Introduction to Computer Architecture: exam

R. Pacalet

2024-12-11

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.

The first question is worth 6 points. The second and third questions are worth 7 points each.

1 CMOS logic

The `2-1 OAI` 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	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

1. Draw the CMOS schematic of `2-1 OAI` using only N and P transistors.

2. Write the boolean equation of the **X** output of **2-1 OAI** 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 **2-1 OAI** and draw it.

1. The CMOS schematic is represented on Figure 1.

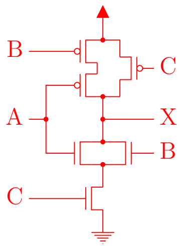

Figure 1: CMOS schematic of the **2-1 OAI** gate

We can observe that the **2-1 OAI** output is **0** if and only if the **C** input is **1** and at least one of **A** or **B** is also **1**. This immediately gives the network of N transistors between the ground and the **2-1 OAI** output: a N transistor which grid is **C** in a series with a group of 2 N transistors in parallel which grids are **A** and **B** respectively. This way, if **C** is **0** or if **A** and **B** are **0** there is no path between the ground and the output, while in all other circumstances there is such a path and the output is **0**. As always with CMOS logic the network of P transistors between the power supply and the **2-1 OAI** output is dual of the network of N transistors: a P transistor which grid is **C** in parallel with a group of 2 P transistors in a series which grids are **A** and **B** respectively.

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

3. Using the style seen in class we could represent the **2-1 OAI** gate as shown on Figure 2.

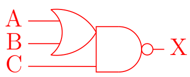

Figure 2: Symbol of the **2-1 OAI** gate

2 Binary representation of data

There are several ways to represent signed integers using bits. In computer systems, the two most frequently encountered are *sign and magnitude* and *two's complement*. In the following we denote $a_{n-1}a_{n-2}\dots a_1a_0$ the n-bits representation of integer A where a_0 is the *Least Significant Bit* (LSB).

1. Consider decimal values 12, -59 and -66. We want to represent them all in *two's complement* on the same number of bits m. What is the minimum value of m?

2. Consider decimal values 12, -59 and -66. Convert them in m-bits *two's complement* (where m is your answer to the preceding question).

3. A p-bits adder is a hardware device that takes two p-bits inputs, adds them as if they were unsigned integers, and outputs the $p+1$-bits result. We denote $A = a_{p-1} \dots a_1 a_0$, $B = b_{p-1} \dots b_1 b_0$ the two p-bits inputs and $S = s_p \dots s_2 s_1 s_0$ the $p+1$-bits output of a p-bits adder. Example: with a 3-bits adder, if inputs are $A = 101(5)$ and $B = 011(3)$, the output is $S = 1000(8)$. If, instead of considering the inputs and the output as unsigned integers, we consider them as signed numbers represented in *sign and magnitude*, the result is sometimes correct, sometimes not.

   - Give an example of two 3-bits *sign and magnitude* inputs for which the output of a 3-bits adder is the correct 4-bits *sign and magnitude* representation of their sum.
   - Give an example of two 3-bits *sign and magnitude* inputs for which the output of a 3-bits adder is **not** the correct 4-bits *sign and magnitude* representation of their sum.
   - Express the necessary and sufficient condition on inputs $A = a_{p-1} \dots a_1 a_0$ and $B = b_{p-1} \dots b_1 b_0$ such that a p-bits adder outputs the correct *sign and magnitude* representation of their sum.

4. What is the tetradecimal (base 14, symbols $0, 1, 2 \dots 9, A, B, C, D$) representation of decimal value 604?

1. 8

   8 bits are sufficient: $-2^7 = -128 \leq -66, -59, 12 \leq 2^7 - 1 = 127$ but 7 are not: $-66 < -2^6 = -64$. So 8 bits are the minimum.

2. $00001100_{2'scomp.} = 12_{10}$
   $11000101_{2'scomp.} = -59_{10}$
   $10111110_{2'scomp.} = -66_{10}$

3. Signed additions in sign and magnitude with a p-bits adder

   - $000_{sign-mag.} + 000_{sign-mag.} = 0000_{sign-mag.}(0_{10} + 0_{10} = 0_{10})$ is correct.
   - $101_{sign-mag.} + 001_{sign-mag.} = 0110_{sign-mag.}(-1_{10} + 1_{10} = 6_{10})$ is not correct.
   - We denote $\mathbf{0}^n$ the all zero n-bits string and $\mathbf{?}^n$ any n-bits string. We denote $Q(A, B)$ the property "*the output of the p-bits adder is the correct sign and magnitude representation of the sum of the A and B inputs considered as signed numbers in sign and magnitude representation*". $Q(A, B)$ is true if and only if one of the 3 following conditions is satisfied:

$a_{p-1} = b_{p-1}$

$A = 01\mathbf{0}^{p-2}$ and $B = 11\mathbf{?}^{p-2}$

$A = 11\mathbf{?}^{p-2}$ and $B = 01\mathbf{0}^{p-2}$

Proof: for any n bits string $X = x_{n-1}x_{n-2}\dots x_1x_0$ we denote $u(X)$ the unsigned integer it represents and $sm(X)$ the signed integer it represents in sign and magnitude. Example: if $X = 1001$, $u(X) = 9$ and $sm(X) = -1$. With these notations we can model the output $S = A + B$ of the p-bits adder with $u(S) = u(A) + u(B)$ and the question becomes "under what condition do we also have $sm(S) = sm(A) + sm(B)$?" By definition of unsigned and sign and magnitude representations we have:

$$\begin{aligned} sm(X) &= (-1)^{x_{n-1}}(u(X) - x_{n-1}2^{n-1}) \\ &= (-1)^{x_{n-1}}u(X) + x_{n-1}2^{n-1} \\ u(X) &= (-1)^{x_{n-1}}sm(X) + x_{n-1}2^{n-1} \end{aligned}$$

From which we can rewrite $u(S) = u(A) + u(B)$ as:

$$\begin{aligned} &(-1)^{s_p}sm(S) + s_p2^p = \\ &(-1)^{a_{p-1}}sm(A) + a_{p-1}2^{p-1} + (-1)^{b_{p-1}}sm(B) + b_{b-1}2^{p-1} \end{aligned} \quad (1)$$

By definition of the p-bits adder, $a_{p-1} = b_{p-1} = 0 \Rightarrow s_p = 0$, and $a_{p-1} = b_{p-1} = 1 \Rightarrow s_p = 1$. This leaves only 6 possible cases for a_{p-1}, b_{p-1} and s_p:

1. $a_{p-1} = b_{p-1} = s_p = 0$. Equation 1 becomes $sm(S) = sm(A) + sm(B) \Leftrightarrow Q(A,B)$.
2. $a_{p-1} = b_{p-1} = s_p = 1$. Equation 1 becomes $-sm(S) + 2^p = -sm(A) + 2^{p-1} - sm(B) + 2^{p-1} \Leftrightarrow sm(S) = sm(A) + sm(B) \Leftrightarrow Q(A,B)$.
3. $a_{p-1} = 0, b_{p-1} = 1, s_p = 0$. Equation 1 becomes $sm(S) = sm(A) - sm(B) + 2^{p-1}$. And we immediately see that we cannot also have $sm(S) = sm(A) + sm(B)$ because this would require $sm(B) = -sm(B) + 2^{p-1} \Leftrightarrow sm(B) = 2^{p-2} \Leftrightarrow B = 01\mathbf{0}^{p-2}$ which contradicts the $b_{p-1} = 1$ hypothesis.
4. $a_{p-1} = 1, b_{p-1} = 0, s_p = 0$. Equation 1 becomes $sm(S) = -sm(A) + 2^{p-1} + sm(B)$. And we immediately see that we cannot also have $sm(S) = sm(A) + sm(B)$ because this would require $sm(A) = -sm(A) + 2^{p-1} \Leftrightarrow sm(A) = 2^{p-2} \Leftrightarrow A = 01\mathbf{0}^{p-2}$ which contradicts the $a_{p-1} = 1$ hypothesis.
5. $a_{p-1} = 0, b_{p-1} = 1, s_p = 1$. Equation 1 becomes $-sm(S) + 2^p = sm(A) - sm(B) + 2^{p-1} \Leftrightarrow sm(S) = -sm(A) + sm(B) + 2^{p-1}$. And we immediately see that we can also have $sm(S) = sm(A) + sm(B)$ if and only if $sm(A) = -sm(A) + 2^{p-1} \Leftrightarrow sm(A) = 2^{p-2} \Leftrightarrow A = 01\mathbf{0}^{p-2}$.

6. $a_{p-1} = 1, b_{p-1} = 0, s_p = 1$. Equation 1 becomes $-sm(S) + 2^p = -sm(A) + 2^{p-1} + sm(B) \Leftrightarrow sm(S) = sm(A) - sm(B) + 2^{p-1}$. And we immediately see that we can also have $sm(S) = sm(A) + sm(B)$ if and only if $sm(B) = -sm(B) + 2^{p-1} \Leftrightarrow sm(B) = 2^{p-2} \Leftrightarrow B = 01\mathbf{0}^{p-2}$.

The 2 first cases can be merged as condition $a_{p-1} = b_{p-1}$ (A and B have same leftmost bit). This ensures property $Q(A, B)$.

The fifth case must be slightly refined to translate the $s_p = 1$ hypothesis into a constraint on A and B. With $A = 01\mathbf{0}^{p-2}$ and $b_{p-1} = 1$ we can have $s_p = 1$ if and only if $b_{p-2} = 1$. The condition on A and B is thus $A = 01\mathbf{0}^{p-2}$ and $B = 11\mathbf{?}^{p-2}$. This also ensures property $Q(A, B)$.

The sixth case is the symmetrical of the fifth: $A = 11\mathbf{?}^{p-2}$ and $B = 01\mathbf{0}^{p-2}$. This also ensures property $Q(A, B)$.

4. $604 = 3 \times 196 + 1 \times 14 + 2 = 3 \times 14^2 + 1 \times 14^1 + 2 \times 14^0 = 312_{14}$

3 RISC-V assembly

In this question we use RV32IM, the RISC-V Instruction Set Architecture (ISA) and the ILP32 Application Binary Interface (ABI) seen during lectures and labs. Reminder: the size of a stack frame **must** be at least 16 bytes and **must** be a multiple of 16 bytes; the general purpose **saved** registers are `sp`, `gp`, `tp`, `s0`, `s1`, . . . , `s11`. Use the provided RISC-V cheat sheet if you don't remember the RV32IM ISA or the ILP32 ABI.

Study the code of function `qux` in Listing 4.

```
qux:
  addi sp,sp,-16
  sw   ra,0(sp)
  mv   s0,a0
  mv   a0,zero
L1:
  andi t1,s0,1
  add  a0,a0,t1
  srli s0,s0,1
  addi a1,a1,-1
  bne  a1,zero,L1
  lw   ra,0(sp)
  addi sp,sp,16
  ret
```

Listing 1: Listing of function ‘qux‘

How many input parameters does function `qux` take? In what register(s)? How many output results does function `qux` return? In what register(s)? Explain what function `qux` does.

Function `qux` has 2 input parameters passed in registers `a0` and `a1`. It returns one result in register `a0`. Function `qux` counts the number of bits equal to one among the `a1` right bits of `a0`.

Does function **qux** fully comply with the ILP32 ABI? Explain why. If it does not explain how to fix it.

No, function **qux** does not fully comply with the ILP32 ABI. It uses and modifies saved register **s0** without saving its content first and restoring it before it returns to the caller. This is a violation of the ABI and can cause errors if the this register is in use when function **qux** is called. The function could be fixed by storing the content of **s0** in the stack frame before modifying it, and by restoring it from the stack frame before returning, as it does for register **ra**:

```
qux:
  addi sp,sp,-16     # allocate a 16 bytes stack frame
  sw   ra,0(sp)      # save ra in stack frame
  sw   s0,4(sp)      # save s0 in stack frame
  mv   s0,a0         # s0 <- a0
  mv   a0,zero       # a0 <- 0 (initialize bit count)
L1:
  andi t1,s0,1       # t1 <- s0 AND 1 (righmost bit of s0)
  add  a0,a0,t1      # a0 <- a0 + t1 (count)
  srli s0,s0,1       # s0 <- s0 >> 1 (logical right shift)
  addi a1,a1,-1      # a1 <- a1 - 1 (loop index)
  bne  a1,zero,L1    # goto L1 if a1 != 0
  lw   s0,4(sp)      # restore s0 from stack frame
  lw   ra,0(sp)      # restore ra from stack frame
  addi sp,sp,16      # restore sp
  ret                # return to caller
```

Listing 2: Listing of function ‘qux‘

Could function **qux** be optimized for speed? If yes explain how.

Yes, it could be optimized for speed by completely avoiding the use of saved registers and the need to save and restore them in the stack frame. As it does not call another function it could even avoid saving and restoring **ra**. Thanks to this there is no need to allocate and deallocate a stack frame and 6 instructions are saved:

```
qux:
  mv   t0,a0         # t0 <- a0
  mv   a0,zero       # a0 <- 0 (initialize bit count)
L1:
  andi t1,t0,1       # t1 <- t0 AND 1 (righmost bit of t0)
  add  a0,a0,t1      # a0 <- a0 + t1 (count)
  srli t0,t0,1       # t0 <- t0 >> 1 (logical right shift)
  addi a1,a1,-1      # a1 <- a1 - 1 (loop index)
  bne  a1,zero,L1    # goto L1 if a1 != 0
  ret                # return to caller
```

Listing 3: listing of function ‘qux‘

Another possible optimization would be to stop counting as soon as the shifted value is equal to 0:

```
qux:
  mv    t0,a0         # t0 <- a0
  mv    a0,zero       # a0 <- 0 (initialize bit count)
L1:
  beq   t0,zero,L2    # goto L2 if t0 == 0 (early exit)
  andi  t1,t0,1       # t1 <- t0 AND 1 (righmost bit of t0)
  add   a0,a0,t1      # a0 <- a0 + t1 (count)
  srli  t0,t0,1       # t0 <- t0 >> 1 (logical right shift)
  addi  a1,a1,-1      # a1 <- a1 - 1 (loop index)
  bne   a1,zero,L1    # goto L1 if a1 != 0
L2:
  ret                 # return to caller
```

Listing 4: listing of function ‘qux‘

Note that, as this adds one more test, whether it optimizes or not depends on the input parameters. For instance, with `a0 = 0` and `a1 = 32`, this last optimization completely skips the loop, that is, $5 \times 32 = 160$ instructions. However, with `a0 = 0xffffffff` and `a1 = 32`, it adds 32 test, that is, 32 instructions. Deciding to use this optimization or not shall thus depend on the statistics of the input parameters.

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

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 - 17 - Function arguments
a0 - a1 - Return value(s)

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