Ollscoil Chathair
Bhaile Átha Cliath
Dublin City University

SEMESTER 1 EXAMINATIONS 2023/2024

MODULE: EE496 - Computer Architecture and HDL

PROGRAMME(S):

ECE	BEng Electronic & Computer Engineering
ECSAO	Study Abroad (Engineering & Computing)
ECEI	BEng Electronic & Computer Engineering

YEAR OF STUDY: 4,O

EXAMINER(S):

Assoc. Prof. Xiaojun Wang	(Internal)	(Ext:5808)
Prof. Martin Glavin	(External)	External

TIME ALLOWED: 2 Hours

INSTRUCTIONS: **Answer 4 questions. All questions carry equal marks.**

You may need to refer to the RISC-V instructions, pseudo instructions, and the register set in the Appendices of the exam paper.

Appendix 1: RV32I RISC-V integer instructions
Appendix 2: RISC-V pseudo instructions
Appendix 3: RISC-V register set

PLEASE DO NOT TURN OVER THIS PAGE UNTIL YOU ARE INSTRUCTED TO DO SO.
The use of programmable or text storing calculators is expressly forbidden.
Please note that where a candidate answers more than the required number of questions, the examiner will mark all questions attempted and then select the highest scoring ones.

There are no additional requirements for this paper.

QUESTION 1 ***[TOTAL MARKS: 25]***

Q 1(a) **[10 Marks]**

Write the RISC-V assembly code that corresponds to the following C code snippet.

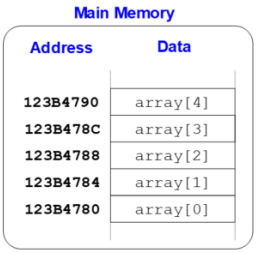
```
// C Code add the numbers from 1 to 99
int sum = 0;
int i;
for (i=1; i<100; i = i+1) {
  sum = sum + i;
}
```

Q 1(b) **[8 Marks]**

If a five-element array is stored in the main memory, as shown below. Convert the following C code fragment into equivalent RISC-V assembly code.

```
// C Code
      int array[5];
      array[2] = array[2] * 2;
      array[3] = array[3] * 4;

# RISC-V assembly code
# s0 = array base address
```



Q 1(c) **[7 Marks]**

The RISC-V assembly code corresponding to the C code is stored in the main memory from the memory address `0x00000300`, as shown below.

```
C Code                  RISC-V assembly code
int main() {
  simple();             0x00000300 main:   jal  ra, simple  # call
  a = b + c;            0x00000304         add  s0, s1, s2
}                       ...                ...

void simple() {
  return;               0x0000051c simple: jr   ra          # return
}
```

i) After execution of the `jal simple` instruction, what is the value in the program count register `PC`, and what is in the return address register `ra`?

ii) After execution of the `jr ra` instruction, what is the value in the program counter register `PC`?

[End of Question 1]

QUESTION 2 ***[TOTAL MARKS: 25]***

Q 2(a) **[5 Marks]**

The `jal` (jump and link) instruction in RSIC-V has the following format:

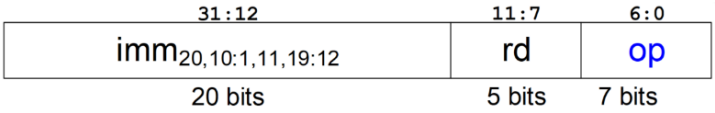
```
jal rd, offset
```

In which the `rd` stores the return register, the `offset` represents the 21-bit offset of the target instruction's memory address relative to the current instruction's memory address.

i) How many instructions can a `jal` instruction jump forwards (i.e., to higher addresses)?
ii) How many instructions can a `jal` instruction jump backwards (i.e., to lower addresses)?

Q 2(b) **[15 Marks]**

The J-type instruction in RISC-V has the following encoding format. Only the most significant 20 bits (20:1) of the 21-bit (20:0) signed immediate value are encoded in the instruction; the least significant bit (bit 0) is not encoded because it is always 0.



Convert the following jump instructions into machine code. Instruction addresses are given to the left of each instruction. The J-Type instruction's 7-bit opcode (i.e., the **op**) is **1101111**.

```
i)    0x1234ABC0       j L1      #jump forward to Label L1
      ...              ...
      0x0000EEEC  L1:  ...

ii)   0x0000C10C  L2:  ...
      ...              ...
      0x000F1230       jal L2    #jump backwards to Label L2
```

Q 2(c) **[5 Marks]**

Write a RISC-V assembly language program for swapping the contents of two registers, a0 and a1, without using any other temporary register. Comment on your code.

[End of Question 2]

QUESTION 3 ***[TOTAL MARKS: 25]***

Q 3(a) **[5 Marks]**

Briefly explain the advantages of pipelined microprocessors.

Q 3(b) **[8 Marks]**

Each `addi`, `bge`, and `jal` instruction takes 4, 3 and 4 cycles in a multicycle RISC-V microarchitecture. How many cycles are required to run the following program on this multicycle RISC-V processor? What is the average CPI (Cycles Per Instruction) of this program?

```
     addi s0, zero, 5     # result = 5
L1:
     bge zero, s0, Done   # if result <= 0, exit loop
     addi s0, s0, -1      # result = result - 1
     j L1
Done:
```

Q 3(c) **[6 Marks]**

RISC-V does not have a native `nor` instruction. However, the `nor` functionality can be implemented using RISC-V instructions. Write a short assembly code snippet to implement: `s3 = s4 nor s5`. Use as few instructions as possible.

Q 3(d) **[6 Marks]**

Draw the logic circuit implied by the SystemVeriog module and explain the correspondence between each logic component and the source code.

```
module q3d(input  logic clk, en, d,
                  output logic q);
  logic n1;
  always_ff @(posedge clk)
     begin
       if (en) begin
        n1 <= d;
        q  <= n1;
       end
     end
endmodule
```

[End of Question 3]

QUESTION 4 ***[TOTAL MARKS: 25]***

Q 4(a) **[10 Marks]**

Write a SystemVerilog model that infers a simple single port 16x32 RAM block with 16 32-bit words. The RAM write is synchronous to a clock rising edge, and when RAM write is enabled (we). The RAM read is asynchronous. The figure below shows the RAM input and output ports.

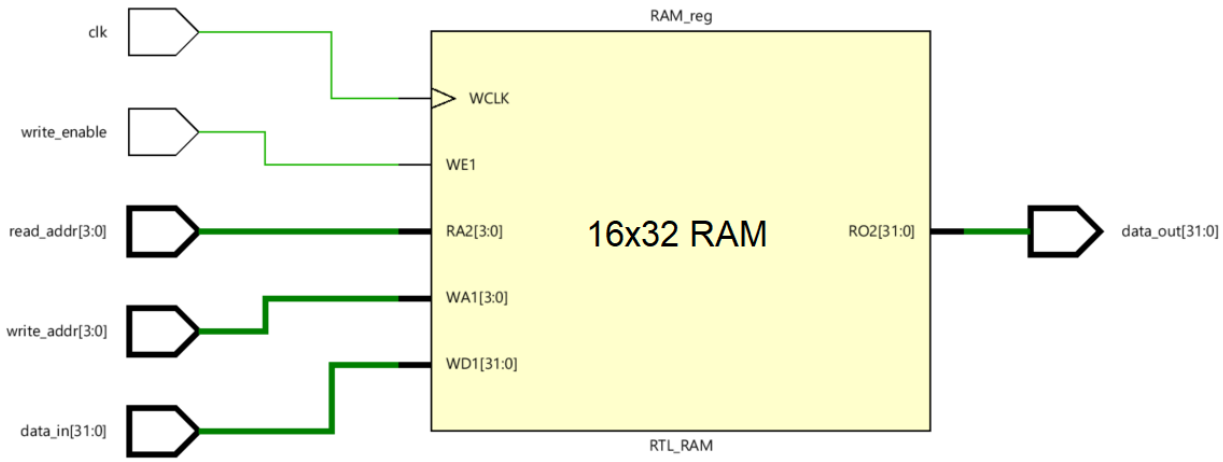


Figure Q 4(a)

Q 4(b) **[7 Marks]**

Figure Q4(c) shows a 32-bit bidirectional IO buffer of a device connected to a bidirectional *data_bus*. The device can write its output data (*data_out*) to the *data_bus* when its output is enabled (i.e. OE = '1') and can read data (data_in) from the *data_bus* when its output is disabled (OE = '0', i.e. $\overline{OE}$ = '1'). Write a SystemVerilog model that infers this bidirectional IO buffer.

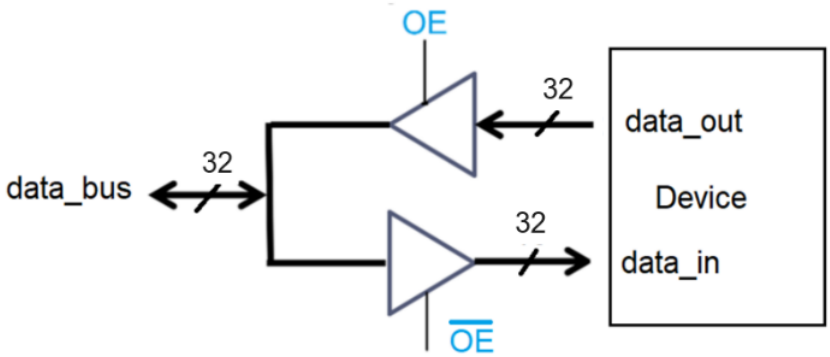


Figure Q 4(c) Bidirectional IO buffer

Question 4 continues on the next page.

Q 4(c) **[8 Marks]**

The truth table of a bus arbiter is shown below, in which **Brq0** to **Brq3** are the bus request signals from devices 0 to 3; **Bgt0** to **Bgt3** are the bus grant signals to devices 0 to 3.

i) write a SystemVerilog model for the bus arbiter and

ii) Draw the logic circuit implied by your SystemVerilog model.

Brq_3	Brq_2	Brq_1	Brq_0	Bgt_3	Bgt_2	Bgt_1	Bgt_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	**1**
0	0	1	X	0	0	**1**	0
0	1	X	X	0	**1**	0	0
1	X	X	X	**1**	0	0	0

[End of Question 4]

QUESTION 5 ***[TOTAL MARKS: 25]***

Q 5(a) **[4 Marks]**

Assuming a multi-cycle RISC-V datapath has five pipeline stages:

1. Instruction *fetch*
2. Instruction *decode*
3. Instruction *execution* (i.e. the ALU)
4. Access *memory* for read or write
5. ALU result is *write-back* to the register-file

Assume that each stage takes the time specified in the table below.

Fetch Instruction from Memory	**Decode** & read operands from reg files	Instruction **Execution** ALU	**Memory** Read/Write	Results **write-back** to Reg files
200 ps	160 ps	120 ps	200 ps	100 ps

Given the times for the *datapath* stages listed above, what would the maximum *clock frequency* be?

Q 5(b) **[4 Marks]**

For the pipelined *datapath* in Q5(a), assuming no hazards or stalls, how long does it take to execute *one* instruction, or what is the instruction *latency*?

Q 5(c) **[4 Marks]**

What is the *throughput* of the pipelined datapath in Q5(a)?

Q 5(d) **[8 Marks]**

Give one example of a data hazard in a pipelined processor and briefly explain how it can be resolved.

Q 5(e) **[5 Marks]**

Briefly describe the *memory hierarchy* of a modern computer system.

[End of Question 5]

Appendix 1: RV32I RISC-V integer instructions

Table B.1 RV32I: RISC-V integer instructions

Instruction	Description	Operation
lb rd, imm(rs1)	load byte	rd = SignExt([Address]$_{7:0}$)
lh rd, imm(rs1)	load half	rd = SignExt([Address]$_{15:0}$)
lw rd, imm(rs1)	load word	rd = [Address]$_{31:0}$
lbu rd, imm(rs1)	load byte unsigned	rd = ZeroExt([Address]$_{7:0}$)
lhu rd, imm(rs1)	load half unsigned	rd = ZeroExt([Address]$_{15:0}$)
addi rd, rs1, imm	add immediate	rd = rs1 + SignExt(imm)
slli rd, rs1, uimm	shift left logical immediate	rd = rs1 << uimm
slti rd, rs1, imm	set less than immediate	rd = (rs1 < SignExt(imm))
sltiu rd, rs1, imm	set less than imm. unsigned	rd = (rs1 < SignExt(imm))
xori rd, rs1, imm	xor immediate	rd = rs1 ^ SignExt(imm)
srli rd, rs1, uimm	shift right logical immediate	rd = rs1 >> uimm
srai rd, rs1, uimm	shift right arithmetic imm.	rd = rs1 >>> uimm
ori rd, rs1, imm	or immediate	rd = rs1 | SignExt(imm)
andi rd, rs1, imm	and immediate	rd = rs1 & SignExt(imm)
auipc rd, upimm	add upper immediate to PC	rd = {upimm, 12'b0} + PC
sb rs2, imm(rs1)	store byte	[Address]$_{7:0}$ = rs2$_{7:0}$
sh rs2, imm(rs1)	store half	[Address]$_{15:0}$ = rs2$_{15:0}$
sw rs2, imm(rs1)	store word	[Address]$_{31:0}$ = rs2
add rd, rs1, rs2	add	rd = rs1 + rs2
sub rd, rs1, rs2	sub	rd = rs1 − rs2
sll rd, rs1, rs2	shift left logical	rd = rs1 << rs2$_{4:0}$
slt rd, rs1, rs2	set less than	rd = (rs1 < rs2)
sltu rd, rs1, rs2	set less than unsigned	rd = (rs1 < rs2)
xor rd, rs1, rs2	xor	rd = rs1 ^ rs2
srl rd, rs1, rs2	shift right logical	rd = rs1 >> rs2$_{4:0}$
sra rd, rs1, rs2	shift right arithmetic	rd = rs1 >>> rs2$_{4:0}$
or rd, rs1, rs2	or	rd = rs1 | rs2
and rd, rs1, rs2	and	rd = rs1 & rs2
lui rd, upimm	load upper immediate	rd = {upimm, 12'b0}
beq rs1, rs2, label	branch if =	if (rs1 == rs2) PC = BTA
bne rs1, rs2, label	branch if ≠	if (rs1 ≠ rs2) PC = BTA
blt rs1, rs2, label	branch if <	if (rs1 < rs2) PC = BTA
bge rs1, rs2, label	branch if ≥	if (rs1 ≥ rs2) PC = BTA
bltu rs1, rs2, label	branch if < unsigned	if (rs1 < rs2) PC = BTA
bgeu rs1, rs2, label	branch if ≥ unsigned	if (rs1 ≥ rs2) PC = BTA
jalr rd, rs1, imm	jump and link register	PC = rs1 + SignExt(imm), rd = PC + 4
jal rd, label	jump and link	PC = JTA, rd = PC + 4

Appendix 2: RISC-V pseudo instructions

Pseudo instruction	RISC-V Instructions	Description	Operation
nop	addi x0, x0, 0	no operation	
li rd, $imm_{11:0}$	addi rd, x0, $imm_{11:0}$	load 12-bit immediate	rd = SignExtend($imm_{11:0}$)
li rd, $imm_{31:0}$	lui rd, $imm_{31:12}$ * addi rd, rd, $imm_{11:0}$	load 32-bit immediate	rd = $imm_{31:0}$
mv rd, rs1	addi rd, rs1, 0	move (also called "register copy")	rd = rs1
not rd, rs1	xori rd, rs1, -1	one's complement	rd = ~rs1
neg rd, rs1	sub rd, x0, rs1	two's complement	rd = -rs1
seqz rd, rs1	sltiu rd, rs1, 1	set if = 0	rd = (rs1 == 0)
snez rd, rs1	sltu rd, x0, rs1	set if ≠ 0	rd = (rs1 ≠ 0)
sltz rd, rs1	slt rd, rs1, x0	set if < 0	rd = (rs1 < 0)
sgtz rd, rs1	slt rd, x0, rs1	set if > 0	rd = (rs1 > 0)
beqz rs1, label	beq rs1, x0, label	branch if = 0	if (rs1 == 0) PC = label
bnez rs1, label	bne rs1, x0, label	branch if ≠ 0	if (rs1 ≠ 0) PC = label
blez rs1, label	bge x0, rs1, label	branch if ≤ 0	if (rs1 ≤ 0) PC = label
bgez rs1, label	bge rs1, x0, label	branch if ≥ 0	if (rs1 ≥ 0) PC = label
bltz rs1, label	blt rs1, x0, label	branch if < 0	if (rs1 < 0) PC = label
bgtz rs1, label	blt x0, rs1, label	branch if > 0	if (rs1 > 0) PC = label
ble rs1, rs2, label	bge rs2, rs1, label	branch if ≤	if (rs1 ≤ rs2) PC = label
bgt rs1, rs2, label	blt rs2, rs1, label	branch if >	if (rs1 > rs2) PC = label
bleu rs1, rs2, label	bgeu rs2, rs1, label	branch if ≤ (unsigned)	if (rs1 ≤ rs2) PC = label
bgtu rs1, rs2, label	bltu rs2, rs1, offset	branch if > (unsigned)	if (rs1 > rs2) PC = label
j label	jal x0, label	jump	PC = label
jal label	jal ra, label	jump and link	PC = label, ra = PC + 4
jr rs1	jalr x0, rs1, 0	jump register	PC = rs1
jalr rs1	jalr ra, rs1, 0	jump and link register	PC = rs1, ra = PC + 4
ret	jalr x0, ra, 0	return from function	PC = ra
call label	jal ra, label	call nearby function	PC = label, ra = PC + 4
call label	auipc ra, offset31:12 * jalr ra, ra, offset11:0	call far away function	PC = PC + offset, ra = PC + 4
la rd, symbol	auipc rd, symbol31:12 * addi rd, rd, symbol11:0	load address of global variable	rd = PC + symbol
l{b|h|w} rd, symbol	auipc rd, symbol31:12 * l{b|h|w} rd, symbol11:0(rd)	load global variable	rd = [PC + symbol]
s{b|h|w} rs2, symbol, rs1	auipc rs1, symbol31:12 * s{b|h|w} rs2, symbol11:0(rs1)	store global variable	[PC + symbol] = rs2
csrr rd, csr	csrrs rd, csr, x0	read CSR	rd = csr
csrw csr, rs1	csrrw x0, csr, rs1	write CSR	csr = rs1

* If bit 11 of the immediate / offset / symbol is 1, the upper immediate is incremented by 1. symbol and offset are the 32-bit PC-relative addresses of a label and a global variable, respectively.

Appendix 3: RISC-V Register Set

RISC-V Register Set

Name	Register Number	Usage
zero	x0	Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0-2	x5-7	Temporaries
s0/fp	x8	Saved register / Frame pointer
s1	x9	Saved register
a0-1	x10-11	Function arguments / return values
a2-7	x12-17	Function arguments
s2-11	x18-27	Saved registers
t3-6	x28-31	Temporaries

[END OF EXAM]