Computer Architecture exam

Renaud Pacalet - 2025-01-31

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 2 first questions are worth 4 points each. The 2 last questions are worth 6 points each.

1. Hardware timers

1. Explain what a hardware timer is.
2. In order to run a full-featured Operating System (OS) like GNU/Linux, a hardware timer is mandatory. Why? Provide examples of services that an OS cannot offer without a hardware timer.
3. Give at least two other examples of hardware supports without which a hardware timer alone cannot be fully exploited by the OS. Explain.

2. 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:

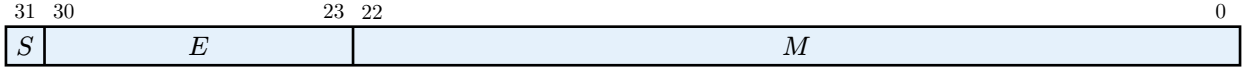

Figure 1: IEEE 754-2019 floating point numbers (32 bits)

- 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 $v(S, E, M)$ the value of the corresponding real number. The standard distinguishes several cases:

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

1. Let `0xB0400000` be the hexadecimal representation of a 32-bits floating point number. What real value does it represent?
2. The addition of floating point numbers is not associative: in some cases $(X + Y) + Z \neq X + (Y + Z)$. Give an example of two 32-bits *normal* ($0 < E < 255$) floating point numbers A and B, in binary form or in hexadecimal form, as you wish, such that $(A + B) - B \neq A$. Explain.
3. Suppose you are a digital hardware designer and you are asked to design a hardware floating point unit using the logic gates and D-flip-flops we saw during the lectures. In your opinion, what operation will be the more complex to design: addition or multiplication? Why?

3. RISC-V assembly coding

Listing 1 shows the source code of function `qux` in RV32I assembly language.

```
1  qux:
2    addi  sp,sp,-16
3    sw    ra,0(sp)
4    sw    s0,4(sp)
5    bne   a1,zero,L1
6    addi  a1,zero,1
7    beq   zero,zero,L3
8  L1:
9    lw    s0,0(a0)
10   addi  a1,a1,-1
11   beq   a1,zero,L2
12   addi  a0,a0,4
13   jal   ra,qux
14   blt   a0,s0,L3
15 L2:
16   addi  a0,s0,0
17 L3:
18   lw    s0,4(sp)
19   lw    ra,0(sp)
20   addi  sp,sp,16
21   jalr  zero,0(ra)
```

Listing 1: The `qux` function

1. Explain what function `qux` does.
2. What input parameters does it take and what is their role?
3. What output results does it return and what are they?
4. Is `qux` compliant with the ILP32 Application Binary Interface (ABI)? Explain. If not fix it and provide the complete code of a compliant version.
5. We assume that each instruction takes exactly one clock cycle to execute (no pipeline, no hazards). For a given combination of the input parameters what is the Worst Case Execution Time (WCET) of the ABI-compliant `qux`? Provide your answer in the form of one or more equations with the input parameters as variables.
6. Could `qux` be optimized to reduce the WCET?

   If not explain why, else propose a complete optimized version and provide the new equation(s) of the WCET.

4. Caches

A data cache contains copies of data from the external memory, that we name *net cache data* in the following, plus some management information (tags, flags, replacement policy information, cache coherence information, …)

We consider a computer system with byte addresses on 32 bits. The basic addressing unit is a 32 bits (4 bytes) word. A data cache is used to improve performance: 2-ways set-associative, write-back, write-allocate, four 32-bits words per line, Least Recently Used (LRU) replacement policy. The total size of its net cache data is 8 kB (8192 bytes).

Starting from an empty cache (all lines invalid), the cache receives a sequence of word read accesses from its processor at the following addresses (in hexadecimal):

```
3A06C1B4, CD7351BC, 556FE81C, 3A06C1B8,
556FE1BC, 3A06C814, CD735818, 556FE1BC,
3A06C818, CD735D7C, 556FE1BC, 3A06C1B4,
CD735D70, CD7351B0, 3A06C814, 556FE1B0
```

The first access is at `3A06C1B4`, the second at `CD7351BC` the last at `556FE1B0`.

Simulate the cache for this sequence of read accesses and write a 16 lines table representing the behavior of the cache. The format must be as shown on Table 1:

Address	**Operation**
`01234567`	`E`
…	…

Table 1: Example of cache behavior table

where:

- Address is the **hexadecimal** address that was read.

- Operation is `H` (hit), `M` (miss without eviction) or `E` (miss with eviction).

Example: `12345678 E` means that the read address was `12345678`, it was a miss and some valid net cache data has been evicted (replaced with some other data). If you wish, and if it helps, you can add other columns on the right to represent other information. If you do so, add the corresponding header cells to name each extra column and add some text to explain what these extra columns represent.

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)