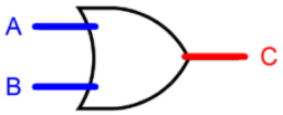
- $C = A \ \& \ B$
- $C = AB$
- $C = A.B$

And the truth table is as follows:

A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

5.1.2 OR Gates

A logical gate which gives `0` if only and only when all of the inputs are low state `0`. Otherwise, it will give `1` as the output. It can be represented like so:



Some ways to represent `OR`:

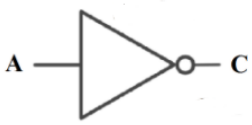
- $C = A \mid B$
- $C = A + B$

And the truth table:

A	B	C
0	0	0
0	1	1
1	0	1
1	1	1

5.1.3 NOT Gates

A logical NOT gate takes `0` or `1`, and produces `1` and `0` respectively as an output. It can be represented like so:



Some ways to represent `NOT`:

- $C = \sim A$
- $C = A'$

And the truth table:

A	C
0	1
1	0

5.1.4 NAND Gate

A logical NAND operates as an AND gate followed by a NOT gate. It can be represented like so (notice the circle at the tip):

Some ways to represent `NAND`:

- $\sim (ab)$
- $(ab)'$

And the truth table is as follows:

A	B	C
0	0	1
0	1	1
1	0	1
1	1	0

5.1.5 NOR Gate

A logical NOR operates as an OR gate followed by a NOT gate. It can be represented like so:

Some ways to represent `NOR`:

- $\sim (a|b)$
- $\sim (a + b)$

- $(a + b)'$

And the truth table is as follows:

A	B	C
0	0	1
0	1	0
1	0	0
1	1	0

5.1.6 XOR Gate

A logical gate which gives `1` if one of two inputs is `1` and `0` otherwise. It can be represented like so:

Some ways to represent `XOR`:

- $a \wedge b$
- $a \oplus b$

And the truth table is as follows:

A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

5.1.7 XNOR Gate

A logical gate operates as a XOR gate followed by a NOT gate. It can be represented like so:

Some ways to represent `XOR`:

- $\sim (a \wedge b)$
- $(a \oplus b)'$

And the truth table is as follows:

A	B	C
0	0	1
0	1	0
1	0	0
1	1	1

5.1.8 Boolean Laws

You can use these to help in simplifying boolean expressions.

- Commutative Property

$$A + B = B + A$$
$$AB = BA$$

- Associative Property

$$A + (B + C) = (A + B) + C$$
$$A(BC) = (AB)C$$

- Distributive Property

$$A(B + C) = AB + AC$$
$$A + BC = (A + B)(A + C)$$

- DeMorgan's Law

$$(A + B)' = A'B'$$
$$(AB)' = A' + B'$$

- Other Properties

$$A + A'B = A + B$$
$$A + A' = 1$$

5.2 Timing Diagrams

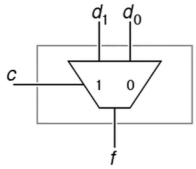
- In the real world, we represent `0` as no voltage and `1` as having voltage.
- Gates have a delay. Rather than instantly computing `A` and `B`, it takes some times for it to compute.
- This can be represented with timing diagrams.

Consider this timing diagram:

- The bottom-most line represents `0` and the top-most line represents `1`.
- The line connecting the bottom to the top, known as a rising edge or rise time, is called a positive edge.
- The line connecting the top to the bottom, known as a falling edge or fall time, is called a negative edge.

5.3 Multiplexer (Mux)

A simple 2-1 mux, like the one shown below, takes in two bits and produces one bit, depending on the way the third input is set.



The truth table for a 2-1 mux is:

C	f
0	d_0
1	d_1

We can represent this truth table by:

$$f = \begin{cases} d_1 & C = 1 \\ d_0 & C = 0 \end{cases}$$

So:

- f should be the value of d_1 whenever C is 1. So, $f = c \cdot d_1$.
- f should be the value of d_0 whenever C is 0. So, $f =\sim C \cdot d_0$.

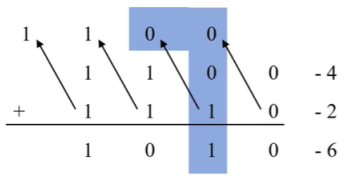
The expression and its corresponding truth table is:

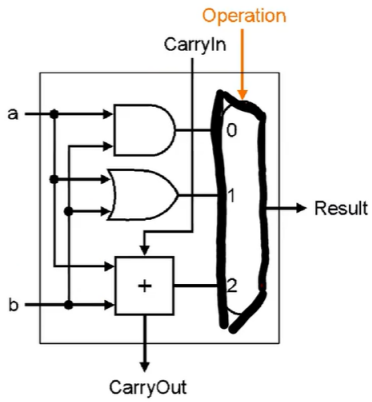
$$f = C \cdot d_1 \mid\sim C \cdot d_0 \iff f = Cd_1 + C'd_0$$

C	d_0	d_1	f
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

5.4 Arithmetic Logic Unit (ALU)

A one-bit ALU looks like this:

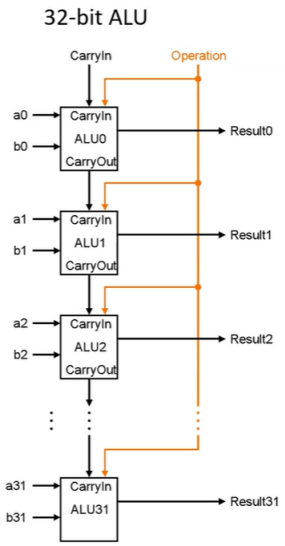




This ALU can perform `ADD`, `OR`, and the `ADD` operations.

5.4.1 32-Bit ALU

Suppose we wanted a 32-bit ALU. Then, we only need to chain the above one-bit ALU 32 times. That is:



The carry-out of bit 0 goes to the carry-in of bit 1, the carry-out of bit 1 goes to the carry-in of bit 2, and so on.

5.4.2 Example Questions

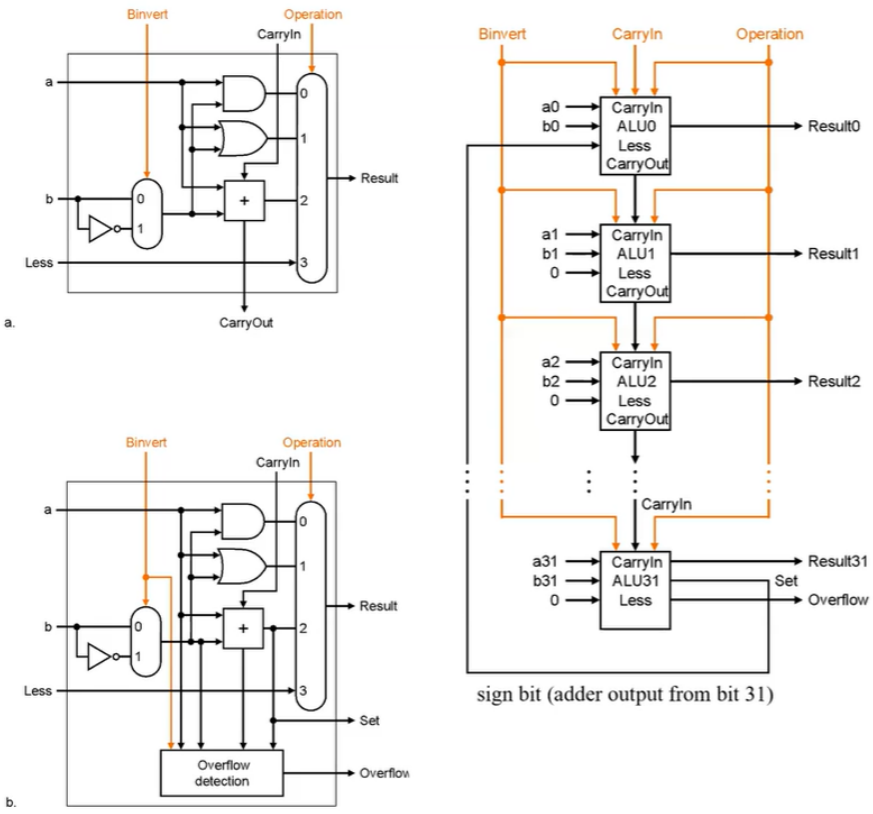
We'd like to implement a means of doing $A - B$ (subtraction) but with only minor changes to the above hardware. How can do achieve this?

- Recall that:

  $$A - B = A + (-B)$$

  Our goal is to perform *twos-complement* on B so we can get the desired result. Thus, we need to invert B and then add one to the result. To do this, we need to provide an option to use bitwise `NOT B` and then add one. Thus, we need to provide an option to use `1` instead of the first carry-in.

Given the full ALU below, what signals accomplish `ADD`?



	Binvert	Carry-In	Operation
A	1	0	0
B	0	1	1
C	1	1	0
D	0	0	1

- The answer is `B`. First, we note that the `OR` gate is defined by operation `1`, so the answers either have to be `B` or `D`. Now, recall that `OR` can be represented by $A + B$. In this sense, it follows that we do *not* want to invert B (which is what answer `D` is saying). Thus, the answer is `B`.

Given the full ALU above, which signals accomplish `SUB`?

	Binvert	Carry-In	Operation
A	1	0	2
B	0	1	2
C	1	1	2
D	0	0	2

- The answer is `C`. Recall that $A - B$ is equivalent to $A + (-B)$. Let's use an example. Suppose we wanted to do $7 - 1$. This is the same as $7 + (-1)$. Of course, we need to take the two's complement of 1 so we can get -1, thus performing $A + (-B)$. In other words:

```
0001 (1) -> 1110 + 1 = 1111 (-1)
```

Continuing on:

```
    7       0111
+ (-1)    + 1111
----------------
    6      10110
```

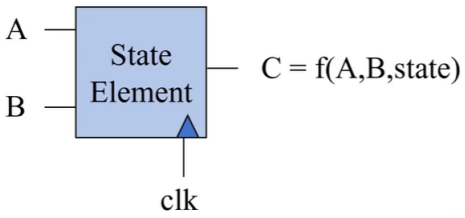
Discarding the extra bit, we have `0110`, which is exactly `6`.

Of course, what we just did is exactly the same as inverting B (or simply, B') and then adding 1, and *then* doing $A + (B' + 1)$. Thus, `Binvert` is `1` and `Carry-In` is `1`.

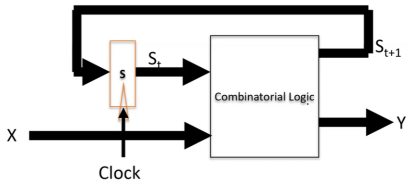
5.5 Sequential Logic

With combinational logic, we generate a bunch of inputs and generate an output. However, this doesn't actually remember the last thing that we computed.

We're going to add a **state element**, which allows us to save some values so we can use them later. This looks something like the diagram below[18]. We will use a clock to distinguish between the different phases of the logic.



One main thing we talked about was the idea of the **sequential network**. A sequential network looks something like:



It has combinational logic (which computes something and then saves it) and state elements (which will save the state).

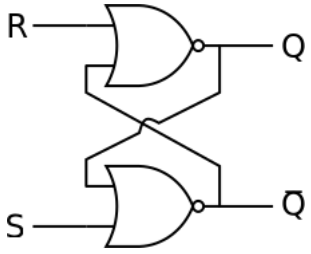
For every clock cycle:

- Use combinational logic to compute something given an input X.
- Output that result. This is simply:
$$Y = f(S_t, X)$$
- Save the result into S_{t+1}. This is:
$$S_{t+1} = g(S_t, X)$$

[18] `clk` is the clock.

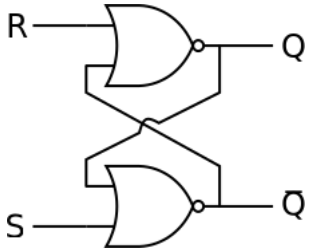
5.5.1 Set/Reset Latch (S-R Latch)

The SR latch has two inputs Set and Reset, and another two inputs Q and $\overline{Q}$ (the inverse of Q) and can be constructed from a pair of `NOR` gates. It looks like:



5.5.2 Example Questions

Consider the following diagram:



Recall that $\overline{Q} = Q' = \sim Q$. *Suppose* $S = 1$ *and* $R = 0$. *What will* Q *be?*

Selection	Q
A	0
B	1
C	Q from before.
D	Q' from before.
E	None of the above.

- The answer is `B`. For S(Set), we are given `1`. Although we don't know the value of the other input, we know that $(1+b)'$ (or 1 NOR b) will always be `0`. With this in mind, we know that the gate associated with S(Set) will output `0`. Thus, we now know that R(Reset) will have inputs `0` (given) and `0` (which we got from S(Set)). Since $(0+0)'$ (or 0 NOR 0) is `1`, we know that Q will be `1`.

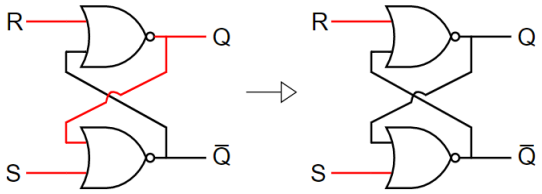
Using the diagram and answer choices from the previous question, and suppose $S = R = 0$. *What will* Q *be?*

- Let's make a few assumptions.
  - Suppose $Q = 1$. Then, R(Reset) will output `1`. This means that the bottom gates will be given two inputs: `0` (given) and `1` (from the top gate). We know that $(0+1)'$ (0 NOR 1) will produce `0`, so the bottom gate will output `0`. Then, going to the top gate, we have inputs `0` (given) and `0` (from the bottom gate), which gives us `1`.
  - Suppose $Q = 0$. Then, R(Reset) will output `0`. This means that the bottom gates will be given inputs `0` (given) and `0` (from top gate). Since 0 NOR 0 is `1`, it follows that the bottom gate will output `1`. Then, going back to the top gate, we have inputs `0` (given) and `1` (from bottom gate), which gives us `0`.

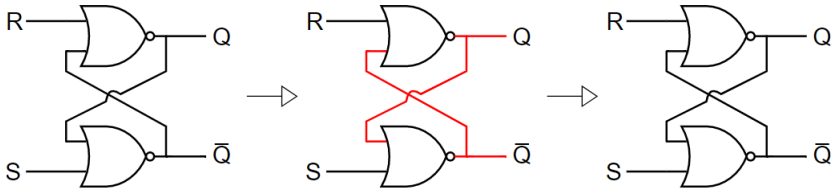
  In both cases, we have showed that for a given input Q, we will always get the same Q. Thus, the answer is `C`.

5.5.3 Issues with the Set/Reset Latch

What if $S = R = 1$? Then, $Q = \overline{Q} = 0$, which violates the idea that $\overline{Q}$ is the inverse of Q.

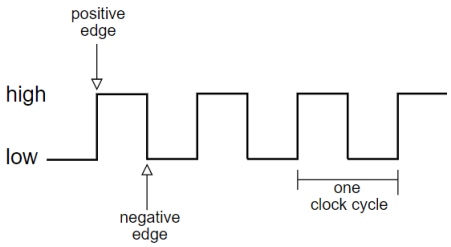


This also creates another issue. That is, once we set $S = R = 0$ at the same time, both `NOR` gates will try to output a `1` at the same time, which will cause both `NOR` gates to output `0`s. In this sense, the S-R latch will keep oscillating back and forth until one `NOR` gate "wins" due to physical differences. This leaves the S-R latch in an unknown issue.



5.5.4 Clock Signals

One way to resolve this issue is through the idea of a clock. In a computer, a clock is a thing that provides a synchronization point so that as a computer is changing from one state to another, it is done based on a *clock cycle*. So, really, a clock is just a periodic signal that oscillates.

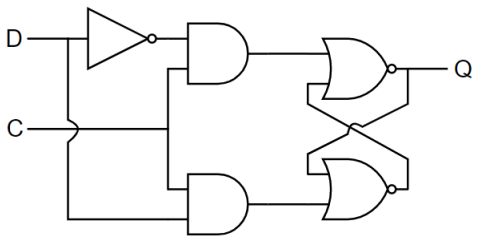


The clock cycle time, also known as the *period*, is the time between positive edges of a clock signal. This is defined by:

$$\text{Frequency} = \frac{1}{\text{Period}}$$

5.5.5 Back to the S-R Latch

We now need to synchronize our S-R latch. To do this, we'll add a clock.



Where our inputs are:

- D, the data.
- C, the clock.

Here, we have a D-latch. When C is low, both inputs to our S-R latch are low[19], meaning the latch holds its state no matter what is happening to D. When C is high, Q gets set to D's current state.

To summarize, we have:

`C`	`D`	Next State of `Q`
0	X	No Change
1	0	`Q = 0`; Reset State
1	1	`Q = 1`; Set State

- When $C = 1$, the latch is "transparent"; $Q = D$.
- When $C = 0$, Q is latched to the last value of D.

TODO

5.6 Machine Code

We want to gain an understanding of machine code so we can bridge the gap between what we know and what the machine knows.

5.6.1 Data Processing Instruction

To represent data processing instructions in machine code, we make use of this format:

31 30 29 28	27	26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
Cond	0	0	I	Opcode	S	Rn	Rd	Operand 2

Where:

- `Cond` - Does this instruction execute conditionally? In this class, this will always be `1110`.[20]
- `I` - Is operand 2 a register or immediate?
  - `0` if operand 2 is a register.
  - `1` if operand 2 is an immediate.
- `Opcode` - What data processing instruction are we using?

[19] Low means `0`; high means `1`.
[20] In CSE 30, we don't care about this.

Opcode	Operation	Action
0000	AND	Op1 AND Op2
0001	EOR	Op1 EOR Op2
0010	SUB	Op1 - Op2
0011	RSB	Op2 - Op1
0100	ADD	Op1 + Op2
0101	ADC	Op1 + Op2 + Carry
0110	SBC	Op1 - Op2 + Carry - 1
0111	RSC	Op2 - Op1 + Carry - 1
1000	TST	Set Condition Code on Op1 AND Op2
1001	TEQ	Set Condition Code on Op1 EOR Op2
1010	CMP	Set Condition Code on Op1 - Op2
1011	CMN	Set Condition Code on Op1 + Op2
1100	ORR	Op1 OR Op2
1101	MOV	Op2
1110	BIC	Op1 AND NOT Op2
1111	MVN	NOT Op2

- `S` - Whether this will set condition codes. In CSE 30, only `cmp` and `tst` set condition codes.
  - `0` if we are not altering condition codes.
  - `1` if we are setting condition codes.
- `Rn` - The first operand register.
- `Rd` - The destination register.
- `Operand 2` - This depends.
  - If we are working with a register (`I` is `0`), then[21]:

11 10 9 8 7 6 5 4	3 2 1 0
00000000	Operand 2

  - If we are working with an immediate (`I` is `1`), then:

11 10 9 8	7 6 5 4 3 2 1 0
Rotate	Imm

    Where `Rotate` is the shift applied to `Imm` and `Imm` is an unsigned 8-bit immediate value. In this class, assume that `Rotate` is `0000`. That is:

11 10 9 8	7 6 5 4 3 2 1 0
0000	Imm

As an example, suppose we wanted to find the corresponding machine code for `add r0, r0, r1`. Then, we know that:

- `Cond` is `1110`.
- `I` is `0` (we're working with a register).
- `OpCode` is `0100`.
- `S` is `0` (we aren't using `cmp` or `tst`).
- `Rn` (which is `r0`, or the middle register) is `0000`.
- `Rd` (which is `r0`, or the left-most register) is `0000`.

[21] In CSE 30, the shift will be 0.

- `Operand 2` is `00000000 0001`.

So, in binary, the corresponding machine code is:

```
1110 000 0100 0 0000 0000 00000000 0001
0xe08000
```

5.6.2 Single Data Transfer Instruction

To represent single data transfer in machine code, we make use of this format:

31 30 29 28	27 26	25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
Cond	01	I	P	U	B	W	L	Rn	Rd	Offset

Where:

- `Cond` - The condition field.

cond	Mnemonic	Name	CondEx
0000	EQ	Equal	Z
0001	NE	Not Equal	$\overline{Z}$
0010	CS/HS	Carry Set/Unsigned Higher or Same	C
0011	CC/LO	Carry Clear/Unsigned Lower	$\overline{C}$
0100	MI	Minus/Negative	N
0101	PL	Plus/Positive or Zero	$\overline{N}$
0110	VS	Overflow/Overflow Set	V
0111	VC	No Overflow/Overflow Clear	$\overline{V}$
1000	HI	Unsigned Higher	$\overline{Z}C$
1001	LS	Unsigned Lower or Same	Z or $\overline{C}$
1010	GE	Signed Greater Than or Equal	$\overline{N \oplus V}$
1011	LT	Signed Less Than	$N \oplus V$
1100	GT	Signed Greater Than	$\overline{Z}(\overline{N \oplus V})$
1101	LE	Signed Less Than or Equal	Z or $N \oplus V$
1110	AL	Always/Unconditional	Ignored

- `I` - Whether the offset is is a immediate value (`0`) or a register (`1`).
- `P U B W L` - Various bits.

Bit	T	U
0	Immediate Offset in `Offset`	Subtract Offset from Base
1	Register Offset in `Offset`	Add Offset to Base.

P	W	Index Mode
0	0	Post-Index
0	1	Not Supported
1	0	Offset
1	1	Pre-Index

L	B	Instruction
0	0	`STR`
0	1	`STRB`
1	0	`LDR`
1	1	`LDRB`

Mode	ARM Assembly	Address	Base Register
Offset	`LDR R0, [R1, R2]`	`R1 + R2`	Unchanged
Pre-Index	`LDR R0, [R1, R2]!`	`R1 + R2`	`R1 = R1 + R2`
Post-Index	`LDR R0, [R1], R2`	`R1`	`R1 = R1 + R2`

- `Rn` - The base operand register.
- `Rd` - The source/destination register.
- `Offset` - This depends.
  - If we are working with a register (`I` is `0`), then:

11 10 9 8 7 6 5 4 3 2 1 0
The Immediate Offset

  - If we are working with an immediate (`I` is `1`), then:

11 10 9 8 7 6 5 4	3 2 1 0
Shift	`Rm`

    Where `Rm` is the offset register.

As an example, suppose we are given the following:

```
1110 01 1 1 1 0 0 1 0010 0100 0000 0000 0011
```

- We know that:
  - `I` is `1`, so we're working with a register.
  - `P` is `1` and `W` is `0`, so we're working with an offset.
  - `U` is `1`, so we're adding the offset to the base.
  - `L` is `1` and `B` is `0`, so we're working with `LDR`.
  - `Rn` is `0010`, so the base operand register is `r2`.
  - `Rd` is `0100`, so the source/destination register is `r4`.
  - Since `I` is `1`, we know that `Rm` is `0011`, so the offset register is `r3`.

  So, the answer is:

```
ldr     r4, [r2, r3]
```

5.6.3 Branch Instruction

To represent branch instructions in machine code, we make use of this format:

31 30 29 28	27 26 25	24	23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Cond	101	L	Offset

Where:

- `Cond` - The condition field. Refer to the table in the previous page.
- `L` - The link bit.
  - If this is just a branch, this is `0`.
  - If this is a branch with link, this is `1`.

  The branch instructions contain a signed twos-complement 24-bit offset. This is shifted left two bits, sign-extended to 32-bits, and added to the `PC`. The branch offset must take account of the pre-fetch operation, which causes the PC to be 2 words[22] (8 bytes) ahead of the current instruction.

[22] 1 word is 4 bytes.

For example, suppose we have the following ARM code:

```
8004:       cmp     r2, r3
8008:       blt     label           // ?
8012:       add     r0, r0, r1
8016:       sub     r0, r0, r3
8020:       str     r0, [r4]
8024:       b       end
8028:   label:                      // ?
8032:       add     r0, r0, r2
```

What should the offset (in decimal) be for the bit instruction?

- We are interested in the branch instruction and the associated label. We know that the branch is at `8008`, which is the `PC`. But, the program counter must be 2 words ahead of the current instruction, so `PC` is at `8016`. The target label is at `8028`, so $8028 - 8016 = 12$ bytes. But, since 12 bytes is 3 words, the answer is 3.