CS3432 Computer Organization
Fall 2025. Final Exam
05/13/2025 – 4:00PM to 6:45PM MDT (theoretical part)
05/13/2025 – 4:00PM to 11:59PM MDT (practical part)

All documents allowed
Electronic devices **without** Internet connectivity allowed[1]
You need to turn in the theoretical part of the exam (Sections 1, 2 and 3) by 05/13/2025 6:45PM MDT on paper, directly to your instructor. You need to turn in the practical part of the exam (Section 4) by 05/13/2025 11:59PM MDT by email to `utep-arc-spring-2025-final@christoph-lauter.org`.

1 Encoding Risc-V Instructions

The *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2* contains the following table on page 104 of the document:

imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

The table lists the encodings of various instructions. It gives the 32 bits of each machine instruction per operation. In the table, bit b_{31}, the MSB, is on the left, bit b_0, the LSB, is on the right. 5 bits are reserved in the instruction encodings for the register numbers, per register (`rs1`, `rs2` or `rd`).

Use the table to encode the assembly instruction

```
xor r4,r7,r3
```

Give 4 bytes (32 bits) for the encoded instruction. Write the bytes in hexadecimal notation. Remember that Risc-V is little-endian, so give the byte with the LSB first.

[1] Put your device into airplane mode.

2 Representing Numbers in Computers

1. Encode the value 47806 as a 32bit unsigned integer. Give 8 hexadecimal digits for your final answer.

2. Encode the value -47806 as a 32bit signed integer in two's complement. Give 8 hexadecimal digits for your final answer.

3. Encode the value -47806.65625 as an IEEE754 binary32 Floating-Point number. Give 8 hexadecimal digits for your final answer. Explain the different computational and encoding steps you go through.

4. Decode the IEEE754 binary64 Floating-Point number `0x4084230000000000`. Give a real number, written in decimal, for your final answer. Explain the different decoding and computational steps you go through.

3 Creating a 1 out of 2^k Decoder with NANDs

A 1 out of 2^k decoder is a hardware unit that has k input bits $s_{k-1}, s_{k-2}, \ldots, s_1, s_0$ that form a number $s = \sum_{i=0}^{k-1} s_i\, 2^i$ in binary. The decoder also has 2^k binary outputs $t_0, t_1, \ldots, t_{2^k-1}$. At any moment, only one of the outputs t_s is 1; all other outputs are 0. The number s of the output t_s that is activated (i.e. set to 1) is given by the input number s.

Let us consider the example of a 1 out of 4 decoder. In this case, $k = 2$ and we have input bits s_1 and s_0, as well as outputs t_0, t_1, t_2 and t_3. We get the following truth table

s_1	s_0	t_0	t_1	t_2	t_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

1. Assuming $k = 1$ to start, design a 1 out of 2 decoder with NAND gates. Your circuit has 1 input s_0 and two outputs t_0 and t_1.

2. Assuming $k = 2$ and using as many 1 out of 2 decoders and extra NAND gates, design a 1 out of 4 decoder. Your circuit has 2 inputs s_1 and s_0 and four outputs t_0, t_1, t_2 and t_3. It implements the truth table given above.

3. Describe a recursive mechanism to design a 1 out of 2^k decoder for any number of input bits k.

4 Coding in C and Risc-V Assembly (Practical Part)

Phone networks, landline and mobile, use digital transmission since the 1970ies. The voice of each party on the call is digitized by measuring the voltage on the phone's microphone 8000 times each second. That voltage results in a 16bit signed integer number. As transmission of 128kbit/s is too expensive, a non-IEEE754-compliant special Floating-Point representation is used to compress each 16bit signed integer to an 8bit byte. This reduces the data-rate required to 64kbit/s.

For historical reasons, two different of these special Floating-Point representations are used world-wide. The United States, Canada and Japan use an encoding called ulaw. All other countries use a different encoding called alaw. As long as a call stays within the US and Canada (or goes to Japan without crossing any other country), no conversion from ulaw to alaw and back is required. However, as soon as a call crosses into any other country, a conversion is required.

This means that, for example, when someone in El Paso, TX, USA, calls someone in Cuidad Juárez, Chihuahua, Mexico, a computer is needed to convert from ulaw to alaw and back, byte-per-byte, 8000 times per second, in real-time, for the whole duration of the call.

Imagine you work for a company that produces routers for international networking. Your boss charges you with programming the part that performs alaw-to-ulaw and ulaw-to-alaw conversion. The final result must run very efficiently, without any overhead, as the device is cheap and does not have a lot of compute power. Your boss wants you to eventually come up with two functions

```
unsigned char alaw_to_ulaw_ez(unsigned char a);
```

and

```
unsigned char ulaw_to_alaw_ez(unsigned char u);
```

that perform the job and they want these functions to be handcoded in efficient Risc-V assembly.

Neither you nor your boss have a precise understanding on how ulaw or alaw work. You are hence happy to find a conversion routines in a file `g711.c` on the Internet. The file contains two functions `alaw_to_ulaw` and `ulaw_to_alaw` that read like this:

```
unsigned char alaw_to_ulaw(unsigned char a) {
  return slin_to_ulaw(alaw_to_slin(a));
}

unsigned char ulaw_to_alaw(unsigned char u) {
  return slin_to_alaw(ulaw_to_slin(u));
}
```

Hence, you are confident that you can succesfully complete the job your boss gave you. It suffices to translate the functions to Risc-V assembly!

Unfortunately, you then discover that everything goes down a rabbit-hole very fast: `alaw_to_ulaw` calls `alaw_to_slin` and `slin_to_ulaw`, which in turn call other functions! Overall, there are over 250 lines of C to be translated to Risc-V assembly and some of the functions, like `slin_to_ulaw`, look very nasty.

However, your boss just asked you for assembly for `alaw_to_ulaw_ez` and `ulaw_to_alaw_ez`. If only there were a way to rewrite `alaw_to_ulaw` and `ulaw_to_alaw` as `_ez` variants, without calling other function and doing nasty bit fiddling...

You decide that this is the path to go:

1. You add a `main` function to the given code to see what comes out of `alaw_to_ulaw` and `ulaw_to_alaw` as a function of the different inputs that these functions take. How many different inputs are there? Could you precompute the answers and just store them? Or are there too many?

2. You rewrite `alaw_to_ulaw` and `ulaw_to_alaw` as `alaw_to_ulaw_ez` and `ulaw_to_alaw_ez` in C, based on what you observed. You make sure that your `_ez` variants do not call any other function and that they stay simple.

3. **Most importantly, you modify your `main` function to test your `_ez` variants of the functions against the existing functions `alaw_to_ulaw` and `ulaw_to_alaw`.** For any possibly input, your corresponding `_ez` function must output the same result as the original function.

4. Once you are convinced that your `alaw_to_ulaw_ez` and `ulaw_to_alaw_ez` functions, as you have coded them in C, work, you translate them to Risc-V assembly.

For all the work in this Section, you suppose that you are on a system where the C data type `unsigned char` yields an 8bit (1byte) unsigned number, `unsigned short` yields a 16bit unsigned number and that `signed short` yields a 16bit signed number in two's complement.

For this practical part, you need to turn in:

1. A C source code file `g711.c` where you added a `main` function that does a reasonable job, as well as a C code implementation of the functions `alaw_to_ulaw_ez` and `ulaw_to_alaw_ez`. Your C source code file must compile without warnings. You must not modify the existing functions in the file `byte_is_zero`, `byte_top`, `alaw_to_slin`, `slin_to_alaw`, `ulaw_to_slin`, `slin_to_ulaw`, `alaw_to_ulaw` and `ulaw_to_alaw`.

2. An assembly file `g711.s` where you implement the functions `alaw_to_ulaw_ez` and `ulaw_to_alaw_ez` in Risc-V assembly. Your assembly file is supposed to be translatable to an encoded binary object with a Risc-V assembler.

For this practical part, be sufficiently lazy but smart: you are not required to actually translate the C code directly to assembly. You are just required to provide the functions `alaw_to_ulaw_ez` and `ulaw_to_alaw_ez` in your assembly code.