Introduction to Computer Architecture exam (1 hour)

Renaud Pacalet - 2026-01-26

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.

The exam is closed book, closed notes, except one A4-sized, double sided, cheat sheet. Calculators are allowed 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. Advice: quickly go through the document and answer the easy parts first.

The first and second questions are worth 6 points each. The third question is worth 8 points.

1. CMOS logic

The `qux` logic gate has three inputs `A`, `B`, `C`, and one output `X`. Its CMOS schematic is shown on Figure 1.

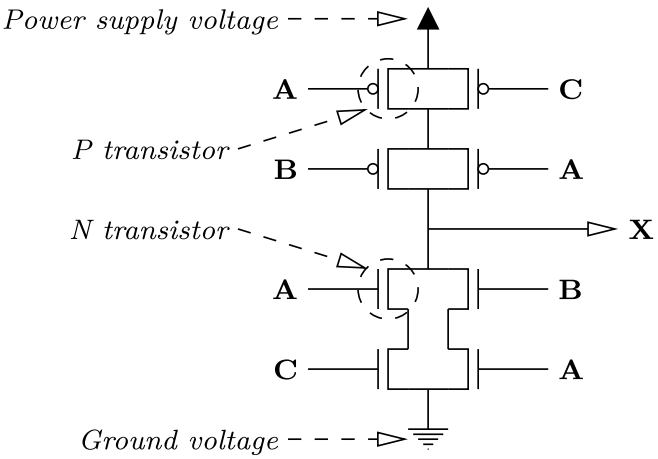


Figure 1: The `qux` logic gate

1. Write its truth table.

The truth table of `qux` is shown in Table 1.

A	B	C	X
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

Table 1: Truth table

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: `NOT A AND B` is ambiguous; `(NOT A) AND B` is not.

From the truth table we can deduce the boolean equation: $X = \text{NOT}((A \text{ AND } (B \text{ OR } C))$

3. Can you find another CMOS schematic for `qux` with less transistors? If not explain why, else draw the optimized CMOS schematic.

An optimized schematic for `qux` with 6 transistors instead of 8 is shown on Figure 2.

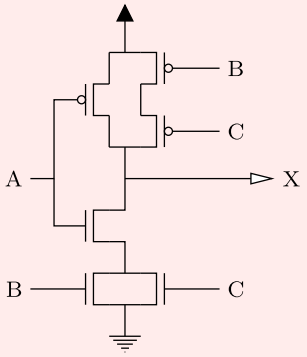


Figure 2: The optimized `qux` logic gate

4. Imagine a graphical symbol for `qux` and draw it.

A possible symbol for `qux` could be as shown on Figure 3.

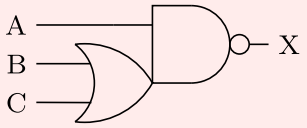


Figure 3: Symbol for the `qux` logic gate

2. Binary representation of numbers

Notations:

- Unless otherwise stated, numeric values are in base 10.
- $X \bmod Y$, where Y is not null, is the non-negative remainder of the integer division of X by Y : $10 \bmod 4 = 2, -11 \bmod 4 = 1$.
- $\lfloor V \rfloor$ is the largest integral value not greater than V : $\lfloor \frac{7}{3} \rfloor = 2$.

Let $a_8 a_7 a_6 ... a_1 a_0$ be the 9-bits two's complement representation of signed integer A.

1. What is the range of integer values that A can take?

   $A \in \{-256, ..., 255\}$

2. What is the 9-bits two's complement representation of $A \bmod 128$?

   $00 a_6 a_5 ... a_1 a_0$

3. Let $a_8 a_7 a_6 ... a_1 a_0 111$ be the 12-bits two's complement representation of integer B. What is the mathematical relation between integers A and B?

   $B = 8A + 7$

4. What is the 9-bits two's complement representation of $\lfloor \frac{A}{16} \rfloor$?

   $a_8 a_8 a_8 a_8 a_8 a_7 a_6 a_5 a_4$

5. What is the 9-bits two's complement representation of 202?

   011001010

6. What is the 9-bits two's complement representation of -162?

   101011110

3. RISC-V assembly

In this question we use RV32I, the RISC-V Instruction Set Architecture (ISA) **without** multiplications and divisions, and the ILP32 Application Binary Interface (ABI) seen during lectures and labs. Reminder: according the ILP32 ABI, the size of a stack frame **must** be a multiple of 16 bytes; the general purpose **saved** registers are `sp`, `gp`, `tp`, `s0`, `s1`, ..., `s11`; the general purpose **non-saved** registers are `ra`, `t0`, `t1`, ..., `t6`, `a0`, `a1`, ..., `a7`. Use the provided RISC-V cheat sheet if you don't remember the RV32I ISA or the ILP32 ABI.

Important: only basic instructions are allowed, pseudo-instructions are **forbidden**.

1. Assembly coding of function `foo`

   Write in RV32I assembly language a function named `foo` which input parameters A, B are two 32-bits unsigned numbers, and that returns $|A - B|$, the absolute value of their

difference. Your code **must** fully comply with the ILP32 ABI. The returned value **must** be always correct (no overflows). Comment each instruction on the same line after a `#` sign as in the following example:

```
addi t0,t1,0   # t0 = t1 + 0 = t1
```

Listing 1 shows a possible source code of `foo`.

```
.text
foo: # inputs A,B in a0,a1, output in a0
  bgeu a0,a1,foo1  # if a0>=a1 (A>=B) goto foo1; unsigned comparison
  sub  a0,a1,a0    # a0 = a1-a0 (B-A)
  jalr zero,0(ra)  # return
foo1:
  sub  a0,a0,a1    # a0 = a0-a1 (A-B)
  jalr zero,0(ra)  # return
```

Listing 1: The `foo` function

Per the ILP32 ABI, function inputs are passed in registers `a0`, `a1`, …, `a7` and results are returned in registers `a0` and `a1`. So, `foo` receives its two inputs A, B in `a0`, `a1`, and must return its final result in `a0`. We first test if $A \geq B$ (as A and B are unsigned we use the `bgeu` instruction, not `bge` that compares signed numbers); if it is, we go to `foo1`, compute `a0 = a0-a1` $= A - B$ and return. Else we compute `a0 = a1-a0` $= B - A$ and return. There is no need for a stack frame because we do not call other functions and we do not modify any saved register.

Note: we cannot subtract first, test the sign of the result, and take the negation if it is, as in:

```
sub a0,a0,a1
blt a0,zero,negative
...
negative:
sub a0,zero,a0
```

because the inputs are unsigned numbers; if $A > 2^{31}$ and $B = 0$, for example, $A - B > 2^{31}$ would be considered negative while it is not.

2. Assembly coding of function `bar`

   Assume function `foo` has been coded not by you but by one of your colleagues; you do not know anything about its code, except that it fully complies with the ILP32 ABI.

   Write in RV32I assembly language a function named `bar` which input parameters A, B, C, D are four 32-bits unsigned numbers, and that:

   - calls function `foo` to compute $X = |A - B|$,
   - calls function `foo` to compute $Y = |C - D|$,
   - calls function `foo` to compute $Z = |X - Y|$,
   - computes $T = \lfloor \frac{X+Y}{2} \rfloor$ where $\lfloor V \rfloor$ is the largest integral value not greater than V,
   - returns Z and T.

   Your code **must** also fully comply with the ILP32 ABI. The returned values **must** be always correct (no overflows). Comment each instruction on the same line after a `#` sign.

Listing 2 shows a possible source code of `bar`.

```
1  .text
2  bar: # inputs A,B,C,D in a0,a1,a2,a3, outputs Z,T in a0,a1
3    addi sp,sp,-16   # allocate 16 bytes stack frame
4    sw   ra,0(sp)    # save ra in stack frame
5    sw   a2,4(sp)    # save a2 (C) in stack frame
6    sw   a3,8(sp)    # save a3 (D) in stack frame
7    jal  ra,foo      # a0 = foo(a0,a1) (|A-B| = X)
8    sw   a0,12(sp)   # save a0 (X) in stack frame
9    lw   a0,4(sp)    # restore C from stack frame in a0
10   lw   a1,8(sp)    # restore D from stack frame in a1
11   jal  ra,foo      # a0 = foo(a0,a1) (|C-D| = Y)
12   lw   a1,12(sp)   # restore X from stack frame in a1
13   and  t0,a0,a1    # t0 = a0 and a1 (Y and X)
14   andi t0,t0,1     # t0 = t0 and 1 (U = Y and X and 1)
15   srli t1,a0,1     # t1 = a0 >> 1 (Y/2)
16   srli t2,a1,1     # t2 = a1 >> 1 (X/2)
17   add  t1,t1,t2    # t1 = t1 + t2 (Y/2 + X/2)
18   add  t1,t1,t0    # t1 = t1 + t0 (T = Y/2 + X/2 + U)
19   sw   t1,4(sp)    # save t1 (T) in stack frame
20   jal  ra,foo      # a0 = foo(a0,a1) (|Y-X| = |X-Y| = Z)
21   lw   a1,4(sp)    # restore T from stack frame in a1
22   lw   ra,0(sp)    # restore ra from stack frame
23   addi sp,sp,16    # restore sp
24   jalr zero,0(ra)  # return
```

Listing 2: The `bar` function

Per the ILP32 ABI, function inputs are passed in registers `a0`, `a1`, …, `a7` and results are returned in registers `a0` and `a1`. So, function `bar` receives its inputs A, B, C, D in `a0`, `a1`, `a2`, `a3`, and must return its final results Z, T in `a0`, `a1`. As `bar` calls another function (`foo`) it must allocate a stack frame (Line 3), at least to store its own return address that it receives in `ra` (Line 4). It also uses the stack frame to store values in non-saved registers that must be preserved across function calls. 16 bytes are enough.

`bar` saves `a2`, `a3` in its stack frame (Lines 5 and 6) before the first call of `foo` because `foo` could modify these non-saved registers. It then calls `foo` (Line 7) to compute $X = |A - B|$ (A is already in `a0` and B in `a1`); the result is returned in `a0` and immediately saved in the stack frame (Line 8). Next, C and D are restored from the stack frame, respectively in `a0` and `a1` (Lines 9 and 10), and `foo` is called again (Line 11) to compute $Y = |C - D|$ (again returned in `a0`). X is then restored from the stack frame in `a1` (Line 12).

What comes next is a way to compute $T = \left\lfloor \frac{X+Y}{2} \right\rfloor$ without overflows. It is based on the fact that $\left\lfloor \frac{X+Y}{2} \right\rfloor = \left\lfloor \frac{X}{2} \right\rfloor + \left\lfloor \frac{Y}{2} \right\rfloor + U$ where $U = 1$ if X and Y are odd, else $U = 0$. As $\left\lfloor \frac{X}{2} \right\rfloor, \left\lfloor \frac{Y}{2} \right\rfloor \in \{0, ..., 2^{31} - 1\}$ there cannot be an overflow when computing their sum. So, U is computed in `t0` with bitwise AND operations (Lines 13 and 14), Y and X are divided by 2 with logical right shifts (Lines 15 and 16), and $T = \left\lfloor \frac{X}{2} \right\rfloor + \left\lfloor \frac{Y}{2} \right\rfloor + U$ is computed in `t1` with regular `add` instructions (Lines 17 and 18). `t1` (containing T) is then saved in the stack frame (Line 19) because there is one more call to `foo` to come that could overwrite this non-saved register.

Finally, `foo` is called one last time with Y still in `a0` and X in `a1`, returning $Z = |X - Y|$ in `a0` (Line 20). T is restored from the stack frame in `a1` (Line 21); with Z in `a0` and T in `a1` the function is almost ready to return. Before that, it must only restore `ra` from the stack frame (Line 22), restore `sp` by adding 16 to it (Line 23), and return (Line 24).

RISC-V Instruction-Set

Erik Engheim <erik.engheim@ma.com>

Arithmetic Operation

Mnemonic	Instruction	Type	Description
ADD rd, rs1, rs2	Add	R	rd ← rs1 + rs2
SUB rd, rs1, rs2	Subtract	R	rd ← rs1 - rs2
ADDI rd, rs1, imm12	Add immediate	I	rd ← rs1 + imm12
SLT rd, rs1, rs2	Set less than	R	rd ← rs1 < rs2 ? 1 : 0
SLTI rd, rs1, imm12	Set less than immediate	I	rd ← rs1 < imm12 ? 1 : 0
SLTU rd, rs1, rs2	Set less than unsigned	R	rd ← rs1 < rs2 ? 1 : 0
SLTIU rd, rs1, imm12	Set less than immediate unsigned	I	rd ← rs1 < imm12 ? 1 : 0
LUI rd, imm20	Load upper immediate	U	rd ← imm20 << 12
AUIP rd, imm20	Add upper immediate to PC	U	rd ← PC + imm20 << 12

Logical Operations

Mnemonic	Instruction	Type	Description
AND rd, rs1, rs2	AND	R	rd ← rs1 & rs2
OR rd, rs1, rs2	OR	R	rd ← rs1 | rs2
XOR rd, rs1, rs2	XOR	R	rd ← rs1 ^ rs2
ANDI rd, rs1, imm12	AND immediate	I	rd ← rs1 & imm12
ORI rd, rs1, imm12	OR immediate	I	rd ← rs1 | imm12
XORI rd, rs1, imm12	XOR immediate	I	rd ← rs1 ^ imm12
SLL rd, rs1, rs2	Shift left logical	R	rd ← rs1 << rs2
SRL rd, rs1, rs2	Shift right logical	R	rd ← rs1 >> rs2
SRA rd, rs1, rs2	Shift right arithmetic	R	rd ← rs1 >> rs2
SLLI rd, rs1, shamt	Shift left logical immediate	I	rd ← rs1 << shamt
SRLI rd, rs1, shamt	Shift right logical imm.	I	rd ← rs1 >> shamt
SRAI rd, rs1, shamt	Shift right arithmetic immediate	I	rd ← rs1 >> shamt

Load / Store Operations

Mnemonic	Instruction	Type	Description
LD rd, imm12(rs1)	Load doubleword	I	rd ← mem[rs1 + imm12]
LW rd, imm12(rs1)	Load word	I	rd ← mem[rs1 + imm12]
LH rd, imm12(rs1)	Load halfword	I	rd ← mem[rs1 + imm12]
LB rd, imm12(rs1)	Load byte	I	rd ← mem[rs1 + imm12]
LWU rd, imm12(rs1)	Load word unsigned	I	rd ← mem[rs1 + imm12]
LHU rd, imm12(rs1)	Load halfword unsigned	I	rd ← mem[rs1 + imm12]
LBU rd, imm12(rs1)	Load byte unsigned	I	rd ← mem[rs1 + imm12]
SD rs2, imm12(rs1)	Store doubleword	S	rs2 → mem[rs1 + imm12]
SW rs2, imm12(rs1)	Store word	S	rs2(31:0) → mem[rs1 + imm12]
SH rs2, imm12(rs1)	Store halfword	S	rs2(15:0) → mem[rs1 + imm12]
SB rs2, imm12(rs1)	Store byte	S	rs2(7:0) → mem[rs1 + imm12]

Branching

Mnemonic	Instruction	Type	Description
BEQ rs1, rs2, imm12	Branch equal	SB	if rs1 = rs2 pc ← pc + imm12
BNE rs1, rs2, imm12	Branch not equal	SB	if rs1 ≠ rs2 pc ← pc + imm12
BGE rs1, rs2, imm12	Branch greater than or equal	SB	if rs1 ≥ rs2 pc ← pc + imm12
BGEU rs1, rs2, imm12	Branch greater than or equal unsigned	SB	if rs1 >= rs2 pc ← pc + imm12
BLT rs1, rs2, imm12	Branch less than	SB	if rs1 < rs2 pc ← pc + imm12
BLTU rs1, rs2, imm12	Branch less than unsigned	SB	if rs1 < rs2 pc ← pc + imm12 << 1
JAL rd, imm20	Jump and link	UJ	rd ← pc + 4 pc ← pc + imm20
JALR rd, imm12(rs1)	Jump and link register	I	rd ← pc + 4 pc ← rs1 + imm12

Pseudo Instructions

Mnemonic	Instruction	Base instruction(s)
LI rd, imm12	Load immediate (near)	ADDI rd, zero, imm12
LI rd, imm	Load immediate (far)	LUI rd, imm[31:12] ADDI rd, rd, imm[11:0]
LA rd, sym	Load address (far)	AUIPC rd, sym[31:12] ADDI rd, rd, sym[11:0]
MV rd, rs	Copy register	ADDI rd, rs, 0
NOT rd, rs	One's complement	XORI rd, rs, -1
NEG rd, rs	Two's complement	SUB rd, zero, rs
BGT rs1, rs2, offset	Branch if rs1 > rs2	BLT rs2, rs1, offset
BLE rs1, rs2, offset	Branch if rs1 ≤ rs2	BGE rs2, rs1, offset
BGTU rs1, rs2, offset	Branch if rs1 > rs2 (unsigned)	BLTU rs2, rs1, offset
BLEU rs1, rs2, offset	Branch if rs1 ≤ rs2 (unsigned)	BGEU rs2, rs1, offset
BEQZ rs1, offset	Branch if rs1 = 0	BEQ rs1, zero, offset
BNEZ rs1, offset	Branch if rs1 ≠ 0	BNE rs1, zero, offset
BGEZ rs1, offset	Branch if rs1 ≥ 0	BGE rs1, zero, offset
BLEZ rs1, offset	Branch if rs1 ≤ 0	BGE zero, rs1, offset
BGTZ rs1, offset	Branch if rs1 > 0	BLT zero, rs1, offset
J offset	Unconditional jump	JAL zero, offset
CALL offset12	Call subroutine (near)	JALR ra, ra, offset12
CALL offset	Call subroutine (far)	AUIPC ra, offset[31:12] JALR ra, ra, offset[11:0]
RET	Return from subroutine	JALR zero, 0(ra)
NOP	No operation	ADDI zero, zero, 0

32-bit instruction format

	31 30 29 28 27 26 25	24 23 22 21 20	19 18 17 16 15	14 13 12	11 10 9 8 7	6 5 4 3 2 1 0
R	func	rs2	rs1	func	rd	opcode
I	immediate		rs1	func	rd	opcode
SB	immediate	rs2	rs1	func	immediate	opcode
UJ	immediate				rd	opcode

Register File

r0	r1	r2	r3
r4	r5	r6	r7
r8	r9	r10	r11
r12	r13	r14	r15
r16	r17	r18	r19
r20	r21	r22	r23
r24	r25	r26	r27
r28	r29	r30	r31

Register Aliases

zero	ra	sp	gp
tp	t0	t1	t2
s0/fp	s1	a0	a1
a2	a3	a4	a5
a6	a7	s2	s3
s4	s5	s6	s7
s8	s9	s10	s11
t3	t4	t5	t6

ra - return address
sp - stack pointer
gp - global pointer
tp - thread pointer

t0 - t6 - Temporary registers
s0 - s11 - Saved by callee
a0 - a7 - Function arguments
a0 - a1 - Return value(s)