# Introduction to Computer Architecture: exam

R. Pacalet

2022-12-01

The text in black is the original one. The text in red is examples of the expected correct answers. Only this text was expected, possibly in shorter form, nothing more. The text in blue is extra comments about the expected correct answers. Warning: the course changes frequently (content, vocabulary, examples…); some questions and answer proposals can thus be partly or completely out of scope. Warning: some questions can be answered in many different ways; the proposed answers are just examples and they are not exhaustive.

You can use any document but communicating devices are strictly forbidden. Please number the different pages of your paper and indicate on each page your first and last names. You can write your answers in French or in English, as you wish. Precede your answers with the question's number. If some information or hypotheses are missing to answer a question, add them. If you consider a question as absurd and thus decide to not answer, explain why. If you do not have time to answer a question but know how to, briefly explain your ideas. Note: copying verbatim the slides of the lectures or any other provided material is not considered as a valid answer. Advice: quickly go through the document and answer the easy parts first.

## 1 CMOS logic (2.5 points)

The `baz` logic gate has two inputs `A` and `B`, one output `X` and the following CMOS schematic:

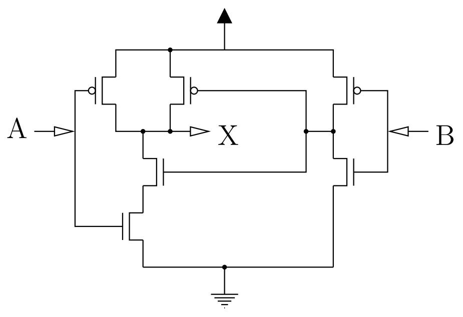

Figure 1: The `baz` logic gate

1. Write its truth table.

2. Write the boolean equation of the `X` output of `baz` using the `NOT`, `AND` and `OR` operators and parentheses. Do not assume any precedence between the boolean operators, use parentheses to make your equation non ambiguous.
3. Imagine a graphical symbol for `baz` and draw it.

1. The truth table is:

| A | B | X |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

2. $X = (\text{NOT } A) \text{ OR } B$.

3. Using the style seen in class we could represent the **`baz`** gate as shown on Figure 2.

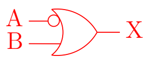

Figure 2: Symbol of the **`baz`** gate

## 2 RISC-V assembly (1 points)

1. Explain what an ABI is and what it is used for.

2. What difficulties could be encountered in a software project if no ABI was agreed on by the developers?

1. An ABI (for Application Binary Interface) is a set of conventions that compilation tool chains or assembly programmers use to guarantee the proper behavior of software programs. It specifies the use of registers and of the memory. For instance it defines what registers must be preserved across function calls, what register is used to store the return address when calling functions, what registers are used to pass arguments to functions and to return results, what registers are used as stack pointer, frame pointer...

2. Without an ABI it would not be possible for different software components to cooperate. Component A could store arguments in some registers before calling a function from component B while component B would expects the arguments to be passed in other registers. Or component A and B would use different stack pointers which would lead to memory corruption. Another example of difficulties is that of the saved registers: component A would expect a set of saved registers to be preserved across function calls but a function implemented by component B would modify them and preserve other registers.

## 3 CMOS logic (2.5 points)

The **qux** logic gate has 3 inputs **A**, **B** and **C**, one output **X** and the following truth table:

| A | B | C | X |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 |

1. Draw the CMOS schematic of **qux** using only N and P transistors.
2. Write the boolean equation of the **X** output of **qux** using the **NOT**, **AND** and **OR** operators and parentheses. Do not assume any precedence between the boolean operators, use parentheses to make your equation non ambiguous.
3. Imagine a graphical symbol for **qux** and draw it.

1. We can observe that the **qux** output is `1` if and only if the **C** input is `0` and at least one of **A** or **B** is also `0`. This immediately gives the network of P transistors between the power supply and the **qux** output: a P transistor which grid is **C** in series with a group of 2 P transistors in parallel which grids are **A** and **B** respectively. This way, if **C** is not `0` or if **A** and **B** are `1` there is no path between the power supply and the output, while in all other circumstances there is such a path and the output is `1`. As always with CMOS logic the network of N transistors between the ground and the **qux** output is dual of the network of P transistors: a N transistor which grid is **C** in parallel with a group of 2 N transistors in series which grids are **A** and **B** respectively. The CMOS schematic is represented on Figure 3.

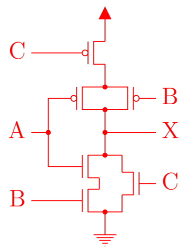

Figure 3: CMOS schematic of the **qux** gate

2. `X = (NOT C) AND ((NOT A) OR (NOT B)) = NOT (C OR (A AND B))`

3. Using the style seen in class we could represent the **qux** gate as shown on Figure 4.

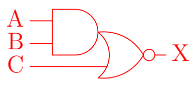

Figure 4: Symbol of the **qux** gate

## 4 Arithmetic and Logical Unit (ALU) (5 points)

A 32-bits hardware microprocessor implements the RV32I Instruction Set Architecture (RISC-V 32 bits integer ISA, without extensions). We want to design an ALU for this microprocessor, starting with a small subset of the various operations. The ALU operates on two 32-bits input operands, $A = a_{31} \cdots a_0$ and $B = b_{31} \cdots b_0$, that can represent signed numbers in two's complement, unsigned numbers or anything else that fits on 32 bits (ASCII characters...). A control input $f$ selects the ALU operation. The ALU outputs a 32-bits result $S = s_{31} \cdots s_0$ and three one bit flags, $vs$, $vu$ and $sgn$. The $vs$, $vu$ and $sgn$ flags are always equal to 0 except when the ALU operation is an addition and:

- if, considering the operands as signed numbers, there is an overflow $vs$ takes value 1.
- if, considering the operands as unsigned numbers, there is an overflow $vu$ takes value 1.
- if, considering the operands as signed numbers, the result is strictly negative $sgn$ takes value 1; **important**: even if there is an overflow, $sgn$ must be exact.

The functional specification of the ALU is summarized in Table 3.

Table 3: ALU functional specification

| $f$ | ALU operation |
|---|---|
| `0` | addition ($S \leftarrow A + B$) |
| `1` | bitwise **`XOR`** ($S \leftarrow A$ XOR $B$) |

At the heart of the ALU there will 32 identical **`ael`** elements connected together as shown on Figure 5. The $i^{th}$ **`ael`** receives one bit of the first ALU operand ($a_i$), one bit of the second ALU operand ($b_i$), one carry input ($c_i$) from the previous **`ael`** (or 0 for the first **`ael`**), plus the control input $f$ that selects the ALU operation. It outputs a one bit result ($s_i$) and a carry output ($c_{i+1}$). Inside **`ael`** the inputs and outputs are named $f$, $a$, $b$, $x$, $s$ and $y$, as shown on the leftmost **`ael`** instance on Figure 5.

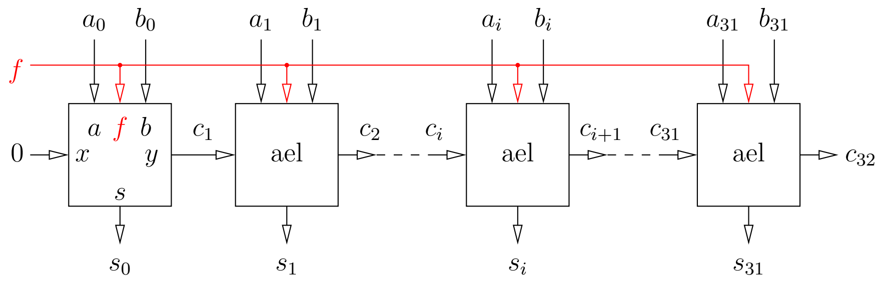

Figure 5: 32-bits ALU

1. Write the truth tables of the $s$ and $y$ outputs of **`ael`** depending on its $a$, $b$, $x$ and $f$ inputs.

2. Design the schematic of **`ael`** using only the logic gates and symbols of Figure 6. Try to optimize your design such that it uses as few hardware as possible.
3. Using the same logic gates and symbols design a circuit to compute the **`vs`** flag. The inputs can be any signal of Figure 5.
4. Do the same for the **`vu`** flag.
5. Do the same for the **`sgn`** flag.

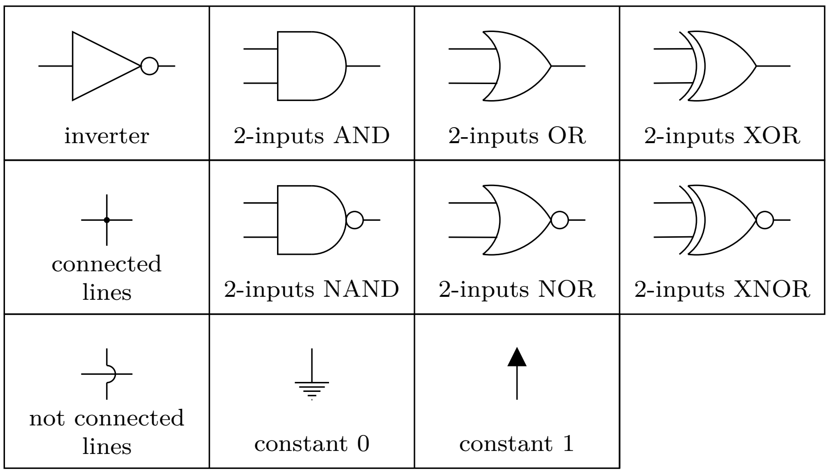

Figure 6: Logic gates and symbols

1. The $y$ output of **`ael`** is meaningless when $f = 1$. We represent this in the truth table with a - to indicate that any value would be fine.

| $f$ | $x$ | $a$ | $b$ | $y$ | $s$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | - | 0 |
| 1 | 0 | 0 | 1 | - | 1 |
| 1 | 0 | 1 | 0 | - | 1 |
| 1 | 0 | 1 | 1 | - | 0 |
| 1 | 1 | 0 | 0 | - | 0 |
| 1 | 1 | 0 | 1 | - | 1 |
| 1 | 1 | 1 | 0 | - | 1 |
| 1 | 1 | 1 | 1 | - | 0 |

2. The schematic of **`ael`** is shown on Figure 7. It is slightly optimized by reusing the same XOR gate for the addition and bitwise XOR operations.

The $y$ output is always that of the addition operation, even for the bitwise XOR operation, because in this case we don't care $y$ while anything else would cost extra hardware. It is computed as seen in class for the carry output of a full adder: $y = ((a \text{ XOR } b) \text{ AND } x) \text{ OR } (a \text{ AND } b)$.

The $s$ output is computed as $s = (a \text{ XOR } b) \text{ XOR } (x \text{ AND } (\text{NOT } f))$. When $f = 0$ (addition), this simplifies as $s = a \text{ XOR } b \text{ XOR } x$, which is indeed the equation of the sum bit of a full adder. When $f = 1$ (bitwise XOR), this simplifies as $s = a \text{ XOR } b \text{ XOR } 0 = a \text{ XOR } b$.

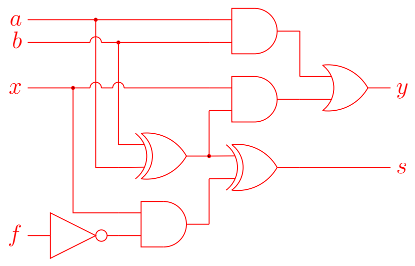

Figure 7: Schematic of the **ael** element

3. As seen in class the equation of the **vs** flag is $vs = c_{31} \text{ XOR } c_{32}$. The schematic is thus as shown on Figure 8.

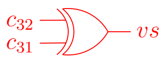

Figure 8: Schematic of the **vs** flag

4. The unsigned overflow flag **vu** is the output carry $c_{32}$ itself. The schematic is thus as shown on Figure 9 where the diamond symbol represents the renaming.

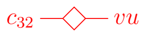

Figure 9: Schematic of the **vu** flag

5. The sign flag **sgn** is the same as the most significant bit of the 33 bits output we would have if we were sign-extending the two inputs from 32 to 33 bits ($a_{32} = a_{31}$ and $b_{32} = b_{31}$). This is thus $sgn = a_{32} \text{ XOR } b_{32} \text{ XOR } c_{32} = a_{31} \text{ XOR } b_{31} \text{ XOR } c_{32}$. The schematic is thus as shown on Figure 10.

## 5 RISC-V assembly (2 points)

In the following we use the RV32I Instruction Set Architecture (ISA) with the Integer, Long and Pointer 32 bits (ILP32) Application Binary Interface (ABI). Only basic instructions are allowed, pseudo-instructions are **forbidden**. We assume that each executed instruction takes exactly one clock cycle.

1. Write in RV32I assembly language a function named `foo` that takes two 32-bits signed numbers as arguments denoted $a$ and $b$, computes their sum

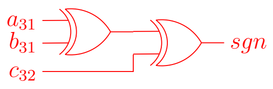

Figure 10: Schematic of the **sgn** flag

$s = a + b$, their difference $d = a - b$ and returns the bitwise exclusive or $s$ XOR $d$ between the sum and the difference.

2. Assuming the input arguments are random and independent what is the average number of clock cycles taken by your `foo` function?

3. Could you optimize it for speed? If yes propose a faster version and calculate the new average number of clock cycles per `foo` execution.

1. The ABI specifies that the two arguments are passed in registers `a0` (argument $a$) and `a1` (argument $b$), that the result shall be returned in register `a0` and that the return address is in register `ra`. It also specifies that we can freely use the `t0` to `t6` registers to store intermediate values because they are not *saved* registers. The `foo` function could be written as follows where we do not use the stack, store the sum in register `t0` and the difference in register `t1`:

```
foo:
  add   t0,a0,a1      # t0 <- a0+a1  (s <- a+b)
  sub   t1,a0,a1      # t1 <- a0-a1  (d <- a-b)
  xor   a0,t0,t1      # a0 <- t0 XOR t1  (a0 <- s XOR d)
  jalr  zero,0(ra)    # return at ra, discard PC+4
```

2. The number of clock cycles taken by the `foo` function is exactly 4.

3. There is no obvious way to optimize `foo`.

## 6 Binary representation of numbers (5 points)

Unless otherwise stated all given integer values are in base 10. Let $A$ be an integer and $a_8a_7a_6\cdots a_1a_0$ its 2's complement representation on 9-bits.

1. If $A \geq 0$ what is the 10 bits sign and magnitude representation of $A$?
2. If $A < 0$ what is the 10 bits sign and magnitude representation of $A$?
3. What is the minimum number of bits to represent +67 in 2's complement?
4. What is the minimum number of bits to represent -265 in 2's complement?
5. What is the minimum number of bits to represent -1024 in 2's complement?
6. What is the minimum number of bits to represent +128 in 2's complement?
7. Write the 8 bits binary sign and magnitude representation of +67.
8. Write the 8 bits binary sign and magnitude representation of -102.
9. Write the 8 bits binary sign and magnitude representation of -96.
10. Write the 8 bits binary sign and magnitude representation of +125.

1. $A = a_9a_8a_7a_6\cdots a_1a_0$ with $a_9 = 0$
2. $A = b_9b_8b_7b_6\cdots b_1b_0$ with $b_9 = 1$ and $b_8b_7b_6\cdots b_1b_0 = \overline{a_8a_7a_6}\cdots\overline{a_1a_0} + 1$ (we denote $\overline{x}$ the inverse of bit $x$)
3. 8 ($-2^7 = -128 \leq 67 \leq 127 = 2^7 - 1$ but $2^6 - 1 = 63 < 67$)

4. 10 ($-2^9 = -512 \leq -265 \leq 511 = 2^9 - 1$ but $-265 < -256 = -2^8$)
5. 11 ($-1024 = -2^{10}$)
6. 9 ($-2^8 = -256 \leq 128 \leq 255 = 2^8 - 1$ but $2^7 - 1 = 127 < 128$)
7. $67_{10} = 01000011_2$ ($67 = 64 + 2 + 1 = 2^6 + 2^1 + 2^0$)
8. $-102_{10} = 11100110_2$ ($102 = 64 + 32 + 4 + 2 = 2^6 + 2^5 + 2^2 + 2^1$)
9. $-96_{10} = 11100000_2$ ($96 = 64 + 32 = 2^6 + 2^5$)
10. $125_{10} = 01111101_2$ ($125 = 64+32+16+8+4+1 = 2^6+2^5+2^4+2^3+2^2+2^0$)

## 7 RISC-V assembly (2 points)

In the following we use the RV32I Instruction Set Architecture (ISA) with the Integer, Long and Pointer 32 bits (ILP32) Application Binary Interface (ABI). Only basic instructions are allowed, pseudo-instructions are **forbidden**. We assume that each executed instruction takes exactly one clock cycle.

A programmer wrote the following assembly code for a function `bar`:

```
bar:
  addi  sp,sp,-32
  sw    s0,28(sp)
  addi  s0,sp,32
  sw    ra,-4(s0)
  lw    t0,0(a0)
  lw    t1,0(a1)
  andi  t2,a2,1
  beq   t2,zero,label
  addi  t2,zero,t0
  addi  t0,zero,t1
  addi  t1,zero,t2
label:
  sw    t0,0(a0)
  sw    t1,0(a1)
  lw    ra,-4(s0)
  lw    s0,28(sp)
  addi  sp,sp,32
  jalr  zero,0(ra)
```

1. Explain what the input arguments and output results are and what the `bar` function does.
2. Assuming the input arguments are random and independent what is the average number of clock cycles taken by the `bar` function?
3. Do you think this code is correct? If not explain what is wrong with it and write a new code with the errors fixed.
4. Could the code be optimized for speed? If yes propose a faster version and calculate the new average number of clock cycles per `bar` execution.

1. `bar` takes 3 input parameters in registers `a0`, `a1` and `a2`. `a0` and `a1` are the addresses in memory of two 32-bits words. `a2` is an integer value. `bar` loads the two 32-bits words. It then swaps them if `a2` is odd. Finally it stores back the two 32-bits words in memory. There is no output result (or, equivalently, the output results are the unmodified input parameters).
2. If `a2` is even `bar` takes 14 clock cycles. If `a2` is odd `bar` takes 17 clock cycles. In average `bar` takes $(14 + 17)/2 = 15.5$ clock cycles.

3. The code is not correct because registers `s0` and `ra` are saved at the same position in the stack frame (if `s0 = sp + 32`, `28(sp) = -4(s0)`). Storing and restoring `ra` at address `-8(s0)` instead of `-4(s0)` suffices to fix the bug:

```
bar:
  addi sp,sp,-32
  sw   s0,28(sp)
  addi s0,sp,32
  sw   ra,-8(s0)
  lw   t0,0(a0)
  lw   t1,0(a1)
  andi t2,a2,1
  beq  t2,zero,label
  addi t2,zero,t0
  addi t0,zero,t1
  addi t1,zero,t2
label:
  sw   t0,0(a0)
  sw   t1,0(a1)
  lw   ra,-8(s0)
  lw   s0,28(sp)
  addi sp,sp,32
  jalr zero,0(ra)
```

4. The code could be optimized by not using the stack, by testing `a2` first, and with a smarter swap of the two 32-bits words:

```
bar:
  andi t0,a2,1
  beq  t0,zero,label
  lw   t0,0(a0)
  lw   t1,0(a1)
  sw   t0,0(a1)
  sw   t1,0(a0)
label:
  jalr zero,0(ra)
```

The new average number of clock cycles is $(3 + 7)/2 = 5$.