Computer Architecture: exam

R. Pacalet

2024-02-07

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 RISC-V assembly coding (6 points)

In this exercise we use the RV32I Instruction Set Architecture (ISA) **without** the multiplication and division extension. We code according the ILP32 Application Binary Interface (ABI) seen during the lectures and the labs, **without** any exception: **any** function that we write can be called from any piece of software about which we only know that it 100% respects the same ABI. If needed use the provided RISC-V reference card on assembly language syntax.

In order to check if an unsigned number `N` is a multiple of 7, without divisions, we can use the following algorithm:

```
1. If N equals 0 or 7 answer "yes" and stop,
2. if N is less or equal 13 answer "no" and stop,
3. if N is even divide it by 2,
4. else (if N is odd) divide it by 2 and add 4,
5. goto step 1.
```

Note: in step 4 the division is an integer division (`17/2 = 8`); remember that you **cannot** use the multiplication and division instructions.

1. In RV32I assembly language code a function `step` that takes a 32-bits **unsigned number** in register `a0`, implements steps 3 and 4 of the algorithm, and returns the modified number in register `a0`. Examples: `step(5) = 6`, `step(28) = 14`, `step(127) = 67`.
2. In RV32I assembly language code a function `is_multiple_of_7` that implements the complete algorithm and that uses function `step` for steps 3 and 4. `is_multiple_of_7` takes a 32-bits **unsigned number** in register

`a0` and returns 1 in register `a0` if the number is a multiple of 7, else 0. Examples: `is_multiple_of_7(0) = 1`, `is_multiple_of_7(5) = 0`, `is_multiple_of_7(28) = 1`, `is_multiple_of_7(127) = 0`.

3. Assuming each instruction takes exactly one clock cycle to execute what is the Best Case Execution Time (BCET) of your `is_multiple_of_7` function? For what input value?
4. Assuming each instruction takes exactly one clock cycle to execute what is the Worst Case Execution Time (WCET) of your `is_multiple_of_7` function? For what input value?

2 Branch prediction (6 points)

Definitions and notations

- **Miss-Prediction per executed Branch Instruction (MPBI)**: the number of times a given branch instruction has been wrongly predicted divided by the total number of times this same branch instruction has been executed. The lower the MPBI, the better the prediction.
- $\mathbf{M}_\infty$: for a given branch instruction, the limit of the MPBI when the number of times the branch instruction is executed tends to infinity, if this limit exists. Undefined if it does not exist.
- **Branch outcome**: the actual decision (**not the prediction**) for a given branch instruction; Taken or Not taken, denoted `T` and `N`, respectively.
- **Periodic infinite sequence of outcomes**: a sequence of outcomes that starts with a finite sequence, the stem, which can be empty, and continues with a finite cycle that repeats infinitely. We represent these sequences as `STEM(CYCLE)*` where `STEM` is the shortest possible stem and `CYCLE` is the shortest possible cycle. Example: `TNTN NNNTTTT NNNTTTT NNNTTTT`... is a periodic infinite sequence of outcomes, its shortest possible stem is `TNTN`, its shortest possible cycle is `NNNTTTT` and we represent it as `TNTN(NNNTTTT)*`.

Questions

A branch instruction is predicted using the 2-bits saturating counter (4-states) branch predictor studied during the lecture on pipelines and represented on Figure 1. We assume that the predictor is initialized in the **Strong Taken (ST)** state and that there is no collision with other branch instructions: the predictor is only used to predict the branch instruction of interest.

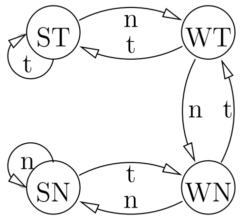

Figure 1: Saturating counter

1. Imagine an infinite sequence of outcomes of the branch instruction such that the 2-bits saturating counter has a MPBI equal to 1 (that is, it always predicts wrongly). Represent your sequence using the `STEM(CYCLE)*` notation.

2. Imagine a RV32IM assembly code snippet with a branch instruction that would produce such a sequence. Label `B1` the branch instruction of interest.

Instead of the 2-bits saturating counter we decide to use the variant predictor, also studied during the lecture on pipelines and represented on Figure 2.

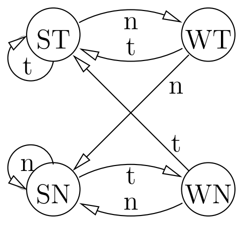

Figure 2: Variant

We assume that the predictor is initialized in the **Weak Not taken (WN)** state and that there is no collision with other branch instructions: the predictor is only used to predict the branch instruction of interest.

3. Imagine an infinite sequence of outcomes of a branch instruction such that the variant predictor has a MPBI equal to 1. Represent your sequence using the `STEM(CYCLE)*` notation.

4. Imagine a RV32IM assembly code snippet that would produce such a sequence. Label `B2` the branch instruction of interest.

In order to avoid these undesirable situations we decide to use a two-levels prediction strategy with 2-bits **local** Branch History Shift Registers (BHSR). We still assume that there are no collisions.

5. What would be the $\mathbf{M}_\infty$ for branch instruction `B1` in your first code snippet, with 2-bits saturating counters?

6. What would be the $\mathbf{M}_\infty$ for branch instruction `B2` in your second code snippet, with variant predictors?

3 Binary representation of data (4 points)

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 $a_{n-1}a_{n-2}\ldots a_1a_0$ the n-bits representation of integer A. In *sign and magnitude* a_{n-1} is the *sign* bit. In *two's complement* a_{n-1} is the *Most Significant Bit* (MSB). In both representations 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?

4 Instruction Set Architecture (4 points)

The RV32IM Instruction Set Architecture (ISA) is the one we studied during the lectures and the labs. RV32IM means **R**ISC-**V** **32** bits **I**nteger with **M**ultiplication and division extension.

1. In the RV32IM ISA there are 32 General Purpose Registers (GPRs). Could we rework this ISA to add more GPRs? What would be the limitations?
2. What is an addressing mode?
3. What addressing modes are supported by the RV32IM ISA?
4. Give an example of an addressing mode that is not supported by the RV32IM ISA.

5 RISC-V 5-stages pipeline (4 points)

In the 5-stages RISC-V pipeline represented on Figure 3 we assume that:

- There are no structural pipeline hazards.
- Nothing has been done to solve the other types of pipeline hazards.

We run the assembly program shown on Listing 1.

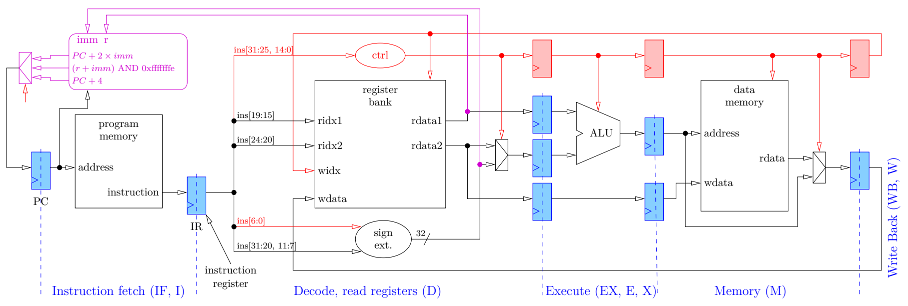

Figure 3: A 5-stages RISC-V pipeline

```
        sw    t0,0(t1)       # mem[0+t1] <- t0
        add   t0,t0,t2       # t0 <- t0+t2
        lw    t3,0(t0)       # t3 <- mem[0+t0]
        beq   s0,t0,label    # if s0==t0 goto label
        lw    s1,0(s1)       # s1 <- mem[0+s1]
        andi  t0,t0,0xff     # t0 <- t0 AND 0xff
label:
        ...
```

Listing 1: Assembly program

1. Identify the various pipeline hazards that may lead to unexpected behavior.
2. For each pipeline hazard:
   - In which class of pipeline hazards does it fall?
   - What technique is the best to deal with it?

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)