Computer Architecture exam (2 hours) (1 hour)

Renaud Pacalet - 2026-02-04

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 6 questions are worth 4 points each. Yes, that's a total of 24, but 20 is the maximum you can get.

1. Floating point numbers

Computer systems approximately represent real numbers with what is called floating point numbers. The IEEE standard 754-2019 specifies the format of a 32 bits floating point number and the value of the real number it represents:

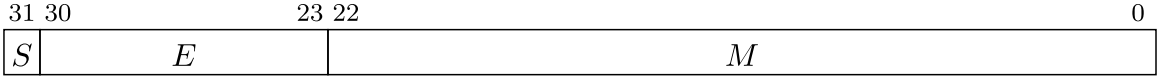

Figure 1: IEEE 754-2019 floating point numbers (32 bits)

- The leftmost bit (bit number 31) is the sign bit $S \in \{0, 1\}$.
- Bits 30 down to 23 encode an 8 bits unsigned integer $E \in \{0, ..., 255\}$.
- Bits 22 down to 0 encode a 23 bits unsigned integer $M \in \{0, ..., 2^{23} - 1\}$.

We denote $\mathrm{v}(S, E, M)$ the value of the corresponding real number. The standard distinguishes several cases:

- $\mathrm{v}(S, 0 < E < 255, M) = (-1)^S \times 2^{E-127} \times (1 + M \times 2^{-23})$: *normal* numbers
- $\mathrm{v}(S, 0, M \neq 0) = (-1)^S \times 2^{-126} \times (M \times 2^{-23})$: *subnormal* (very small) numbers
- $\mathrm{v}(S, 0, 0) = (-1)^S \times 0$ (± 0)
- $\mathrm{v}(S, 255, 0) = (-1)^S \times \infty$ $(\pm\infty)$
- $\mathrm{v}(S, 255, M \neq 0) = \mathrm{NaN}$ (Not-a-Number)

1. Let `0xC1FC0000` be the hexadecimal representation of a 32-bits floating point number. What real value does it represent?

2. Of course, the floating point representation of real numbers is not perfect and most real numbers can be represented only approximately. Let A be the floating point number nearest to $\frac{1}{3}$. Write the binary and hexadecimal forms of A.

2. Branch prediction

Consider the Variant of the Saturating Counter (VSC) branch predictor which diagram is represented on Figure 2 where `ST`, `WT`, `WN` and `SN` represent *Strong Taken*, *Weak Taken*, *Weak Not taken* and *Strong Not taken*, respectively. The transition labels `t` and `n` represent the actual branch outcomes *taken* and *not taken*, respectively. The VSC branch predictor predicts *taken* when it is in states `ST` or `WT` and *not taken* when it is in states `SN` or `WN`.

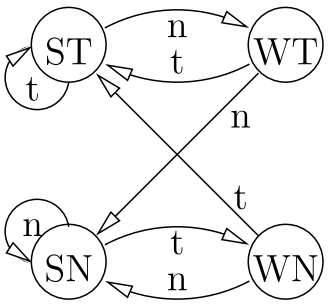

Figure 2: Variant of saturating counter

A RISC-V processor is equipped with a VSC branch predictor and executes the assembly code shown on Listing 1, where N is some strictly positive integer:

```
1  repeat:
2    addi t0,zero,0          # t0 <- 0
3    addi t1,zero,N          # t1 <- N
4  loop:
5    addi t0,t0,1            # t0 <- t0 + 1
6    bne  t0,t1,loop         # if t0 != t1 goto loop
7    beq  zero,zero,repeat   # goto repeat
```

Listing 1: Nested loops

1. Calculate the Total Number of Mispredictions (TNM) and the Mispredictions Per Branch Instruction ratio (MPBI) of the VSC branch predictor for the first branch instruction (Line 6). Reminder: the MPBI is the ratio TNM / total number of executions of the branch instruction. Note that the outermost loop being an infinite loop this branch instruction is executed an infinite number of times, so TNM can also be infinite, while MPBI is in $[0, 1]$. Warning: these numbers may depend on the value of N and/or on the initial state of the predictor; consider all cases.

2. What advantage(s) of this variant can you imagine, compared to the classic saturating counter shown on Figure 3?

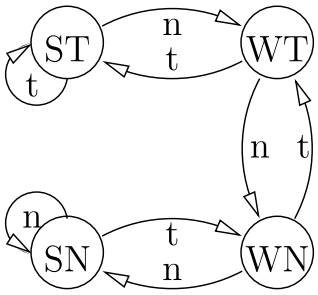

Figure 3: Two bits saturating counter

3. Computer hardware architectures

1. Give at least two examples of techniques that can be used to improve the performance of a processor.
2. Assuming performance is not an issue, propose two ways to reduce the size of a hardware implementation of the RISC-V 32 bits integer base ISA (RV32I, without multiplication/division extension).
3. A *one clock cycle per instruction* hardware implementation of an ISA is usually less efficient than a pipelined one. Where does it come from?
4. Compare a *one clock cycle per instruction* hardware implementation of an ISA and the *micro-coded, multiple clock cycles per instruction* hardware implementation of the same

ISA. In your opinion, which one exhibits the best performance? Why? What about their respective sizes?

4. Hardware support for Operating Systems

1. In order to run a full-featured Operating System like GNU/Linux or Microsoft Windows a hardware timer is mandatory. Propose an example of another type of hardware support which is mandatory to run a full-featured Operating System like GNU/Linux or Microsoft Windows.

   Explain.

2. Briefly explain what a Memory Management Unit (MMU) is and what it does.

3. Give examples of services that a MMU provides and that are impossible or far less efficient in MMU-less computer systems.

4. Is a MMU absolutely mandatory to run a full-featured Operating System like GNU/Linux or Microsoft Windows or is it there for performance reasons only? Explain.

5. RISC-V 5-stages pipeline

In a 5-stages RISC-V pipeline as seen during the course we run the RV32I assembly program shown on Listing 2.

```
1    xor x2,x2,x2     # x2 <- x2 xor x2
2    add x3,x4,x2     # x3 <- x4 + x2
3    bne x2,x3,diff   # if x2 != x3 goto diff
4    beq x0,x0,eq     # goto eq
5  diff:
6    lw x7,0(x6)      # x7 <- mem[0 + x6]
7    sw x6,0(x5)      # mem[0 + x5] <- x6
8  eq:
9    lw x8,0(x5)      # x8 <- mem[0 + x5]
```

Listing 2: RV32I assembly program

1. Identify the various pipeline hazards.
2. For each hazard:
   - In which class of hazards does it fall?
   - Which technique(s) is/are the best to deal with it?

6. RISC-V assembly coding

Function `pop1` takes a memory address in register `a0`, loads the byte at this address from memory, counts the number of bits equal to 1 in the byte (its *Hamming weight*), and returns this count in register `a0`. Example: Table 1 represents a small portion of the memory, with addresses and byte values in hexadecimal.

Address	Value
...	...
0x10010003	0x00
0x10010002	0x03
0x10010001	0x0f
0x10010000	0xff
...	...

Table 1: The state of a memory area

If function `pop1` is called with value `0x10010000` in register `a0`, it returns value 8 in register `a0` because the memory contains byte `0xff` at address `0x10010000`, and `0xff` is the same as `11111111` in binary, that is, 8 bits equal to 1. If it is called with value `0x10010002` in register `a0`, it returns value 2 in register `a0`.

A software engineer has been asked to code another function, named `pop`, that takes a starting address in `a0`, an ending address in `a1`, calls `pop1` on all addresses between - and including - the two, accumulates the returned byte Hamming weights, and returns the sum in `a0`. Example: with the same memory state shown on Table 1, if `pop` is called with `0x10010001` in `a0` and `0x10010003` in `a1`, it returns $4 + 2 + 0 = 6$ in `a0`.

The engineer has been told that the ending address is always larger or equal the starting address (no need to check), and that they should ignore overflows. They have also been told that their code shall be compliant with the ILP32 Application Binary Interface (ABI) that we studied during the lectures and used in the labs.

The engineer wrote the code shown on Listing 3.

```
1  pop:
2    addi  sp,sp,-16       # allocate stack frame
3    sw    ra,0(sp)        # save ra in stack frame
4    li    a2,0            # initialize total
5    mv    a3,a0           # copy starting address in a3
6  pop_loop:
7    call  pop1            # a0 = Hamming weight of current byte
8    add   a2,a2,a0        # update total
9    addi  a3,a3,1         # a3 = address of next byte
10   mv    a0,a3           # copy address of next byte in a0
11   bgeu  a1,a0,pop_loop  # if ending address >= address of next byte, continue
12 pop_end:
13   mv    a0,a2           # copy total in a0
14   lw    ra,0(sp)        # restore ra from stack frame
15   addi  sp,sp,16        # restore sp
16   ret                   # return
```

Listing 3: The RISC-V assembly code of function pop

Unfortunately their code is **not** compliant with the ILP32 Application Binary Interface (ABI). Explain why and write an ABI-compliant code for function `pop`.

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)