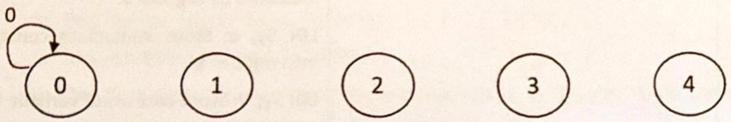
16/17

1. **[10 marks]**
A theme park offers locker rental to its visitors. To use a locker, a visitor deposits 4 tokens, one at a time, into the locker's token slot.

Design a sequential circuit with states *ABC* for the locker's door using a *D* flip-flop for *A*, a *T* flip-flop for *B*, and a *JK* flip-flop for *C*. The circuit consists of 5 states representing the number of tokens a visitor has deposited: 0, 1, 2, 3 and 4 (or, in binary, *ABC* = 000, 001, 010, 011 and 100). The visitor can deposit only one token at a time. When the circuit reaches the final state 4, it remains in state 4 even if the visitor continues to put tokens into the slot.

Let the external input *t* denotes a token.

a. Complete the given state diagram on the Answer Booklet. The state values are shown in decimal. The value on the arrow represents *t*. [2 marks]



b. Write the **simplified SOP expressions** for the flip-flop inputs. [8 marks]

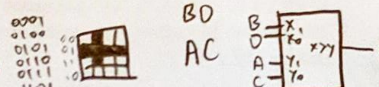
B>A or B=A and D>C

2. **[10 marks]**

a. Given the following Boolean function:

$$F(A,B,C,D) = \Sigma m(1, 4, 5, 6, 7, 13)$$

BD
AC



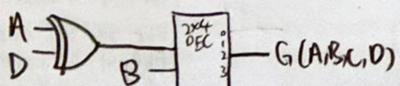
You are to implement *F* using a single **2-bit magnitude comparator** with <u>no additional logic gates</u>. Note that complemented literals are not available. [5 marks]

b. Given the following Boolean function:

$$G(A,B,C,D) = \Sigma m(2, 11)$$

0010
1011

You are to implement *G* using a single **2×4 decoder** with one-enable and active high outputs, and one 2-input exclusive-OR gate. Note that complemented literals are not available. [5 marks]



3. **[10 marks]**
Study the following MIPS program. Arrays *A* and *B* are integer arrays.

```
     # $s0 contains the starting address of array A
     # $s1 contains the starting address of array B
     add  $s2, $s0, $zero       #inst 1
     add  $s3, $s1, $zero       #inst 2
L1:  lw   $t0, 0($s2)           #inst 3
     lw   $t1, 0($s3)           #inst 4
     bne  $t0, $zero, L2        #inst 5
     beq  $t1, $zero, done      #inst 6
L2:  slt  $t2, $t0, $t1         #inst 7
     beq  $t2, $zero, L3        #inst 8
     sw   $t0, 0($s3)           #inst 9
     sw   $t1, 0($s2)           #inst 10
L3:  addi $s2, $s2, 4           #inst 11
     addi $s3, $s3, 4           #inst 12
     j    L1                    #inst 13
done:
```

0xE04800 08 0C 10 14 18 1C 20 24 28 2C 0xE0480030

a. Give the instruction encoding in hexadecimal for instruction 1 (**add $s2, $s0, $zero**). The opcode for **add** is 0 and the funct value for **add** is 0x20. [2 marks]

000000 10000 00000 10010 00000 100000 0x02009020

b. Give the instruction encoding in hexadecimal for instruction 6 (**beq $t1, $zero, done**). The opcode for **beq** is 0x04. [2 marks]

000100 01001 00000 0x1D 0x 11200020

c. If instruction 13 (**j L1**) is at memory address **0xE0480030**, give the instruction encoding in hexadecimal for instruction 13. The opcode for jump is 0x02. [2 marks]

000010 0000 0100 1000 0000 0000 0000 10
0x08120002

d. The following are the initial values of the array elements in arrays *A* and *B*.

Array *A*:	3	~~-1~~ 5	7	~~0~~ 3	~~2~~ 6	~~-5~~ 0	9	0	-9	1
Array *B*:	2	~~5~~ -1	7	~~3~~ 0	~~6~~ 2	~~0~~ 5	8	0	-3	2

Fill in the final values of the elements. [4 marks]

$t0 → ~~A[0]~~ ~~3~~ ~~-1~~ 0
$t1 → ~~2~~ ~~5~~ 3

B[0] = A[0]
A[0] = B[0]

bne $t0 == 0? N N
beq $t1 == 0? ~~N~~
$t2 → $t0 < $t1? N Y
beq $t2 == 0? ~~Y~~ N
~~N~~ swap

4. **[35 marks]**

Zephyr is a 32-bit stack-based processor with the specifications shown below. In this table, registers "x" and "y" serve as placeholders for actual general purpose registers \$1, \$2, ..., \$8, the capital letter "V" refers to a single variable, the capital letter "A" refers to the first element of an array, and the small letter "c" refers to a constant. The capital letters "X" and "Y" refer to the top-most and second elements of the stack respectively. All constants and displacements are 2's complement signed values.

0111
1001
$-2^3+1=-7$

Addressing Architecture:	Stack based.
Number of General Purpose Registers:	Eight (\$1, \$2, ..., \$8)
Special registers:	Stack pointer (\$sp) Program counter (\$pc)
Instruction formats:	Fixed length 32-bit instructions
Arithmetic instructions: Arithmetic instructions. X is the top-most operand on the stack, Y is the next operand in the stack.	**ADD**: X and Y are popped off the stack, X+Y are pushed back onto the stack. **SUB**: X and Y are popped off the stack, X-Y is pushed back onto the stack. **MUL**: X and Y are popped off the stack, X*Y is pushed back onto the stack. **DIV**: X and Y are popped off the stack. X/Y is pushed back onto the stack.
Stack instructions: Stack manipulation instructions. Special register \$sp points to the top-most element of the stack. Stacks are assumed to be arbitrarily large, while popping an empty stack will cause an error, but we WILL NOT consider that here. We will assume that the stack is never empty nor full.	**PUSHI c**: Push immediate value c onto the stack. **PUSH \$y**: Push register y onto the stack. **POP \$y**: Pop topmost item on the stack into register y. **ZERO**: Reset \$sp to bottom of stack.

4. (continued)

Load/Store Instructions: These are load and store instructions that get data from memory to registers and vice versa. All addresses are byte addresses.	**LW \$y, \$x**: Load 32-bit word stored in address pointed to by register x into register y. **SW \$y, \$x**: Store 32-bit word in register y, into the address pointed to by register x. **LB \$y, \$x**: Load a single byte stored in the address indicated by register x, into the lowest (bits 7-0) bits of register y. **SB \$y, \$x**: Store the lowest 8-bits (bits 7-0) of register y into the byte address indicated by register x. **LDI \$y, c**: Store immediate constant c into register y. **LDI \$y, V**: Store address of variable V into register y. **LDI \$y, A**: Store base address of array A into register y. **INCW \$y**: Register y is incremented by 4. **DECW \$y**: Register y is decremented by 4. **INC \$y**: Register y is incremented by 1. **DEC \$y**: Register y is decremented by 1.
B-type instructions: These are compare and branch instructions.	**BEQ \$x, \$y, displ**: Jump to address (\$pc+4) + 4*displ if register x == register y. **BNE \$x, \$y, displ**: Jump to address (\$pc+4) + 4*displ if register x != register y **BLT \$x, \$y, displ**: Jump to address (\$pc+4) + 4*displ if x < y **BGT \$x, \$y, displ**: Jump to address (\$pc+4) + 4*displ if x>y

LDI $1,0 # $1→0 counter
LDI $2, x # $2→A[0]
LDI $5, 5 # $5→5
Loop: BEQ $1,$5, End # start loop
LDI $3,3 # $3→3
LW $4,$2 # $4→A[i]
BEQ $4, $3, Skip
BGT $4, $3, Skip
PUSHI 5
PUSH $4
ADD # $4=$4+5
POP $4
SW $4,$2
Skip: INCW $2 # pointer at A[i]
INC $1 # i increment
BEQ $5,$5, Loop.
End:

4. (continued)

a. Using the instruction set given above, write the Zephyr assembly language equivalent of this program. Ensure that your code is properly commented. [5 marks]

```
for (i=0; i<5; i++)
    if (x[i] < 3)
        x[i] = x[i] + 5;
```

LDI $1, 0 # $1→0 counter
LDI $2, x # $2→&A[0]
LDI $5,5 # $5→5
Loop: BEQ $1, $5, End

All offsets in Zephyr are expressed as 16-bit word addresses, while registers are expressed as 3-bit register numbers (000_2=$1, 001_2=$2, ..., 111_2=$8). Similarly, all constants in Zephyr are 16-bit long.

There are six classes of instructions:

- A: No operands (e.g. ADD) 32 op
- B: One register operand (e.g. PUSH $1) 3b r 29 op
- C: One constant operand (e.g. PUSHI c) 16b c 16b op
- D: One register and one constant operand (e.g. LDI $1, c) 3b rt 16b constant 13b op
- E: Two registers (e.g. LW $1, $2) 3b rs 3b rt 26 op
- F: Two registers and a displacement (e.g. BEQ $1, $2, displ) 10b opcode 3b rs 3b rt 16b displ

b. Sketch the instruction formats for all 6 classes, assuming that all 32 bits of a Zephyr instruction word are utilized fully, and that we maximize the number of opcode bits possible each time. [12 marks]

Max: $2^5 - 2^2$
Min: $2^3 - 1 + 2^2$

c. If we utilize an expanding opcode scheme for Zephyr, what is the maximum number of opcodes possible, assuming that there are at least one instruction in each class? Show your working and reasoning process. Where convenient you may leave your answers in terms of powers of 2. [5 marks]

maximize 32bit opcode, then decreasing opcodes

4294966240

$2^{32} - 2^3 - 2^3 - 2^{10} - 2^3 - 2^3 + 5$

$2^{32} - 4(2^3 - 1) - (2^{10} - 1)$

d. What is the minimum number of opcodes possible, assuming that there are at least one instruction in each class? Show your working and reasoning process. Again, where convenient you may leave your answers in terms of powers of 2. [5 marks]

maximize 10bit opcode.

$2^{10} + 4(2^3 - 1) + (2^{10} - 1)$

$(2^{10} - 1) + 2^{10} + 4(2^3) + 2^{10} - 5 = 2075$

e. What is the furthest forward distance that you can branch to, in the BEQ, BNE, BLT and BGT instructions? Express your answer in number of instructions. [2 marks]

$PC = (PC+4) + 4 * 2^{15}$

2^{15} instructions = 32768.

f. Suppose that we have an array A of words (i.e. we access elements of A one word at a time). What is the maximum size of A, expressed in words? 2^{16} [3 marks]

g. Suppose again that we have an array B of bytes (i.e. we access elements of B one byte at a time). What is the maximum size of B, expressed in bytes? 2^{18} [3 marks]

5. **[15 marks]**
In this question we want to modify the (non-pipelined) MIPS datapath to support two new instructions: BLT and BGT – "branch on less than" and "branch on greater than".

The BLT and BGT instructions are shown below:

BLT:

6	5	5	16
0x08	rs	rt	displ

001000

beq: 000100

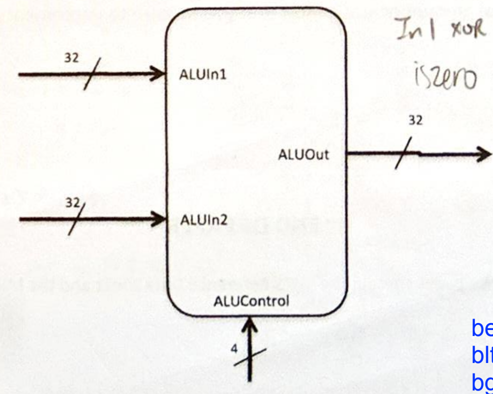
BGT:

0x12	rs	rt	displ

001100

a. The ALU for the MIPS processor is shown below as a single block with two 32-bit inputs, and one 32-bit output. Show, by adding AT MOST ONE 32-input logic gate and any additional wires, how to generate the **IsZero** and **IsNegative** signals. The IsZero signal is 1 when ALUIn1 – ALUIn2 is zero, and the IsNegative signal is 1 when ALUIn1 – ALUIn2 is negative. [4 marks]

In1 XOR In2 = 0 when isZero=1.
isZero = XNOR In1, In2.

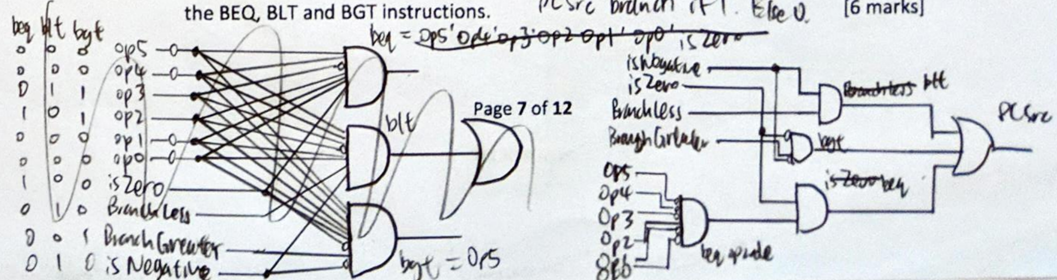


beq = 000100
blt = 001000
bgt = 001100

b. The CONTROL unit in the datapath must now generate **BranchLess** and **BranchGreater** signals from the instruction bits. Sketch the combinational circuits to generate these signals. [5 marks]

c. For your convenience the MIPS datapath is shown in page 10. Sketch the combinational logic circuit needed to generate the **PCSrc** control signal to support the BEQ, BLT and BGT instructions. [6 marks]

PCSrc branch if 1. Else 0.



6. **[20 marks]**
Suppose we have a cache that has an access time of 5ns, and a main memory with an access time of 80ns.

   a. What is the memory access time when you have a cache hit? 5ns ✓ [2 marks]

   b. What is the memory access time when you have a cache miss? 85ns ✓ [3 marks]

   c. You run some benchmarks on your system and find that 10,000 accesses take a total of 70 microseconds (1 microsecond = 1000 nanoseconds). What is the miss rate of your cache? [5 marks] 2.5%

$x(85) + (1-x)(5) = \cancel{70000}\ 7$

$80x + 5 = 7$

cache size $= 2^{16}$ bytes
block size $= 2^5$ bytes
set size $= 2^7$ bytes
index $= 9$ bits

Your cache is implemented as a 4-way set associative write-back cache totaling 64KB. Each cache block holds 8 words of 4 bytes each. CPU addresses are 32 bits long.

~~2^{16} bits~~
~~offset =~~

   d. How many bits per set do you require to store the tags? 18 ✓ [4 marks]

offset = 5 bits index = 9 bits tag = ~~14~~ 18

   e. Assuming that the 64KB of cache refers purely to "usable cache" – i.e. cache that is used only to store data or instructions, and not overheads like tag bits, what is the total amount of static RAM that you require to implement this cache? [6 marks]

cache size 2^{16} ~~bits~~ bytes $\nearrow 2^5$ bits

block size = 32 bytes = 8 words. offset = 5 bits

set size $= 2^7$ bytes

n. sets $= 2^9$ set index = 9 bits

tag = 18 bits

valid bit = 1 bit
tag = 18 bits
Stored words = 32 bits
$\} \times 4^2 \times 2^9$

Additional memory needed = ~~(1+18) × 4 × 2^9~~ 2^9

= ~~38912~~

~~ END OF PAPER ~~~

(The next two pages contain the MIPS Reference Data sheet and the MIPS Datapath.)

n sets $= 2^9$

set size $= 2^7$ bytes

add $2^9 \times 4(18+1)$ to 64KB

4864 bytes + 64×2^{10} bytes

70400 bytes

68.75KB ✓