CS3432 Computer Organization
Spring 2025. Midterm II
04/15/2025 – 4:30PM to 5:50PM MDT

Theoretical part: All documents allowed - No internet allowed[1]
Practical part: Documents allowed - Internet allowed - Your submission must be your own work

You need to turn in the theoretical part of the exam (Sections 1 and 2) by 04/15/2025 5:50PM MDT on paper, directly to your instructor. You need to turn in the practical part of the exam (Section 3) by 04/16/2025 1:59PM MDT by email to `utep-spring-2025-arc-midtermII@christoph-lauter.org`.

1 Decoding Risc-V Instructions

On Risc-V, the CPU loads a 32bit, i.e. 4byte, instruction word at each turn of the van Neumann cycle. The lower 7bits (from the least significant bit b_0 through the bit b_6) always indicate the basic instruction opcode. In the case when these 7bits read binary 1100011, the instruction is a conditional branch instruction (`BEQ, BNE, BLT, BLTU, BGE, BGEU`). In this case of a conditional branch, the rest of the 32bit instruction word above the 7 low-order bits, decode according to the Risc-V B-immediate type of instructions.

For the B-immediate type of instructions, the following split of the bits in the instruction word is performed:

- Bits b_0 through b_6 contain the basic instruction opcode (see above).
- Bit b_7 contains bit i_{10}, i.e. bit number 10 of the 12bit immediate value.
- Bits b_8 through b_{11} contain bits i_0 through i_3 of the 12bit immediate value.
- Bits b_{12} through b_{14} contain a bit pattern that indicates which one of the conditional branch instructions is executed. In the case of a `BEQ` (branch if equal) instruction, that bit pattern is 000.
- Bits b_{15} through b_{19} encode the number of the first register taken into account by the instruction.
- Bits b_{20} through b_{24} encode the number of the second register taken into account by the instruction.
- Bits b_{25} through b_{30} contain bits i_4 through i_9 of the 12bit immediate value.
- Bit b_{31} contains bit i_{11} of the 12bit immediate value.

A `BEQ` instruction is taken if the values in the first register and the second register are equal. If a branch gets taken, the immediate gets multiplied by 2, which yields a 13bit value, then sign-extended from 13 to 32bits to form a 32bit value. That 32bit value is added to the current address in the program counter `PC`, discarding a possible carry in bit position 33. If the branch is not taken, the next instruction that is executed is at `PC+4`.

[1]Airplane mode **without** Wifi!

Suppose the following values are contained in the registers:

`r0`	`0x00000000`
`r1`	`0xdeadbeef`
`r2`	`0x00000001`
`r3`	`0xcafebabe`
`r4`	`0x0000002a`
`r5`	`0xcafebabe`
`r6`	`0xaffeaffe`
`r7`	`0x0000002a`

Suppose the current `PC` value is `0x00ff0010` and suppose that at this address `0x00ff0010`, the instruction `0xfc438ee3` can be found in memory.

Decode the instruction, indicate at which address the `PC` will point after the execution of the instruction and explain the behavior.

2 A Parity Bit Generation Unit

When digital data is transmitted over wires or with wireless transmission techniques or when digital data is stored in memory and then read back, transmission or storage errors may flip a bit in the data. This happens with very low probability, but it still happens.

An easy, though not very good, test if such a 1-bit error has occurred can be performed using a so-called parity check. In each chunk of original data, say a byte or a 32bit word, the number of bit set to 1 is counted. If the number of bits set to 1 is even (*par*), an extra parity bit set to 1 is added to the chunk of data. If the number of bits set to 1 is odd (*impar*), the extra parity bit is set to 0. The chunk of data is then transmitted or stored along with its extra parity bit. Upon reception resp. data retrieval, the parity of bits set to 1 is determined again and compared with the parity bit. If a 1-bit error occurred, the parity bit is not found to be valid; the data can be flagged as *invalid*.

Modern processors, including very simple and cheap micro-controller CPUs, include hardware to compute the parity bit p for a word w of 2^n bits (with $n = 1, 2, 3, \ldots$). In this exercise, you will design a circuit for such parity bit computation hardware.

1. Design a NXOR gate with 2 bits of input, a and b, and 1 bit of output c, using nothing but 2-bit input, 1-bit output NAND gates. A NXOR gate is defined by the following truth table.

a	b	c
0	0	1
0	1	0
1	0	0
1	1	1

2. Design a parity bit generation circuit taking a $4 = 2^2$ bit input $w = w_3w_2w_1w_0$ and producing a parity bit p. The output p is 1 if and only if w contains an even number of bits set to 1. Use NXOR gates as your basic brick. Observe that the sum of two even numbers is even and that the sum of two odd number is even but that the sum of an odd and an even number (or of an even and an odd number) is odd.

3. Design a parity bit generation circuit taking a $8 = 2^3$ bit input $w = w_7 w_6 \ldots w_1 w_0$ and producing a parity bit p. The output p is 1 if and only if w contains an even number of bits set to 1. Recycle the circuit with a 4 bit input, possibly instantiating several of these circuits, and use extra NXOR gates.

4. Describe a recursive circuit design strategy for a circuit taking a 2^n bit input $w = w_{2^n-1} w_{2^n-2} \ldots w_1 w_0$ and producing a parity bit p, for any $n = 1, 2, 3 \ldots$. The output p is 1 if and only if w contains an even number of bits set to 1.

3 Conversion to Hexadecimal ASCII (Practical Part)

For this Section, you need to translate the functions marked for translation inside the C source code file `convert.c` that is given on the course website to Risc-V assembly. Start by modifying the C code so it can easily be translated to Risc-V afterwards. You need to turn in the modified C code, which still needs to compile and work correctly, as well as the Risc-V assembly code in `convert.s`.

Here is an example of a run of the program:

```
Please enter a string:
Hello World!
You entered the string:
Hello World!
The hexadecimal representation of the ASCII byte sequence is
48656C6C6F20576F726C6421
```