CS1 Lab 5: Bit Operations and Cache Calculations

In the previous lab, you will explored timing on the Arduino, and how to make the LEDs blink specific patterns. In this lab, you will explore different bit-wise operations on the Arduino. You will also write a small assembly program to calculate the various components of a memory address with regards to a specific cache memory architecture.

1 Getting Started

Make sure you have everything you need to complete this lab. These items should be included in your CS1 Lab Kit. You will need to wire up all 8 LEDs supplied in your kit. If you are missing a resistor or LED, ask your TA for replacements.

- Arduino Nano.
- USB cable.
- Atmel ATmega328 Datasheet.
- Atmel 8-bit AVR Instruction Set Manual.
- Arduino Schematic Circuit Diagram.
- Breadboard.
- 8 LEDs.
- 8 330Ω resistors (color coded: Orange-Orange-Black-Black-Brown).
- 9 jumper wires.

2 Pre-Lab: Wiring up 8 LEDs

For this lab we would like to connect all 8 LEDs to the Arduino so that we can display an entire byte of data. In Lab 2, you connected `PORTB` to 4 LEDs to display the lower 4 bits of a byte. Take a look at the Arduino schematic to see if it is possible to connect the upper 4 bits of `PORTB` (i.e. `PB4` to `PB7`) to LEDs.

Pre-Lab Question 1 What prevents us from connecting the upper 4 bits of `PORTB` to external LEDs?

What about using `PORTC` or `PORTD`? Take a look at the Arduino schematic again at `PORTC` . You will find that it does not have a full 8 bits to connect to LEDs. `PORTD` has all 8 bits, but its pins `PD0` and `PD1` are used for the communication over the USB cable while programming the Arduino. Given these constraints, how would we add another 4 bits of output to have 8 LEDs connected to the Arduino?

One solution is to use the upper 4 bits of `PORTD` (i.e. `PD4` to `PD7`) to display the upper 4 bits of the byte that we wish to display. This way, we can still use the lower 4 bits of `PORTB` (i.e. `PB0` to `PB3`) to display the lower 4 bits of the byte.

See the table below for how the LEDs are connected to the Ports Pins:

LED bit	PORT PIN	Arduino Pin
0	PB0	D8
1	PB1	D9
2	PB2	D10
3	PB3	D11
4	PD4	___
5	PD5	___
6	PD6	___
7	PD7	___

Pre-Lab Question 2 Assuming this set-up above, what pins from `PORTD` on the Arduino do we have to connect to the LEDs upper 4 bits?

3 Pre-Lab: Bit Shifting

A common set of operations in assembly are called *bit shifting* operations (`https://en.wikipedia.org/wiki/Bitwise_operation#Bit_shifts`). Bit shifting is where all of the bits in a binary number are moved one bit position to the *left* or to the *right*.

Using the *instruction set* documentation, look up the left bit shifting `LSL` instruction, and consider the code listing below:

```
ldi r16, 0x13
lsl r16
```

Listing 1: Bit shifting with `LSL`

Pre-Lab Question 3 After the `LSL` instruction, what are the contents of the register **r16**?

Pre-Lab Question 4 Effectively, what mathematical operation occurs in bit-shifting to the *left*?

Now, look up the right bit shifting instruction, `LSR` in the *instruction set* documentation and consider the code below:

```
ldi r16, 0x13
lsr r16
```

Listing 2: Bit shifting with `LSR`

Pre-Lab Question 5 After the `LSR` instruction, what are the contents of the register **r16**?

Pre-Lab Question 6 Effectively, what mathematical operation occurs in bit-shifting to the *right*?

4 Pre-Lab: Bit Masking with bitwise AND

Another set of binary operators commonly used in assembly are *bitwise operators* (`https://en.wikipedia.org/wiki/Bitwise_operation`). Bitwise operations operate on the individual bits of one or more binary numbers. When applying a bitwise operation on two equal-length binary numbers, the corresponding bits from same bit positions in both numbers are evaluated with the operator.

One operator is the *logical AND* operator whose effective operation is multiplying the bits from to binary numbers with each other. In the table below, the bits in the first row are *AND*'d with the bits in the second row to produce the resulting third row.

	0	0	1	1
AND	0	1	0	1
Result	0	0	0	1

You can see that a 1 exists in the results only when *both* bits being *AND*'d are also 1.

AND operations are useful for *bit masking* operations. Bit masking is where you want to isolate a portion of bits in a binary number, and ignore the other bits by setting those bits to 0. We do this by using one binary number as a *bit mask* where we *set* the bits to 1 for the bit position we want to isolate. Then, by *AND*ing the bit mask with another binary number, the resulting byte contains only the isolated bit positions.

Using the *instruction set* documentation for the AND assembly instruction, consider this code:

```
1   ldi r16, 0x38
2   ldi r17, 0x2A
3   and r17, r16
```

Listing 3: Bit masking with logical AND

Pre-Lab Question 7 After the AND instruction in Line 3, what are the contents of the register **r16**?

Pre-Lab Question 8 After the AND instruction in Line 3, what are the contents of the register **r17**?

Pre-Lab Question 9 If we consider the binary number in **r16** as a *bit mask*, then what bit positions in **r17** are being isolated and remain in **r17** as a result of the above code?

5 Pre-Lab: Bit Composition with bit-wise OR

Another bitwise operator that is useful to use is the *logical OR* operator, where the results of each bit position is 0 if both bits are 0, while otherwise the result is a 1. In the table below, the bits in the first row are *OR*'d with the bits in the second row to produce the resulting third row.

	0	0	1	1
OR	0	1	0	1
Result	0	1	1	1

You can see that 0 exists in the results only when *both* bits being *OR*'d are also 0.

OR operations are useful for composing binary from parts of other binary numbers when used with bit masking. Using the instruction set documentation for the *OR* operation, consider this code:

```
1   ldi r16, 0x38
2   ldi r17, 0x2A
3   ldi r18, 0x0F
4   and r16, r18
5   ldi r18, 0xF0
6   and r17, r18
7   or r16, r17
```

Listing 4: Bit masking with logical OR

Pre-Lab Question 10 *BEFORE* the OR instruction in Line 7 executes, what are the contents of **r16**?

Pre-Lab Question 11 *BEFORE* the OR instruction in Line 7 executes, what are the contents of **r17**?

Pre-Lab Question 12 *AFTER* the OR instruction in Line 7 executes, what are the contents of **r16**?

You can see that after the AND instruction in Line 4, the lower 4 bits of **r16** remain (i.e the upper 4 bits are all 0). After the AND instruction in Line 6, the upper 4 bits of **r17** remain (i.e. the lower 4 bits are all 0). With the OR instruction in Line 7, these two parts are combined into a single binary number and is saved in register **r16**.

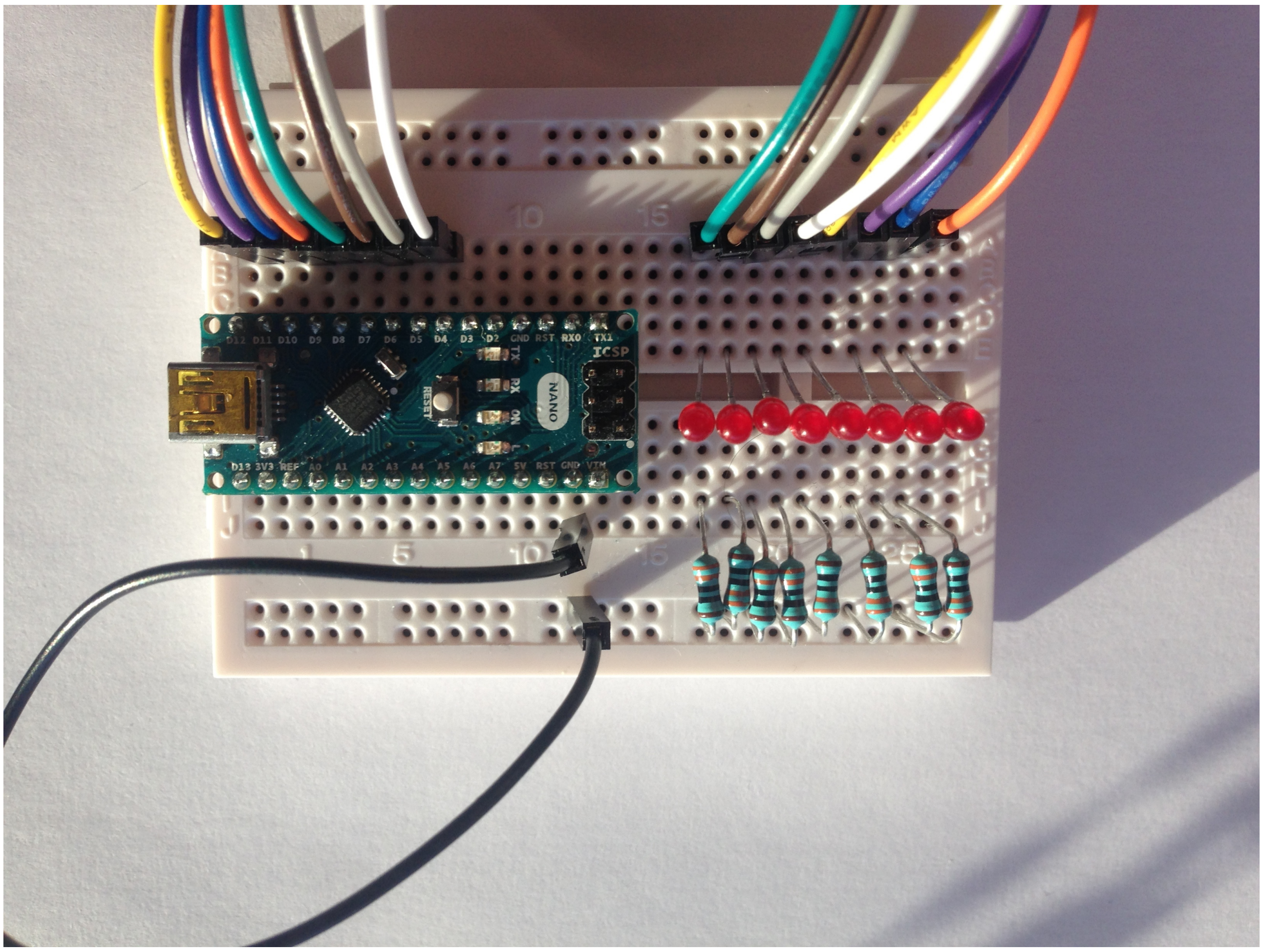
You will use *bit shifting*, the *logical AND* and *logical OR* operator in your Lab 5 session to manipulate bits in the Arduino registers. Other commonly used bitwise operators exist on the Arduino, and you may be interested in looking up the *NOT* and *XOR* instructions in the *instruction set* documentation.

Use the answers from these **Pre-Lab Questions** to complete the Pre-Lab 5 Quiz on KEATs. These Pre-Lab Questions will aid you in the completion of the lab in your lab session. Be sure to submit your Pre-Lab Quiz *before* the deadline!

6 Lab: Wire up 8 LEDs to output a single byte

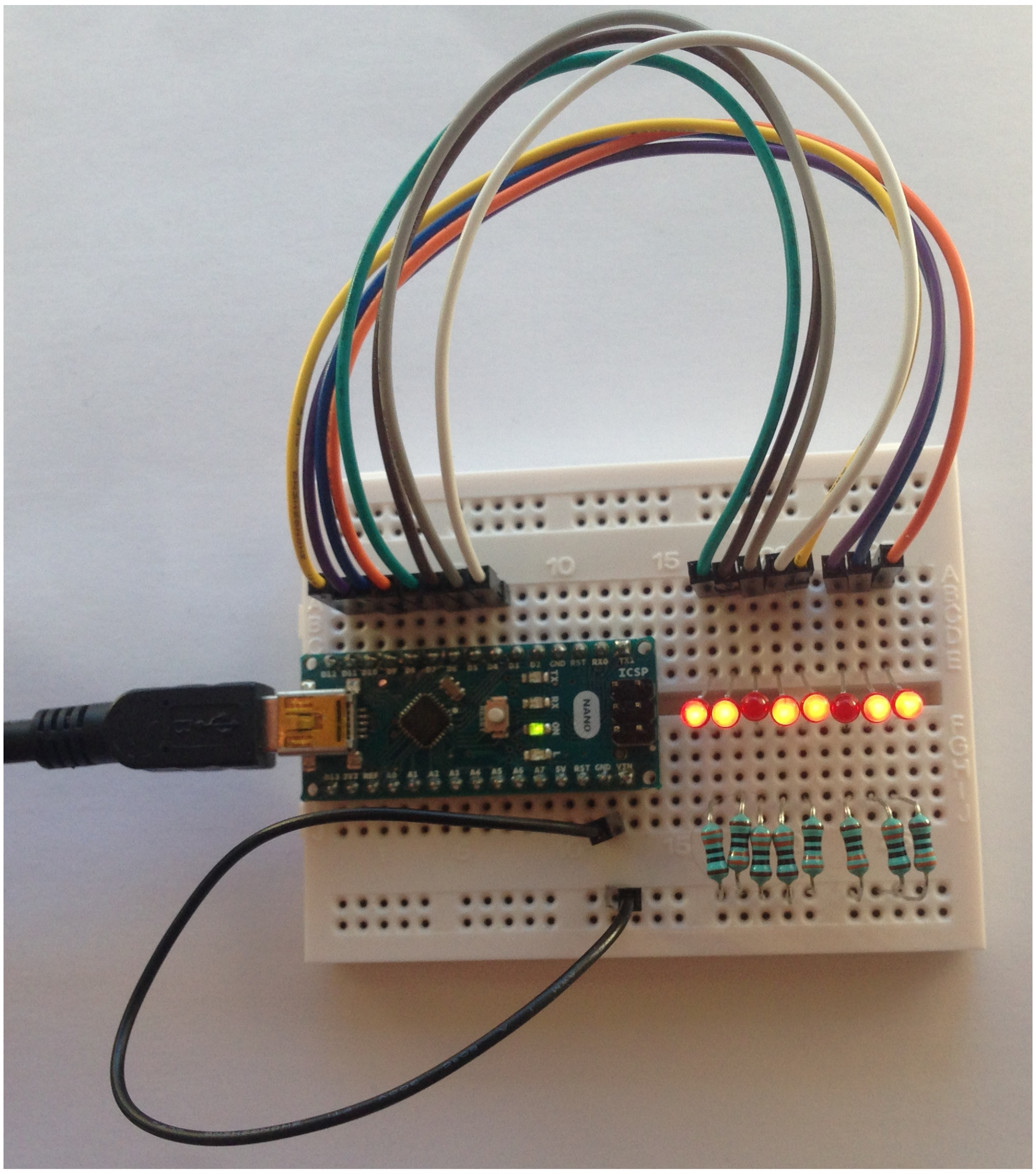
Before we start programming the Arduino, we need to wire up the 8 LEDs to the appropriate pins to PORTB and PORTD as described in the Pre-Lab Section 2. You will find the set-up similar to that of wiring PORTB from Lab 2.

As you can see in the photo below, 8 LEDs are lined up next to each other, so that a single byte of data can be read from the output. They are wired such that bit 0 of the byte output is the **right**-most LED, and bit 7 is the **left**-most LED.



Now that you have 8 LEDs wired up to the Arduino, let us write a program that outputs a single byte, hexadeciamal 0xDB, to test the wiring set-up on the LEDs.

1. Using a copy of a previous program from Lab 2, create a new file and name it `write_DB.s`. Your program should do all of the set-up that is required to write output to `PORTB` and `PORTD` , in particular:
   (a) Declare the appropriate code symbols with `.eq`
   (b) Clear the `SREG` register at the beginning of the program.
   (c) Set-up the *lower 4 bits* of `PORTB` for **output** with the `DDRB` register
   (d) Set-up the *upper 4 bits* of `PORTD` for **output** with the `DDRD` register
2. Add code to your program that loads the value that we want to output (hexadecimal `0xDB`) into a register.
3. Write out the value from the register to **both** `PORTB` and `PORTD` using separate `OUT` instructions. This should in effect display the value on the LEDs with the pattern in the photo below:



4. Make sure that you are able to correctly write the hexadecimal `0xDB` value out to the LEDs before moving on to the next part.
5. Think of a new byte value to write out the LEDs, and test a variation of your assembly program with that value to make sure that you get the LED pattern that you would expect. Remember, for any

value you wish to display on the LEDs, you will have to write it to both PORTB and PORTD as the LEDs are connected to different bits of both ports.

Congratulations! You have successfully written a byte out to the LEDs! Make sure to demonstrate your code to a TA for any feedback.

7 Lab: Bit shifting

Now that you can successfully display a byte value, let us write a program that uses the *bit shifting* operators LSL and LSR we explored in the Pre-Lab.

1. Copy your program from the previous Section 6 and call the new file bit_shifting.s.
2. Start by loading the hexadecimal value 0x55 into a register. Display that value on the LEDs like you did with your write_DB.s program.
3. Test your program to make sure that you can read the value 0x55 on the LEDs. You should see a bit pattern where every other LED is turned on.
4. Add code to your program to bit shift this value in the register 1 bit position to the *left*. Think about which operator you would use to do a left bit shift operation. Display this new value out on the LEDs.
5. Test your program to make sure that you have a new value displayed on the LEDs. *What values/bit pattern should you see on the LEDs?*
6. Add code to your program that bit shifts the current value in the register 1 bit position back to the *right*. Think about which operator you would use to do a right bit shift operation.
7. Test your program to make sure that you have a new value displayed on the LEDs. *What values/bit pattern should you see on the LEDs?*
8. At this point, *should* have a program that displays the original value (hexadecimal *0x55*) on the LEDs. Even though different values are displayed on the LEDs throughout the code, they are displayed too quickly for the human eye to perceive.

   Remember, in the previous Lab 4, we used nest loops to create a 0.5 second *delay*. Add delay code to your program to have a 0.5 second delay after each time a value displayed on the LEDs. Be *careful* that you are use *different* registers for storing you loop counters and the value that you are writing out to the LEDs.
9. Test your program to make sure that it works as you expect. It should:
   (a) display the value 0x55 on the LEDs
   (b) delay for 0.5 seconds
   (c) shift bits to the left 1 position and display the new value on the LEDs
   (d) delay for 0.5 seconds
   (e) shift bits to the right 1 position and display the original value to the LEDs
10. Finally, it would be nice to not have to reset your Arduino every time you want to see this bit shifting. Add a loop to your program so that you bit shifting code repeats indefinitely.

Congratulations! You have successfully used bit-shifting operators to make the LEDs blink back and forth.

8 Lab: Bitwise Operations

Now, you will attempt to use the bitwise operators `AND` and `OR` on binary numbers stored in registers. You will use the operators along with the bit shifting operators to swap the upper and lower *nybbles* in a register. A *nybble* is another word for 4 bits, because a nybble is *half* a byte.

1. Start with a copy of your program from Section 6 and call the new file `swap_nybbles.s`. This should be the program that simply displays a byte on the LEDs.
2. Load the hexadecimal value `0x81` into the register and display it on the LEDs. *Does it produce the value that you expect?*
3. Using the `AND` operator, *bit mask* the lower 4 bits of the value in the register and store it in a new register. Try to preserve the original value in the original register. You can overwrite the bit mask register to do this. Alternatively you can use the `MOV` operator to copy the value of one register into another (Take look in the *instruction set* documentation on how to use `MOV`).
4. Test your program by displaying the lower 4 bits on the LEDs. *What value do you expect to be displayed on the LEDs?*
5. Once you have a program that displays these 4 bits. Add code that shifts these 4 bits to the *left* so that they are in the upper 4 bits of the register. (This should be 4 `LSL` operations).
6. Test your program by displaying this new shifted value on the LEDs. *What value do you expect to be displayed on the LEDs?*
7. Add code to your program that bit masks the *upper* 4 bits from the original register (i.e. the one that held the hexadecimal value `0x81`) and shifts those bits to the lower 4 bits in a new register. This code should be *very* similar to the code in steps 3-6. Test your program incrementally by displaying the intermediate values of registers on the LEDs.
8. Now you should have two registers, each with the upper and lower nybbles (i.e. 4 bits) from the original register shifted by 4 bit positions. Write code to use the `OR` operator to combine these two nybbles into one byte. Display this byte out to the LEDs. *What value do you expect to be displayed on the LEDs?*
9. Your program should have swapped the upper and lower bits in the original register, and as a result you should see the hexadecimal value *0x18* displayed on the LEDs.
10. Put the code that you wrote for the *nybble* swapping in a loop so that the upper and lower nybbles of a register are swapped continuously. Add a 0.5 second delay after the final LED display that you added in step 8 above.
11. Your program should now be constantly swapping the upper and lower 4 bits. Try starting with a different hexadecimal value, *can you get a different LED blinking effect to happen*?

Congratulations! You have successfully used bitwise operator `AND` and `OR` along with bit shifting operators to swap the upper and lower nybbles of a byte and display them on the LED. You are now truly a bit manipulation master, and are ready to tackle the Lab 5 Assembly Program.

9 Lab Assembly Program: My Computer Architecture

For the Lab 5 Assembly program that you will turn in, you will use bit operators that you learned about in this lab to decode a memory address for a computer architecture into its separate cache address components.

On KEATs under Lab 5, you will find a link *What is my computer architecture?* which describes the computer architecture your program will decode memory addresses for. *Make sure that you are logged into KEATs with your account*, and click on this link and read your computer architecture very carefully.

Before you begin writing your program, calculate the number of bits that are necessary for the Tag, Block, and Word Offset fields for your computer architecture. **Include your answer and an explanation of your calculations in your program code as comments at the beginning of your program**. Failing to include a written answer and calculations will result in losing marks for you program.

Your program should include a *subroutine* labelled `decode`. (Be sure to label it "`decode`" in all lower case letters and without anything else added to the label). Your `decode` subroutine will find the Tag, Block, and Word Offset bits for a supplied memory address and display them on the LEDs as bytes using the hardware set-up that we used in this Lab.

Your `decode` sub-routine should assume that the memory address it will decode is broken up into separate bytes and stored in registers. The *lower byte* of the memory address will be stored in register **r20**. The *second lowest byte* of the memory address will be stored in **r21**, and so on using as many registers necessary to store the entire memory address.

For example, if your memory address was 15 bits in size, then the *lower byte* of the memory address is stored in **r20** and the upper 7 bits are stored in **r21**. With this set-up, the *binary* address `00100100011` (hexadecimal `0x0123`) would be represented with the hexadecimal byte `0x23` in register **r20** and the upper 7 bits (i.e. hexadecimal `0x01`) stored in register **r21**. In this case the unused bits should be filled with a binary `0`.

More precisely, have your `decode` subroutine display the fields for your memory address in the following order:

1. First, display the **Tag bits** for the memory address on the LEDs. Fill any unused bits with binary 0's. For example, if you computer architecture has 3 Tag bits, then display the Tag bits as bits 0 to 2 on the LED, and fill in bits 3 to 7 with a binary 0.
2. If more than 1 byte is required to display all of the Tag bits, then display the bytes in *little endian* order.
3. Have your program hold each Tag byte for **0.5 seconds** on the LEDs.
4. Have your program turn off all of the LEDs and delay **1 second** before displaying the next field.
5. Next, display the **Block bits** for the memory address on the LEDs. Just like the Tag bits, fill an unused bits with binary 0's.
6. If more than 1 byte is required to display all of the Block bits, then display the bytes in *little endian* order.
7. Have your program hold each Block byte for **0.5 seconds** on the LEDs.
8. Have your program turn off all of the LEDs and delay **1 second** before displaying the next field.
9. Next, display the **Word offset bits** for the memory address on the LEDs. Just like the Tag bits, fill an unused bits with binary 0's.
10. If more than 1 byte is required to display all of the Word offset bits, then display the bytes in *little endian* order.
11. Have your program hold each Word offset byte for **0.5 seconds** on the LEDs.
12. Finally, have your subroutine turn off all of the LEDs for 1 second before returning from the sub-routine.

Here is an *example* to demonstrate the order of display. For this example, we will assume that there are 3 Tag bits with the Tag field value of binary `010` for this memory address, 9 Block bits with a Block field value of binary `100100011`, and 5 Word offset bits with a Word offset value of binary `01001`. Your `decode` would display the following values on the LEDs:

1. Display the byte `0x02` on the LEDs for 0.5 seconds for the Tag Field.
2. Turn off the LEDs for 1 second.
3. Display the byte `0x23` on the LEDs for 0.5 seconds for the first byte of the Block field.
4. Display the byte `0x01` on the LEDs for 0.5 seconds for the second byte of the Block field.
5. Turn off the LEDs for 1 second.
6. Display the byte `0x09` on the LEDs for 0.5 seconds for Word offset field.
7. Turn off the LEDs 1 second and then return from the sub-routine.

On KEATs under Lab 5, under the link *What is my computer architecture?* you will find a memory address for your computer architecture to demonstrate it's functionality with. Make sure that your program:

1. Loads the demonstration memory address into the correct registers (i.e. registers **r20**, **r21** and so on).
2. Calls the `decode` subroutine to display the Tag, Block, and Word components.
3. Does **not** loop over the `decode` subroutine. Simply have your program halt or loop forever at the end of the code like the assembly program in Lab1 does.
4. Your program is called `my_cache.s`
5. Your program has the *header* in the *program comments* as described on the KEATs, which includes *your name*, *your k-number*, *a short description of what your program does*.

10 What to turn in

When you have tested your program completely, **ONLY** turn in the `my_cache.s` assembly program on KEATs through the Lab 5 Assembly Program submission link.

DO NOT put your program in a .zip or .7z or a Word document. Doing so will result in losing marks for your Lab 5 Assembly Program submission.