LSU EE 4720 Homework 1 Solution Due: 29 January 2021

The solution has been copied into the assignment directory in file `hw01-sol.s`. An HTML version is at `https://www.ece.lsu.edu/ee4720/2021/hw01-sol.s.html`.

Problem 0: Follow the instructions for class account setup and for homework workflow in `https://www.ece.lsu.edu/ee4720/proc.html`. Review the comments in `hw01.s` and look for the area labeled "Problem 1."

Those who want to start before getting to the lab can find the assembler for the entire assignment at `https://www.ece.lsu.edu/ee4720/2021/hw01.s.html`. For MIPS references see the course references page, `https://www.ece.lsu.edu/ee4720/reference.html`. Easy MIPS practice problems can be found in the practice directory, see MIPS Homework and Practice Workflow in `https://www.ece.lsu.edu/ee4720/proc.html`.

Problem 1: The `hw01.s` file has a routine called `getbit`.

(*a*) Complete the `getbit` routine so that it returns the value of a bit from a bit vector that spans one or more bytes. Register `$a0` holds the start address of the bit vector and register `$a1` holds the bit number to retrieve. The most-significant bit of the first byte is bit number 0. When `getbit` returns register `$v0` should be set to `0` or `1`.

For example, a 16-bit bit vector is specified in the assembler below starting at the label `bit_vector_start`:

```
bit_vector_start:
        .byte 0xc5, 0x1f
```

In binary this would be $1100\,0101\,0001\,1111_2$. If `getbit` were called with `$a1=0` then bit number zero, meaning the leftmost bit in $1100\,0101\,0001\,1111_2$, should be returned and so `$v0=1`. For `$a1=2` a `0` should be returned.

Each memory location holds eight bits of the bit vector. For `$a1` values from 0 to 7 the bit will be in the byte at address `$a0`. For `$a1` values from 8 to 15 the bit will be in the byte at address `$a0+1`, and so on.

When the the code in `hw01.s` is run (by pressing F9 for example) a testbench routine will call `getbit` several times. For each call the testbench will print the value returned by `getbit` (meaning the value of `$v0`), whether that value is correct, and if wrong, the correct value. At the end it will print the number of incorrect values returned by `getbit`, which hopefully will be zero when you're done.

See the checkboxes in the code for more information on what is expected.

The solution appears below:

```
getbit:
        srl $t0, $a1, 3     # Compute byte offset from $a0.
        add $t0, $a0, $t0   # Compute address of byte holding bit.
        lbu $t1, 0($t0)     # Load that byte.
        andi $t2, $a1, 0x7  # Compute bit number within loaded byte.
        addi $t2, $t2, 24   # Find left shift amt that puts needed bit in MSB.
        sllv $t1, $t1, $t2  # Shift so that needed bit is most-significant.
        jr $ra
        srl $v0, $t1, 31    # Shift so that needed bit is least significant.
```

The first two instructions, `srl` and `add` compute the address of the byte holding the needed bit. The byte is loaded and shifted left by enough so that the needed bit is in the most-significant position of 32-bit register `t1`. The register is then shifted right by 31 bits so that the needed bit is in the least-significant position and all other bits are zero.

(*b*) The bit vector used by the testbench is specified with:

```
bit_vector_start:  # Note: MIPS is big-endian.
        .byte 0xc5, 0x1f
        .half 0x05af
        .word 0xedcba987
        .ascii "1234"
bit_vector_end:
```

The assembler will convert the lines following data directives `.byte`, `.half`, `.word`, and `.ascii` into binary and place them in memory. The total size will be $2 \times 1 + 2 + 4 + 3 = 11$ bytes. For the purposes of this problem those 11 bytes form a $11 \times 8 = 88$-bit bit vector. In most circumstances for something like the bit vector above one would use the same kind of data directive for all data, say using only `.byte`, but mixing directives is not wrong and in some cases may be convenient for example when the bit vector is constructed by concatenating pieces of different sizes and types. Note that the kind of data directives used above does not affect how `getbit` is written.

Following the bit vector are the tests for the testbench. For each test there is one line consisting of a bit number and the expected return value. For example, the second test sets `$a1=4` and expects a return value of `$v0=0`.

```
testdata:
        .half 0, 1
        .half 4, 0
        .half 10, 0
```

Add a test to the `testdata` data to test the part of the bit vector specified using `.ascii "123"`. The test should be written for `.ascii "123"` and should report an error if the directive were changed to `.ascii "213"`.

The `1234` in the data area above follows two bytes, one half, and one word. The total size of these is $2 \times 1 + 2 + 4 = 8$ bytes. So the `1` in `1234` starts at bit $8 \times 8 = 64$, which is the most-significant bit of the `1`. The ASCII for `1` is 31_{16} and `2` is 32_{16}, these differ in the least-significant bit, so we need to check bit $64 + 7 = 71$. It will be 1 for `1`. So we add data items for checking bit 71 for a 1:

```
testdata:
        .half 71, 1   # Solution to 1b.

        .half 0, 1
        .half 4, 0
        .half 10, 0
```