Introduction to Computer Architecture: exam

R. Pacalet

2023-11-30

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 (6 points)

In the CMOS logic gate of Figure 1 a, b, c are the 3 inputs, x is the output.

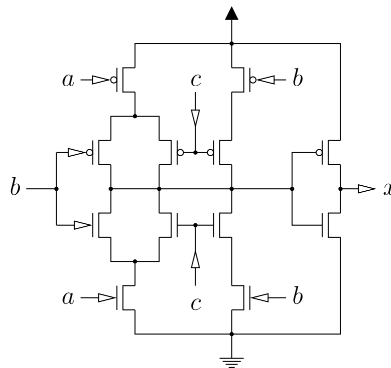

Figure 1: CMOS schematic of logic gate

1. Write its truth table, that is, the table giving the value of x for each of the 8 value combinations of a, b and c.
2. Write the corresponding boolean equation using only symbols a, b, c, parentheses and the boolean operators NOT, OR, AND. Do not assume precedence between boolean operators, use parentheses to make your equation non ambiguous. Example of non ambiguous boolean equation: $\text{NOT}(((\text{NOT}\, a)\, \text{AND}\, c)\, \text{OR}\, b)$.

1. The truth table is:

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

   Building the truth table is straightforward after we observe that there are 2 logic gates in one: the two rigthmost transistors form an inverter that inverts the output of the left part. We can thus first express the condition on the inputs values for the output of the left gate to be `0`, that is, the condition for the network of N transistors to be passing: `(a AND (b OR c)) OR (b and c)`. Next we fill the table with the inverse (because of the inverter on the right), that is, set x to `1` when the condition is true, `0` when it is false.

2. $x = (a\, \text{AND}\, (b\, \text{OR}\, c))\, \text{OR}\, (b\, \text{AND}\, c)$

2 RISC-V assembly (8 points)

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 32 bytes and **must** be a multiple of 32 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.

Functions `foo` and `bar` take a 32 bits input and return a 32 bits result. `bar` has already been coded in RV32IM assembly. It fully complies with the ILP32 ABI but you **do not know how it has been coded**: it possibly modifies **any** non-saved general purpose register (except, of course, the `zero` register).

`foo` calls `bar` twice: first on its input parameter and then on the result. It returns the bitwise AND of the 2 `bar` results. The pseudo-code of `foo` could be the following, in which `val` represents the input of `foo`:

```
foo(val) {
```

```
  tmp = bar(val)
  return tmp AND bar(tmp)
}
```

Write the code of function `foo` in RV32IM assembly. Your code **must** fully comply with the ILP32 ABI. Comment each instruction on the same line after a `#` sign as in the following example:

```
add  t0,t1,t2  # t0 = t1 + t2
```

```
.text
foo:
  addi sp,sp,-32    # allocate 32 bytes stack frame
  sw   ra,0(sp)     # save ra in stack frame
  sw   s0,4(sp)     # save s0 in stack frame
  jal  ra,bar       # a0 = bar(val)
  mv   s0,a0        # s0 = bar(val)
  jal  ra,bar       # a0 = bar(bar(val))
  and  a0,s0,a0     # a0 = bar(val) AND bar(bar(val))
  lw   s0,4(sp)     # restore s0 from stack frame
  lw   ra,0(sp)     # restore ra from stack frame
  addi sp,sp,32     # restore sp
  jalr zero,0(ra)   # return
```

As for any function we first allocate a stack frame to store the return address (in register `ra`) and all *saved* register that we intend to use (register `s0` in our case). 32 bytes are more than the needed 8 bytes but the recommendation is that the size of the stack frames shall be a multiple of 32 bytes. Note: in the real ABI the recommendation is 16 bytes, not 32. As always the stack grows towards the low addresses so we **subtract** 32 from the stack pointer (register `sp`). `sp` now contains the base address of the new stack frame.

Next we save registers `ra` and `s0` in the stack frame, respectively at offset 0 and 4 from the base address of the new stack frame.

Next we call function `bar`. The input parameter is already in register `a0` because it is the same as the input parameter of function `foo` and we did not modify it since the entry in `foo`. The call is simply `jal ra, bar`, or the equivalent pseudo-instruction `call bar`, that stores the return address in `ra` and jumps at the first instruction of `bar`. After the function returns we must save the result (also in register `a0`) because we will need it for the final computation while the second call to `bar` will overwrite register `a0`. So, we copy it in saved register `s0`. We cannot use a temporary register or one of the `a0`, . . . , `a7` registers because these are not saved registers and the call to `bar` could modify them. This is the reason why we chose `s0`, and also why we first saved this register in the stack frame: we modify its content so we must be able to restore it before we return.

Next we call function `bar` a second time. Again, the input parameter is already in register `a0` because it is the result of the first call and we did not modify it. After the function returns the result is in register `a0`, the result of the first call is in register `s0` (and the call did not modify it because `s0` is a saved register).

We thus simply compute the bitwise AND of `a0` and `s0`, and store the result in `a0`, because it is where any function like `foo` must store its result.

The final part simply consists in restoring `ra` and `s0` from the stack frame, restoring `sp` by adding the opposite of what was subtracted when entering the function (32), and returning with `jalr zero,0(ra)` or the equivalent pseudo-instruction `ret`.

Note: instead of using `s0` to store the result of the first call we could have saved it in the stack frame. There would be no need to save, use, and restore `s0`. This new version has 1 instruction less than the other:

```
.text
foo:
  addi sp,sp,-32    # allocate 32 bytes stack frame
  sw   ra,0(sp)     # save ra in stack frame
  jal  ra,bar       # a0 = bar(val)
  sw   a0,4(sp)     # save a0 in stack frame
  jal  ra,bar       # a0 = bar(bar(val))
  lw   t0,4(sp)     # restore first result in t0 from stack frame
  and  a0,t0,a0     # a0 = bar(val) AND bar(bar(val))
  lw   ra,0(sp)     # restore ra from stack frame
  addi sp,sp,32     # restore sp
  jalr zero,0(ra)   # return
```

3 Binary representation of numbers (6 points)

All given integer values are in base 10.

1. We want to represent -256, $+267$, -169 and $+63$ in 2's complement on the same number of bits n. What is the minimum value of n (only one answer)?
2. Write the n bits 2's complement representation of -256, $+267$, -169 and $+63$.
3. We consider a 32 bits signed integer A which 2's complement representation $a_{31}a_{30}\dots a_1a_0$ is stored in RISC-V general purpose register `t0`. We shift `t0` to the left by 2 positions with the RISC-V instruction `slli t0,t0,2` and denote B the new `t0` content, considered as a 32 bits signed integer in 2's complement. Under what condition on A do we have $B = 4 \times A$?

1. $n = 10$
2. `1100000000`, `0100001011`, `1101010111` and `0000111111`
3. $a_{31} = a_{30} = a_{29}$, that is, $-2^{29} \leq A < 2^{29}$

1. $n = 10$ because $+267$ has the largest absolute value of the 4 numbers, and 9 bits can only represent numbers in the $[-2^8 \dots 2^8 - 1]$ range, that is, $[-256 \dots 255]$. This is not enough while with 10 bits we can represent numbers in the $[-2^9 \dots 2^9 - 1]$ range, that is, $[-512 \dots 511]$.
2. Converting from decimal to 2's complement on 10 bits simply consists in decomposing the absolute values in powers of 2, and, for the negative numbers, taking the one's complement and adding 1. Example for -256:

$256 = 2^8 = 0100000000_2 \rightarrow 1011111111 + 1 = 1100000000$. Example for 267: $267 = 256 + 8 + 2 + 1 = 2^8 + 2^3 + 2^1 + 2^0 = 0100001011_2$.

3. The value of A is:

$$A = -2^{31} \times a_{31} + \sum_{i=0}^{i=30} 2^i \times a_i$$

When shifting to the left, the two leftmost bits (a_{31} and a_{30}) are dropped and two 0 bits enter on the right. So, the value of B is:

$$\begin{aligned} B &= -2^{31} \times a_{29} + \sum_{i=0}^{i=28} 2^{i+2} \times a_i \\ &= 2^2 \times \left(-2^{29} \times a_{29} + \sum_{i=0}^{i=28} 2^i \times a_i \right) \\ &= 2^2 \times \left(A + 2^{31} \times a_{31} - 2^{30} \times a_{30} - 2^{29} \times a_{29} - 2^{29} \times a_{29} \right) \\ &= 2^2 \times \left(A + 2^{31} \times a_{31} - 2^{30} \times a_{30} - 2^{30} \times a_{29} \right) \end{aligned}$$

So, $B = 4 \times A$ if and only is $2^{31} \times a_{31} - 2^{30} \times a_{30} - 2^{30} \times a_{29} = 0$, that is, $a_{31} = a_{30} = a_{29}$. The 3 leftmost (most significant) bits of A must be equal. We can thus rewrite A as:

$$\begin{aligned} A &= \left(-2^{31} + 2^{30} + 2^{29} \right) \times a_{29} + \sum_{i=0}^{i=28} 2^i \times a_i \\ &= -2^{29} \times a_{29} + \sum_{i=0}^{i=28} 2^i \times a_i \\ &\Rightarrow -2^{29} \leq A < 2^{29} \end{aligned}$$

Said differently, $B = 4 \times A$ if and only if the 32 bits number A would also fit on 30 bits only.

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