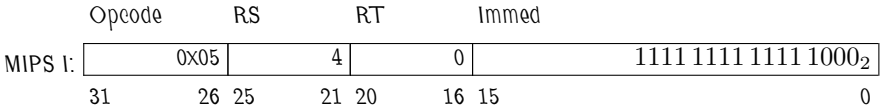
LSU EE 4720 **Homework 2** Solution **Due: 8 March 2023**

Problem 1: The code fragment below was taken from the course hex string assembly example. (The hex string example was not covered this semester. The full example can be found at `https://www.ece.lsu.edu/ee4720/2022/hex-string.s.html`.) The fragment below converts the value in register `a0` to an ASCII string, the string is the value in hexadecimal (though initially backward).

```
LOOP:
        andi $t0, $a0, 0xf    # Retrieve the least-significant hex digit.
        srl $a0, $a0, 4       # Shift over by one hex digit.
        slti $t1, $t0, 10     # Check whether the digit is in range 0-9
        bne $t1, $0, SKIP     # Don't forget that delay slot insn always exec.
        addi $t2, $t0, 48     # If 0-9, add 48 to make ASCII '0' - '9'.
        addi $t2, $t0, 87     # If 10-15, add 87 to make ASCII 'a' - 'z'.
SKIP:
        sb $t2, 0($a1)        # Store the digit.
        bne $a0, $0, LOOP     # Continue if value not yet zero.
        addi $a1, $a1, 1      # Move string pointer one character to the left.
```

(*a*) Show the encoding of the MIPS `bne a0, 0, LOOP` instruction. *Note: This is not the same as the instruction used in last year's Homework 2.* Include all parts, including—especially—the immediate. For a quick review of MIPS, including the register numbers corresponding to the register names, visit `https://www.ece.lsu.edu/ee4720/2023/lmips.s.html`.

The encoding appears below. Note that register `a0` (procedure call argument 0) is a helpful name for `r4`, and so a 4 is placed in the `rs` field. If the `bne` is taken it jumps backward by eight instructions (starting from the delay-slot instruction), and so the immediate field holds a -8 (which is $1111\,1111\,1111\,1000_2$ in a two's complement, 16-bit representation).



(*b*) RISC-V RV32I has a `bne` instruction too, though it is not exactly the same. Show the encoding of the RV32I version of the `bne a0, 0, LOOP` instruction. For this subproblem assume that the `bne` will jump backward eight instructions, just as it does in the code sample above.

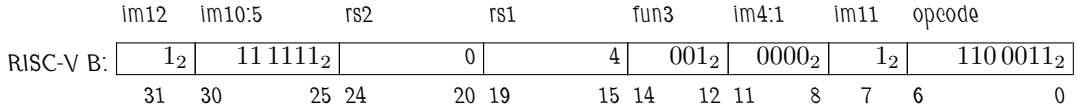
To familiarize yourself with RISC-V start by reading Chapter 1 of Volume I of the RISC-V specification, especially the Chapter 1 Introduction and Sections 1.1 and 1.3. Skip Section 1.2 unless you are comfortable with operating system and virtualization concepts. Other parts of Chapter 1 are interesting but less relevant for this problem. Also look at Section 2.5 (Control Transfer Instructions). The spec can be found in the class references page at `https://www.ece.lsu.edu/ee4720/reference.html`.

The branch instructions are discussed in Section 2.5 under the *Conditional Branches* heading. There are two significant differences with the MIPS `bne`. First, there is no delay slot. That's not relevant in this problem. Second, the immediate field value is used differently. Let `IMM` denote the immediate field value (based on the bits set in the instruction). The branch target is then `PC + 2 * IMM`, where `PC` is the address of the branch. (In MIPS the target would be `PC + 4 + 4 * IMM`.) So, we need to set the immediate to the number of bytes to skip divided by two. The problem says to jump back eight instructions, in RISC-V (and most RISC ISAs) that's 32 bytes, and so the immediate field should be set to -16 which is $1111\,1111\,0000_2$ in a two's complement, 12-bit representation.

Though Section 2.5 shows the encoding of a `bne` instruction, it does not provide the values for opcode fields and their extensions, instead using names: *BRANCH* for the `opcode` field and *BNE* for the `funct3` field. The values can be found in Chapter 24.

The encoding appears below. The instruction field names have been abbreviated, such as `im12` for `imm[12]`. Also note that in RISC-V immediate field names use the bit numbers within the immediate. So, for example, `imm[4:1]` (abbreviated `im4:1` below) indicates that the four bits in the instruction field (0000_2 in the example) are placed in bit positions 4:1 of the immediate. There is no field named `imm[0]` because the corresponding immediate bit is always set to zero. So, the `IMM*2` is computed by putting the twelve immediate bits in the instruction in bits 12:1 of the immediate, setting bit 0, the LSB, to zero.

In MIPS format I-instructions the 16-bit immediate is put in bits 15:0 of the instruction, which is straightforward and easy to understand. In RISC-V B-format instructions the 12-bit immediate is scrambled into four fields of the instruction. Following the convention of the B-format instruction the immediate bit positions will be numbered 12 to 1. (There is a bit position zero but it is always zero and so it does not appear in the instruction.) Bit 12 of the immediate is found in bit position 31 of the instruction. Bits 10:5 [sic] of the immediate are in bits 30:25 of the instruction. Where is bit 11 of the immediate? It's at bit 7, hanging out next to bits 4:1 of the immediate which are at bits 11:8 of the instruction. This bit scrambling is done to simplify hardware, as is explained in section 2.3 of the RISC-V standard. A question about the rationale for this bit scrambling may be asked on the 2023 midterm exam.



(*c*) Consider the four-instruction sequence from the code above:

```
        slti $t1, $t0, 10     # Check whether the digit is in range 0-9
        bne $t1, $0, SKIP     # Don't forget that delay slot insn always exec.
        addi $t2, $t0, 48     # If 0-9, add 48 to make ASCII '0' - '9'.
        addi $t2, $t0, 87     # If 10-15, add 87 to make ASCII 'a' - 'z'.
SKIP:
```

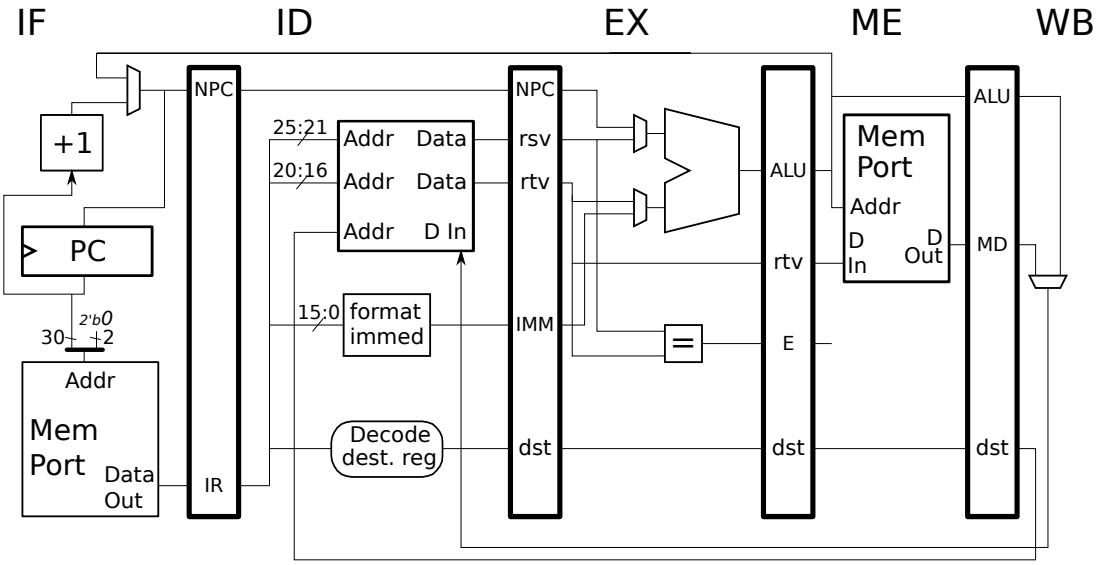
Re-write this sequence in RISC-V RV32I, and take advantage of RISC-V branch behavior to reduce this to three instructions (plus possibly one more instruction before the loop). For this problem one needs to focus on RISC-V branch behavior. Assume that the RISC-V `slti` and `addi` instructions are identical to their MIPS counterparts at the assembly language level. It is okay to retain the MIPS register names. *Hint: One change needs to be made for correctness, another for efficiency.*

Solution appears below. Before the loop is entered the `addi` instruction sets `t6` to 10, a constant that will come in handy. Inside the loop the four MIPS instructions are replaced by three RISC-V instructions. First, the `slti` is no longer necessary because RISC-V has a `blt` (branch less than). The `blt` itself checks whether `t0` is less than 10 (which is in `t6`). Using the `blt` to do the comparison is the efficiency change mentioned in the hint. Because RISC-V lacks delay slots the `addi t2,t0,48` had to be moved before the `blt`. That's the correctness change mentioned in the hint. Notice that RISC-V has register names that are similar to MIPS, such as `t0-t6` for caller-save (temporary) registers.

```
        # Instruction inserted before the loop to put 10 into register t6.
        addi t6, zero, 10
LOOP:

        # These instructions replace the four MIPS instructions.
        addi t2, t0, 48         # Convert assuming t0 in range 0-9 ..
        blt t0, t6, SKIP        # .. if that's correct, branch ..
        addi t2, t0, 87         # .. otherwise convert assuming a-f.
SKIP:
```

Problem 2: *Note: The following problem was assigned in each of the last six years, and its solution is available. DO NOT look at the solution unless you are lost and can't get help elsewhere. Even in that case just glimpse.* Appearing below are **incorrect** executions on the illustrated implementation. For each one explain why it is wrong and show the correct execution.



(*a*) Explain error and show correct execution.

```
# Cycle          0  1  2  3  4  5  6  7
lw r2, 0(r4)     IF ID EX ME WB
add r1, r2, r7      IF ID EX ME WB
```

The `add` depends on the `lw` through `r2`, and for the illustrated implementation the `add` has to stall in `ID` until the `lw` reaches `WB`.

```
# Cycle          0  1  2  3  4  5  6  7     SOLUTION
lw r2, 0(r4)     IF ID EX ME WB
add r1, r2, r7      IF ID ----> EX ME WB
```

(*b*) Explain error and show correct execution.

```
# Cycle          0  1  2  3  4  5  6  7
add r1, r2, r3   IF ID EX ME WB
lw r1, 0(r4)        IF ID -> EX ME WB
```

There is no need for a stall because the `lw` writes `r1`, it does not read `r1`.

```
# Cycle          0  1  2  3  4  5  6  7  SOLUTION
add r1, r2, r3   IF ID EX ME WB
lw r1, 0(r4)        IF ID EX ME WB
```

(*c*) Explain error and show correct execution.

```
# Cycle          0  1  2  3  4  5  6  7
add r1, r2, r3   IF ID EX ME WB
sw r1, 0(r4)        IF ID -> EX ME WB
```

A longer stall is needed here because the **sw** reads **r1** and it must wait until **add** reaches WB.

```
# Cycle          0  1  2  3  4  5  6  7  SOLUTION
add r1, r2, r3   IF ID EX ME WB
sw r1, 0(r4)        IF ID ----> EX ME WB
```

(*d*) Explain error and show correct execution.

```
# Cycle          0  1  2  3  4  5  6  7
add r1, r2, r3   IF ID EX ME WB
xor r4, r1, r5      IF ----> ID EX ME WB
```

The stall above allows the **xor**, when it is in ID, to get the value of **r1** written by the **add**; that part is correct. But, the stall starts in cycle 1 *before* the **xor** reaches ID, so how could the control logic know that the **xor** needed **r1**, or for that matter that it was an **xor**? The solution is to start the stall in cycle 2, when the **xor** is in ID.

```
# Cycle          0  1  2  3  4  5  6  7   SOLUTION
add r1, r2, r3   IF ID EX ME WB
xor r4, r1, r5      IF ID ----> EX ME WB
```