Computer Architecture exam (2 hours)

Renaud Pacalet - 2025-03-19

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 questions are worth 10 points each.

1. Gray code counter

With the standard binary representation of unsigned numbers a value can differ from the next in more than one bit. On 3 bits for instance, the standard binary representation of 5 is 101, which differs in 2 bits from the representation of 6 (110).

A Gray code (after Frank Gray, a physicist and researcher at Bell Labs) is another way of representing unsigned numeric values in binary such that any value differs from its predecessor and successor values in only one bit, and the representation of the largest value ($2^n - 1$ on n bits) also differs in only one bit from the representation of 0. Gray codes are used in various circumstances, for instance to exchange information between different synchronous digital designs with different clock frequencies.

- Invent a 3-bits Gray code to represent numeric values 0 to 7. Represent your solution in the form of a table like Table 1 where you will fill the third row. The representations of 0 and 7 are already provided, do not change them (note that they differ in only one bit):

Value	0	1	2	3	4	5	6	7
Standard	`000`	`001`	`010`	`011`	`100`	`101`	`110`	`111`
Gray code	`000`							`100`

Table 1: Incomplete Gray code

A possible Gray code is shown in Table 2.

Value	0	1	2	3	4	5	6	7
Standard	`000`	`001`	`010`	`011`	`100`	`101`	`110`	`111`
Gray code	`000`	`001`	`011`	`010`	`110`	`111`	`101`	`100`

Table 2: Gray code

- We want to design the combinatorial circuit named `next_gc` that computes the successor value of an input Gray code; when the input is the code of 7 (`100`) it shall output the code of 0 (`000`). The external view of `next_gc` is represented on Figure 1 where `A`, `B`, `C` are respectively the left, middle and right bits of the input Gray code and `X`, `Y`, `Z` are the left, middle and right bits of the output Gray code.

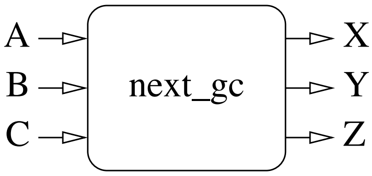

Figure 1: The external view of `next_gc`

So, if the input is set to `ABC = 100`, after the propagation delay, the output shall become `XYZ = 000`. Same for the 7 other input values, according the table of the Gray code you designed.

Design the schematic of the internals of `next_gc` using only the logic gates and symbols of Figure 2. Try to optimize your design such that it uses as few hardware as possible.

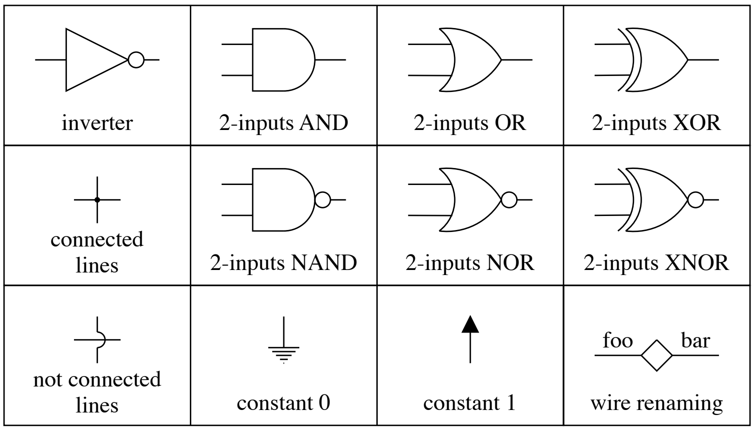

Figure 2: Logic gates and symbols

The truth table of `XYZ` shown in Table 3 is easily deduced from the previous table.

ABC	000	001	011	010	110	111	101	100
XYZ	001	011	010	110	111	101	100	000

Table 3: Truth table

From which we can deduce the 3 boolean equations:

- $X = (B \text{ and } (\text{not } C)) \text{ or } (A \text{ and } C)$
- $Y = (B \text{ and } (\text{not } C)) \text{ or } ((\text{not } A) \text{ and } C)$
- $Z = ((\text{not } A) \text{ and } (\text{not } B)) \text{ or } (A \text{ and } B) = A \text{ xnor } B$

From which we can draw the schematic shown on Figure 3.

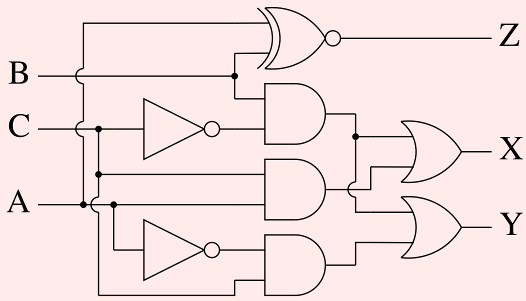

Figure 3: Schematic of the `next_gc` circuit

2. 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 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 RV32IM ISA or the ABI.

Function `avg2` has already been coded by one of your colleagues. You do not know its code, you do not know which registers it uses, you **must** assume that it can modify **any** non-saved registers. All you know is that:

- it is **fully** ABI compliant,
- its inputs are two 32 bits unsigned integers, denoted x and y in the following,
- its single output is a 32 bits unsigned integer, $\lfloor \frac{x+y}{2} \rfloor$, the rounded average of the two inputs; the rounding is toward 0 ($\lfloor \frac{1}{2} \rfloor = 0$),
- it is written in a way that completely avoids overflows, the returned value is always correct.

Assembly coding

Write in RV32IM assembly the code of function `avg4` that takes four 32 bits unsigned integers as inputs (denoted x, y, z, t in the following), calls `avg2` three times to compute their average rounded toward zero, and returns it. Your code **must** fully comply with the ABI. Comment each instruction on the same line after a `#` sign as in the following example:

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

Assembly coding

Listing 1 shows a possible source code of `avg4`. Note that it does **not** compute exactly the average of x, y, z, t rounded toward zero (see the extended explanations).

```
.text
# inputs x,y,z,t in a0,a1,a2,a3 respectively
# output in a0
avg4:
  addi sp,sp,-16   # allocate 16 bytes stack frame
  sw   ra,0(sp)    # save ra in stack frame
  sw   a2,4(sp)    # save z in stack frame
  sw   a3,8(sp)    # save t in stack frame
  call avg2        # a0 = avg2(x,y)
  sw   a0,12(sp)   # save a0 = avg2(x,y) in stack frame
  lw   a0,4(sp)    # a0 = z
  lw   a1,8(sp)    # a1 = t
  call avg2        # a0 = avg2(z,t)
  lw   a1,12(sp)   # a1 = avg2(x,y)
  call avg2        # a0 = avg2(a0,a1) = avg2(avg2(z,t), avg2(x,y))
  lw   ra,0(sp)    # restore ra from stack frame
  addi sp,sp,16    # restore sp
  ret              # return
```

Listing 1: The `avg4` function

Per the ABI, function inputs are passed in `a0`, `a1`, …, `a7` and results are returned in `a0` and `a1`. So, we receive our four inputs x, y, z, t in `a0`, `a1`, `a2` and `a3`, and we must store our final result in `a0` before returning. We call `avg2` three times and each time we pass the two inputs in `a0` and `a1`, and get the result in `a0`:

- a first time to compute the average of x and y,
- a second time to compute the average of z and t,
- a third time to compute the average of the two averages.

As for any function that calls other functions we **must** allocate a stack frame and store register `ra` in it before calling any other function. This is because it contains our return address and any call to other functions overwrites it. If we do not first save it, our own return address is lost and we cannot return to our own caller.

There are three other important data that we must save:

- As they are **non-saved** registers, the first call to `avg2` could modify `a2` and `a3` that contain our z and t inputs; so we **must** absolutely save them somewhere before the first call. We could copy them in **saved** registers but we would then have to first save these **saved** registers in the stack frame because, per the ABI, we must restore them before returning. And of course we would have to restore them from the stack frame before returning. We simplify a bit and save several instructions by saving `a2` and `a3` directly in the stack frame.
- The result of the first call to `avg2` is returned in `a0` that we need for the third call; so we **must** also absolutely save it somewhere before the second call; for the same reasons we also save it in the stack frame.

16 bytes are sufficient to store these four 4-bytes values. As always the stack grows toward the low addresses so we start our function by **subtracting** 16 from the stack pointer `sp` (Line 5). `sp` now contains the base address of the new stack frame.

We save registers `ra`, `a2` and `a3` in the stack frame, at offsets 0, 4 and 8 from the base address of the new stack frame, respectively (Lines 6 - 8).

We call function `avg2` that returns the average of x and y in `a0` (Line 9). The input parameters are already in registers `a0` and `a1` because we did not modify them since the entry in `avg4`. We save the result `a0` in the stack frame, at offset 12 from the base address of the new stack frame (Line 10).

We restore z and t from the stack frame in registers `a0` and `a1` (Lines 11 - 12), and we call `avg2` a second time (Line 13); it returns the average of z and t in `a0`. We restore the result of the first call from the stack frame in register `a1` (Line 14), and we call `avg2` a last time (Line 15); it returns the average of the two averages in `a0`, which is where we want it per the ABI.

The final part simply consists in restoring `ra` from the stack frame (Line 16), restoring `sp` by adding 16, the opposite of what was subtracted when entering the function (Line 17), and returning with pseudo-instruction `ret` (Line 18).

Note that the implementation of Listing 1 does not compute exactly the average rounded toward zero of the four parameters: if we call `avg4` with $x = 0, y = 1, z = 1, t = 2$, the returned value is $\left\lfloor\frac{\lfloor\frac{1}{2}\rfloor+\lfloor\frac{3}{2}\rfloor}{2}\right\rfloor = \left\lfloor\frac{1}{2}\right\rfloor = 0$, while $\left\lfloor\frac{0+1+1+2}{4}\right\rfloor = 1$. It can be shown that this computing error happens if and only if $(x + y) \bmod 4 = 1$ and $(z + t) \bmod 4 = 3$ or the opposite. This can easily be detected beforehand by computing $a = (x \oplus y) \wedge 3$ and $b = (z \oplus t) \wedge 3$, where $\oplus$ and $\wedge$ are the bitwise exclusive OR and the bitwise AND, respectively. If $a \wedge b = 1$ and $a \oplus b = 2$ we are in the error case and adding 1 to any of the four parameters fixes the issue.

Listing 2 shows a fixed source code of `avg4` where the error detection and fix are between Lines 6 and 16.

```
1  avg4_fixed:
2    addi sp,sp,-16   # allocate 16 bytes stack frame
3    sw   ra,0(sp)    # save ra in stack frame
4    sw   a2,4(sp)    # save z in stack frame
5    sw   a3,8(sp)    # save t in stack frame
6    xor  t0,a0,a1    # t0 = x xor y
7    andi t0,t0,3     # t0 = t0 and 3 = a
8    xor  t1,a2,a3    # t1 = z xor t
9    andi t1,t1,3     # t1 = t1 and 3 = b
10   and  t2,t0,t1    # t2 = t0 and t1 = a and b
11   addi t2,t2,-1    # t2 = t2 - 1
12   bne  t2,zero,ok  # if t2 != 0 goto ok (no need to fix)
13   xor  t3,t0,t1    # t3 = t0 xor t1 = a xor b
14   addi t3,t3,-2    # t3 = t3 - 2
15   bne  t3,zero,ok  # if t3 != 0 goto ok (no need to fix)
16   addi a0,a0,1     # a0 = a0 + 1 = x + 1 (fix)
17 ok:
18   call avg2        # a0 = avg2(x,y)
19   sw   a0,12(sp)   # save a0 = avg2(x,y) in stack frame
20   lw   a0,4(sp)    # a0 = z
21   lw   a1,8(sp)    # a1 = t
22   call avg2        # a0 = avg2(z,t)
23   lw   a1,12(sp)   # a1 = avg2(x,y)
24   call avg2        # a0 = avg2(a0,a1) = avg2(avg2(z,t), avg2(x,y))
25   lw   ra,0(sp)    # restore ra from stack frame
26   addi sp,sp,16    # restore sp
27   ret              # return
```

Listing 2: The fixed `avg4` function

But of course, not using `avg2` at all, contrary to the specifications, would be much simpler as shown in Listing 3.

```
avg4_simple:
  add  a0,a0,a1
  add  a0,a2,a3
  srli a0,2
  ret
```

Listing 3: The simple `avg4` function

Accuracy

Under what condition on the four inputs of `avg4` is the output result the **exact** average (no rounding)?

The output result of `avg4` is the **exact** average of x, y, z, t if and only if $x+y$ and $z+t$ are even and $x+y+z+t$ is a multiple of 4 ($(x+y) \bmod 2 = (z+t) \bmod 2 = (x+y+z+t) \bmod 4 = 0$).

We denote:

- P: the output result of `avg4` is the **exact** average of x, y, z, t
- Q_1: $(x+y) \bmod 2 = 0$
- Q_2: $(z+t) \bmod 2 = 0$
- Q_3: $(x+y+z+t) \bmod 4 = 0$
- Q: $Q_1 \land Q_2 \land Q_3$

$Q \Rightarrow P$ is immediate. For the other direction we first remark that, as the rounding is always toward zero, one rounding error cannot compensate for another. So: $P \Rightarrow \lfloor \frac{x+y}{2} \rfloor = \frac{x+y}{2} \Rightarrow Q_1$. Symmetrically, $P \Rightarrow \lfloor \frac{z+t}{2} \rfloor = \frac{z+t}{2} \Rightarrow Q_2$. Finally,

$$P \Rightarrow \left\lfloor \frac{\lfloor \frac{x+y}{2} \rfloor + \lfloor \frac{z+t}{2} \rfloor}{2} \right\rfloor = \frac{\lfloor \frac{x+y}{2} \rfloor + \lfloor \frac{z+t}{2} \rfloor}{2} \Rightarrow \left(\left\lfloor \frac{x+y}{2} \right\rfloor + \left\lfloor \frac{z+t}{2} \right\rfloor \right) \bmod 2 = 0$$

$$\Rightarrow \left(\frac{x+y}{2} + \frac{z+t}{2} \right) \bmod 2 = 0 \Rightarrow \left(\frac{x+y+z+t}{2} \right) \bmod 2 = 0 \Rightarrow Q_3$$

Combining the 3 implications:

$$P \Rightarrow Q_1 \land Q_2 \land Q_3 = Q \blacksquare$$

Avoiding overflows

If you were asked to code function `avg2` how would you avoid overflows? Explanations in natural language are enough, you do not have to provide RV32I code, but you can code if you wish or if it helps explaining.

Avoiding overflows in `avg2` is easy: we use the bitwise AND to save the modulo 2 of the two inputs in temporary registers (Lines 5 - 6 of Listing 4), then we use the logical right shift to divide the two inputs by 2 (Lines 7 - 8), we add them together (Line 9), add 1 if the two saved modulo are 1s (Lines 10 - 11), and we return (Line 12). There is no overflow because after dividing the two inputs by 2 their range is $[0...2^{30}-1]$, so the range of their sum is $[0...2^{31}-2]$. Even if they are odd the final addition of 1 does not cause an overflow, the range of the result is $[0...2^{31}-1]$.

Listing 4 shows the source code of `avg2`.

```
1  .text
2  # inputs x,y in a0,a1 respectively
3  # output in a0
4  avg2:
5    andi  t0,a0,1     # t0 = a0 modulo 2
6    andi  t1,a1,1     # t1 = a1 modulo 2
7    srli  a0,a0,1     # a0 = a0 / 2
8    srli  a1,a1,1     # a1 = a1 / 2
9    add   a0,a0,a1    # a0 = a0 + a1
10   and   t0,t0,t1    # t0 = t0 and t1
11   add   a0,a0,t0    # a0 = a0 + t0
12   ret               # return
```

Listing 4: The `avg2` function

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

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

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 - a7 - Function arguments
a0 - a1 - Return value(s)