# NATIONAL UNIVERSITY OF SINGAPORE

## SCHOOL OF COMPUTING

**EXAMINATION FOR**

**Semester 1 AY2010/11**

## CS2100 – COMPUTER ORGANISATION

**Nov 2010** **Time allowed: 2 hours**

---

**Your Matriculation Number:**



**INSTRUCTIONS TO CANDIDATES**

1. This examination paper consists of **SIX (6)** questions and comprises **TWENTY-ONE (21)** printed pages including this page.
2. This is an **OPEN BOOK** examination. You may use any approved calculators but not any PDA or laptop, especially those capable of external connectivity or communication.
3. Answer *all* questions. Note that the full mark for each question is different.
4. Write your answers on *this* **QUESTION AND ANSWER SCRIPT**. Answer only in the space given. Any writing outside this space will not be considered. No other submission is allowed.
5. Fill in your Matriculation Number with a pen, clearly on every page of this QUESTION AND ANSWER SCRIPT.
6. You may use pencil to write your answers.
7. At the end of the examination, please check to ensure that your script has all the pages properly stapled together.
8. Note that when a number is written as "0xNNNN" it means that "NNNN" is in base 16.

**Total Score** _____ /100

**QUESTION 1** (15 marks)

**(1a)** Encode the MIPS instruction "**`lui $5, 0xFEED`**", leaving your answer as an 8-hexadecimal digit number. (2 marks)

ANSWER:

**`0x3c05feed`**

**(1b)** Encode the MIPS instruction "**`beq $t1, $s0, Label`**", leaving your answer as an 8-hexadecimal digit number. Assume that the PC of this instruction is `0x1000` and `Label` is located at address `0x880`. (2 marks)

ANSWER:

Target_pc = (PC + 4) + (4 * Immed)

Immed = (0x880 – (0x1000 + 4)) / 4 = - 0x1E1 = 0xFE1F (in 16 bits)

**`0x1130fe1f`**

**(1c)** What is the MIPS instruction that is encoded by the hexadecimal number **`0x00016C42`**? (2 marks)

ANSWER:

**`srl $13, $1, 17`**

**(1d)** Convert following C code snippet into MIPS assembly.

```
sum = 0;
for (i=0; i<N; i++) {
      if (A[i] == i) continue;
      sum += A[i];
}
```

Note that: `sum` is in `$s0`, address of the word (4-byte) array `A` is in `$a0`, `i` is in `$t0`, and `N` is in `$s1`. (6 marks)

ANSWER:

```
          li    $s0, 0
          li    $t0, 0
Loop:     slt   $t1, $t0, $s1
          beq   $t1, $0, Exit
          sll   $t2, $t0, 2
          add   $t3, $t2, $a0
          lw    $t4, 0($t3)
          beq   $t4, $t0, Cont
          add   $s0, $s0, $t4
Cont:     addi  $t0, $t0, 1
          j     Loop
Exit:
```

**(1e)** Consider the following C code:

```
if (x > y) goto FARFARAWAY;
```

Suppose `x` is in `$t0`, and `y` is in `$t1`. The PC of this instruction is `0x100` and `FARFARAWAY` is located at address `0xEEEE1800`. Give the MIPS instructions that will implement this `if`-statement. (3 marks)

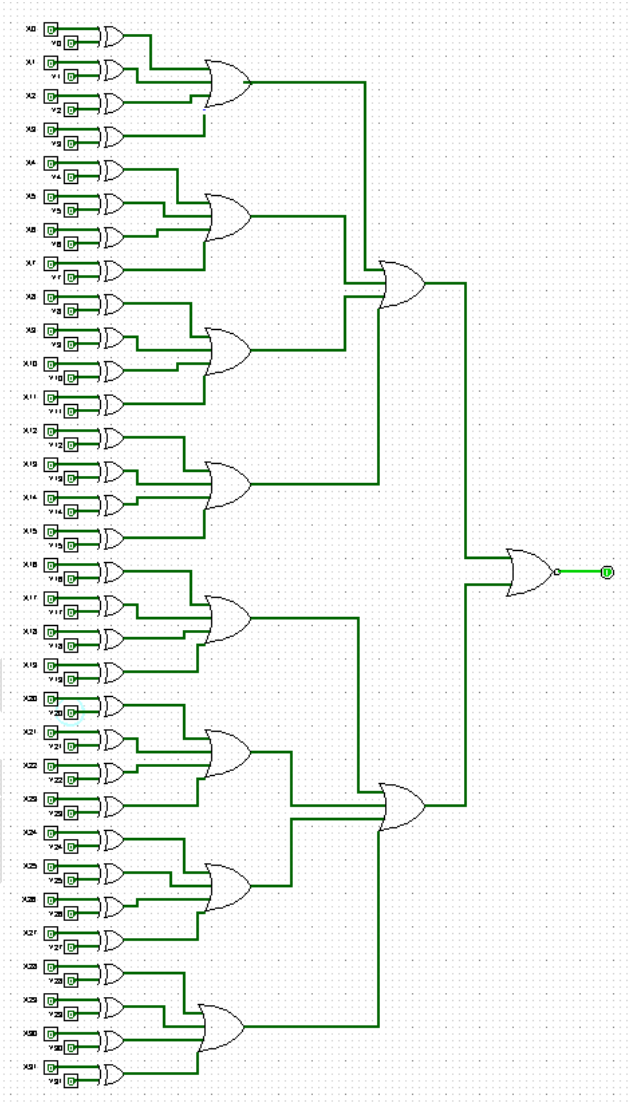
ANSWER:

```
          slt    $t2, $t1, $t0
          bne    $t2, $0, dojump
          j nojump
dojump:   j FARFARAWAY
nojump:   …
```

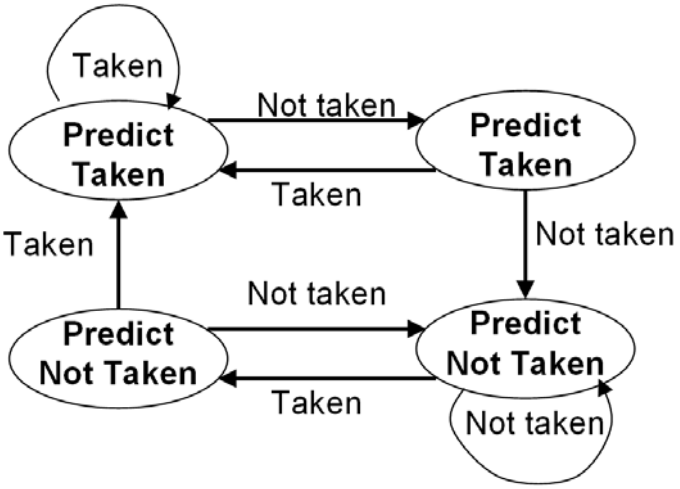
**QUESTION 2** (15 marks)

**(2a)** The MIPS beq/bne requires the comparison of two 32-bit quantities. If it is performed at the ALU, subtraction would be used. However, when we move it to the ID stage, we need a specialized comparator unit that compares the two quantity and outputs a single bit result indicating whether the two quantities are equal or not. There is no need for subtraction. Using basic gates, construct a circuit that will perform such a comparison. Assume that any gate has a maximum fan-in of 4 inputs. (5 marks)

ANSWER:

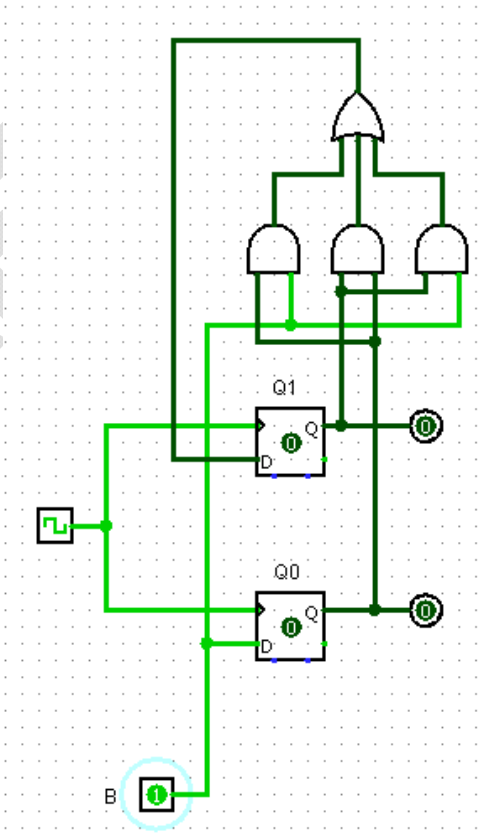


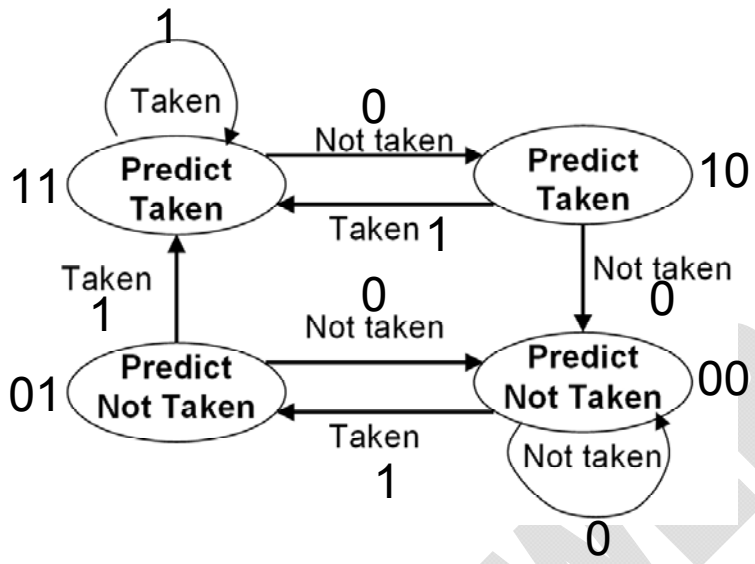
**(2b)** The following is the finite state machine for the 2-bit branch predictor.




Implement this finite state machine (including the prediction output) using two D-flip-flops, and the standard gates. The "taken/not-taken" ("0" = "not taken", "1" = "taken") is a single bit of input. The prediction, i.e., "Predict Taken/Not Taken" is a 1-bit output. Show clearly, how you encode the 4 states and your intermediate workings, including the truth table from which you derived the implementation. You may assume that the flip-flops have a fixed clock input. (10 marks)

ANSWER:






| Q1 | Q0 | B | D1 | D0 |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

$D1 = Q0 \cdot B + Q1 \cdot Q0 + Q1 \cdot B$

$D0 = B$

**QUESTION 3** (20 marks)

**(3a)** Given a NULL-terminated C string **`str`** (i.e., an array of characters terminated by an element that holds the value 0) that holds a string of at most 8 hexadecimal characters, write a MIPS assembly function "**`int convert(str)`**" that converts the hexadecimal string into an integer as the function's return result. The address of `str` is passed as the first argument to `convert`. Note: the ASCII code for '0' to '9'is $48_{10}$ to $57_{10}$, 'A' to 'F' is $65_{10}$ to $70_{10}$, and 'a' to 'f' is $97_{10}$ to $102_{10}$, respectively. If any of the characters is not in these ranges, an error status is returned by returning a '1' in `$v1`. In this case, the content of `$v0` does not matter. If there is no error, then `$v0` should contain the converted value, and `$v1` should be '0'. You may use the various branching pseudo-ops for your convenience. (10 marks)

ANSWER:

```
convert:
     li      $v0, 0       // The return result
     li      $t0, 0       // The index
     li      $t4, 4       // The shift amount
     li      $t1, 8       // Limit
Loop:
     sll     $v0, $v0, $t4 // First iteration, $v0 = 0
     lw      $t2, 0($a0)  // str[i]
     li      $t3, 57      // 48 <= A[i] <= 57
     bgt     $t2, $t3, C65
     li      $t3, 48
     blt     $t2, $t3, Err
     addi    $t2, $t2, -48
     add     $v0, $v0, $t2
     j       Next
C65:
     li      $t3, 70      // 65 <= A[i] <= 70
     bgt     $t2, $t3, C97
     li      $t3, 65
     blt     $t2, $t3, Err
     addi    $t2, $t2, -55
     add     $v0, $v0, $t2
     j       Next
C97:
     li      $t3, 102      // 97 <= A[i] <= 102
     bgt     $t2, $t3, Err
     li      $t3, 97
     blt     $t2, $t3, Err
     addi    $t2, $t2, -87
     add     $v0, $v0, $t2
Next:
     addi    $t0, $t0, 1
     blt     $t0, $t1, Loop
     li      $v1, 0
Return:
```

```
     jr      $ra
Err:
     li      $v1, 1
     j       Return
```

**(3b)** The *n* factorial is defined as `fac(n) = n * fac(n-1)` with `fac(1) = 1`, and `n > 0`. Write a MIPS assembly function that computes the *n* factorial where *n* is the input argument. (10 marks)

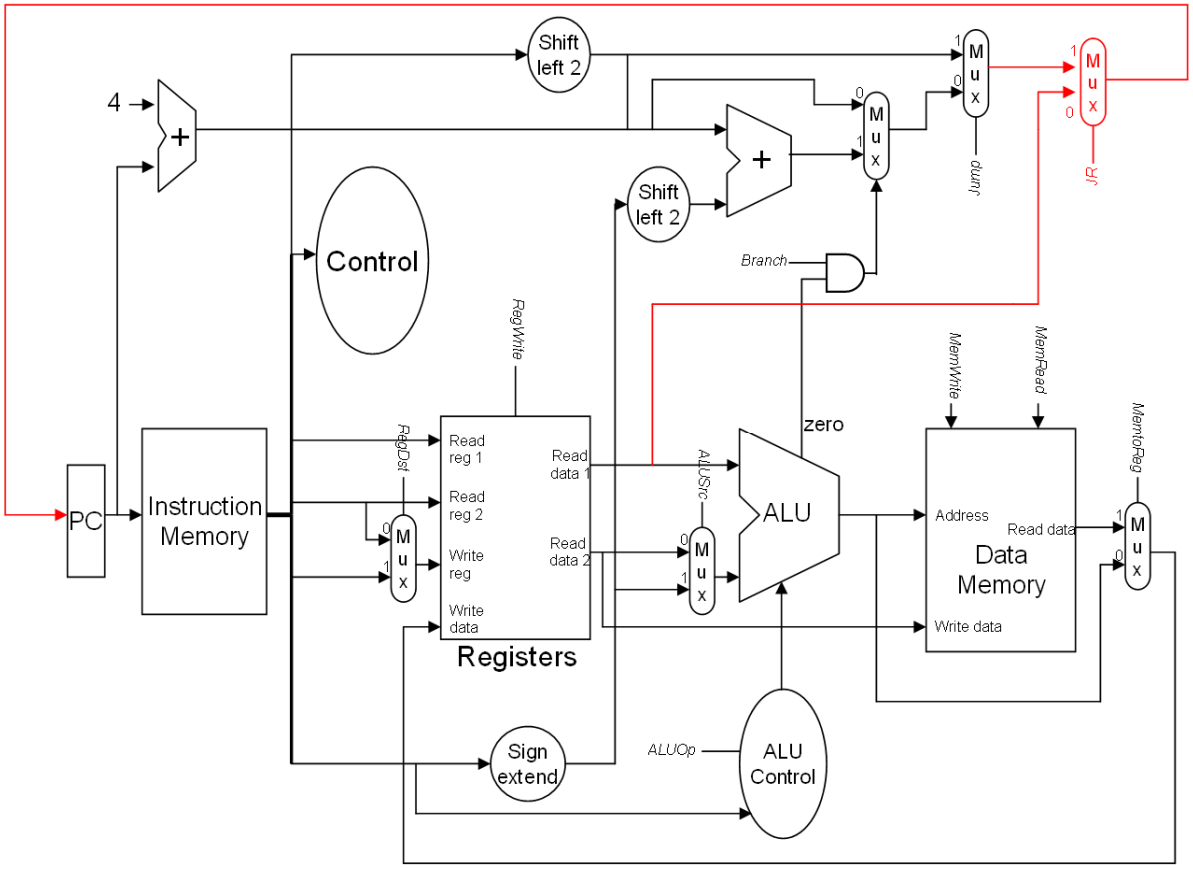
ANSWER:

```
.text
.globl main
# Prompt for a non-negative integer
# and invoke the factorial function.

main:
     addi $sp, $sp, -4   # Make space on stack.
     sw   $ra, 0($sp)    # Save return address.
     la   $a0, prompt
     li   $v0, 4
     syscall             # Display prompt.
     li   $v0, 5
     syscall             # Get integer response.
     move $a0, $v0  # Call factorial function.
     jal  factorial
     move $a0, $v0
     li   $v0, 1
     syscall             # Print integer result.
     la   $a0, endl
     li   $v0, 4
     syscall             # Print endl.
     li   $v0, 0         # Return zero.
     lw   $ra, 0($sp)    # Restore return address.
     addi $sp, $sp, 4    # Restore stack pointer.
     jr   $ra

factorial:
     addi $sp, $sp, -8   # Make space on stack.
     sw   $ra, 0($sp)    # Save return address.
     li   $v0, 1         # 0! = 1
     beqz $a0, zero # Special case for 0!
     sw   $a0, 4($sp)    # Save our argument.
     addi $a0, $a0, -1   # Calculate (n-1)!
     jal  factorial # Result in v0.
     lw   $a0, 4($sp)     # Restore our argument.
     mul  $v0, $a0, $v0  # n! = n * (n-1)!
zero:
     lw   $ra, 0($sp)    # Restore return address.
     addi $sp, $sp, 8    # Restore stack pointer.
     jr   $ra       # Return.

.data
prompt: .asciiz "Enter a non-negative integer: "
endl:     .asciiz "\n"
```

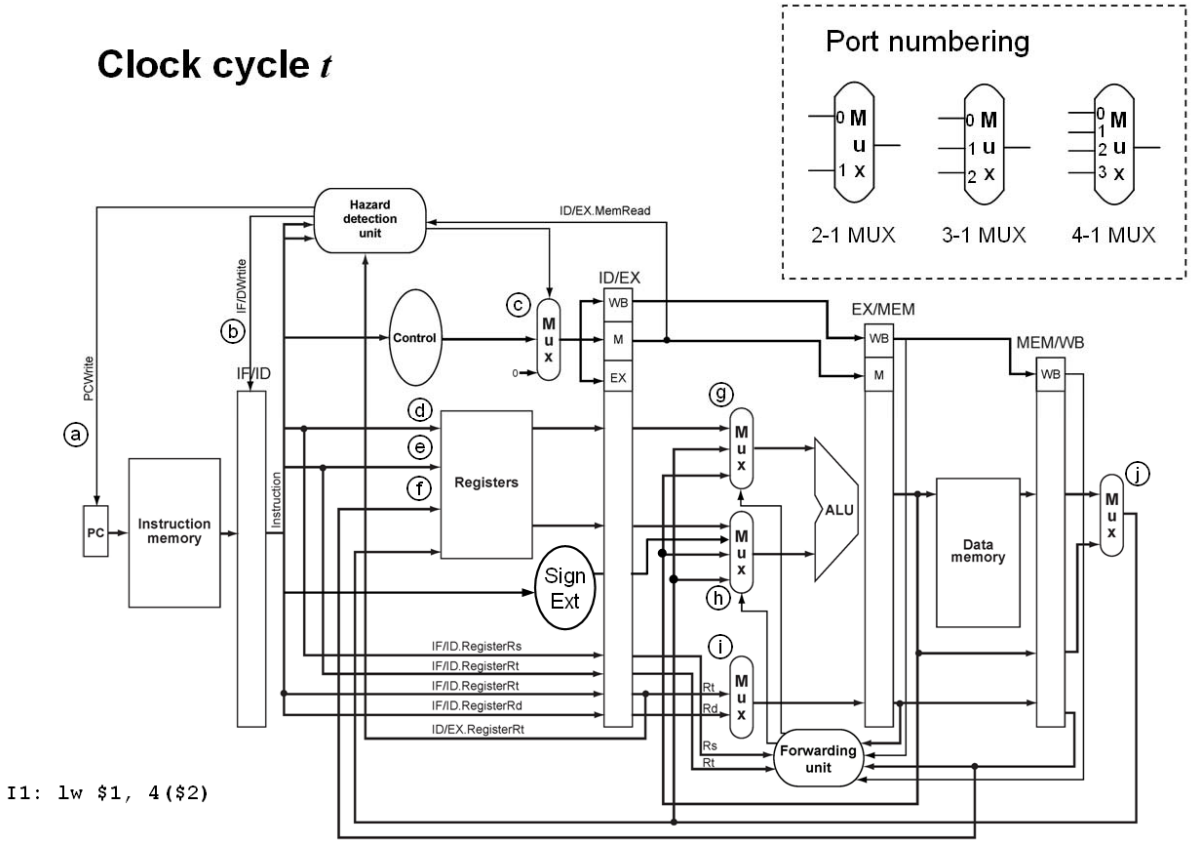
**QUESTION 4** (20 marks)

**(4a)** The diagram below is a simplified version of the single cycle datapath. By drawing additional components and signal lines, including possibly new control signals (which you should define), show how support for the **`jr`** ("jump register") instruction can be added. (10 marks)

ANSWER:



**(4b)** Consider the following pipelined data path at clock cycle *t*. (The multiplexor's inputs are labeled as per the diagrams in the box with the dotted border.) The instruction **`I1`**, namely **`lw $1, 4($2)`** is currently at the instruction fetch stage.




After **`I1`**, the following instructions enter the pipeline in order:

```
I2: add $1, $1, $1
I3: sub $2, $1, $5
```

**`I2`** starts execution in the IF stage in cycle *t*+1, **`I3`** starts at cycle *t*+2, and so on. For the cycles, what are the values of the respective signals marked by the circled letters? Assume that the instructions executing/executed prior to cycle *t* has no effect over the current execution (other than the default PC = PC+4). (10 marks)

ANSWER:

| Cycle | Signal | Value | Reason |
|---|---|---|---|
| *t*+1 | ⓐ | 1 | Enable PC write |
| | ⓑ | 1 | Enable IF write |
| | ⓒ | 0 | Pass on control signal |
| | ⓓ | 2 | Rs of LW |
| | ⓔ | 1 | Rt of LW |
| | ⓕ | ? | Unable to determine – depends on the previous instruction |
| *t*+2 | ⓐ | 0 | Hazard detected. Introduce a stall. |
| | ⓑ | 0 | Hazard detected. Introduce a stall. |
| | ⓒ | 1 | No-op injected. |
| | ⓓ | 1 | Rs of ADD |
| | ⓔ | 1 | Rt of ADD |
| | ⓕ | ? | Unable to determine – depends on the previous instruction |
| | ⓖ | 0 | Rs of LW |
| | ⓗ | 1 | Immediate of LW |
| | ⓘ | 0 | LW writes to Rt |
| *t*+3 | ⓐ | 1 | Resume instruction fetch |
| | ⓑ | 1 | Resume instruction decode |
| | ⓒ | 0 | Control for ADD |
| | ⓓ | 1 | Rs of ADD |
| | ⓔ | 1 | Rt of ADD |
| | ⓕ | ? | Unable to determine – depends on the previous instruction |
| | ⓖ | 0 | No-op |
| | ⓗ | 0 | No-op |
| | ⓘ | 0 | No-op |
| *t*+4 | ⓐ | 1 | Enable PC write |
| | ⓑ | 1 | Enable IF write |
| | ⓒ | 0 | Pass on control signal of SUB |
| | ⓓ | 1 | Rs of SUB |

| | | | |
|---|---|---|---|
| | ⓔ | 5 | Rt of SUB |
| | ⓕ | 1 | Rt of LW |
| | ⓖ | 1 | Forward from WB stage |
| | ⓗ | 3 | Forward from WB stage |
| | ⓘ | 1 | ADD writes to Rd |
| | ⓙ | 0 | LW data is from memory. |
| *t*+5 | ⓕ | 0 | NOP |
| | ⓖ | 2 | Forward Rs value from MEM stage |
| | ⓗ | 0 | Rt of SUB |
| | ⓘ | 1 | Rd of SUB |
| | ⓙ | 0 | NOP |
| *t*+6 | ⓕ | 1 | Rd of ADD |
| | ⓙ | 1 | Pass on ALU result |

**QUESTION 5** (20 marks)
A machine with byte addresses and a word size of 32 bits and address width of 32 bits has a small 1024-block *direct-mapped* write-back cache with each block consisting of 2 words.

**(5a)** Propose a C/Java data structure that can be used to hold all the content of the cache. You may assume that the data type "long" holds a 4-byte quantity. (4 marks)

ANSWER:

```
struct cache_block {
    long tag;       // Tag
    int valid_p;    // The valid bit
    int dirty_bit; // For write-back
    long word[2];
} cache[1024];
```

**(5b)** Using C or Java, write a function that will take a 32 bit address as an argument (typed as "long") and return a 32 bit data (again, a "long") if it is found in the cache. It sets a global (integer) flag to indicate a hit (1) or a miss (0). You do not need to concern yourself with further miss processing in the lower cache hierarchy. (8 marks)

ANSWER:

```
int hit_flag = 0;

long read_cache(long addr) {
    int blkno, wordno;
    long tag;

    wordno = (addr >> 2) & 1;
    blkno = (addr >> 3) & 0x3FF;
    tag = addr >> 13;

    if (cache[blkno].tag == tag) { // hit
       hit_flag = 1;
       return(cache[blkno].word[wordno]);
    }
    else { // miss
       hit_flag = 0;
       return 0;
    }
}
```

**(5c)** Using C or Java, write a code fragment will cause a lot of *compulsory* misses in this cache. (8 marks)

ANSWER:

The following loop will result in a lot of misses:

```
int A[huge_array_size];

for (i=0; i<some_big_N; i++) {

     A[random(i) % huge_array_size] = i;

}
```

where "`random(i)`" produces an integer random number between 0 and the maximum 32 bit integer.

**QUESTION 6** (10 marks)
A machine has a physical address width of 32 bits and a virtual address width of 32 bits. The following content of a full associative 4-entry TLB:

| Valid Bit | Virtual Page Number | Physical Page Number | Dirty | Ref | Access |
|---|---|---|---|---|---|
| 1 | 0x41081 | 0x1111 | 1 | 20 | rw |
| 1 | 0x6245 | 0xDEED | 0 | 1 | rx |
| 1 | 0x57C | 0x4AC | 0 | 8 | rw |
| 0 | 0x30A2 | 0x221 | 1 | 9 | rw |

**(6a)** Suppose an access to the memory location at virtual address `0x000AF9F4` is translated to the physical address `0x000959F4`, what is the size of a page in this machine? (2 marks)

ANSWER:

8Kbytes.

**(6b)** Suppose a malicious program tries to overwrite the text segment of the code by *writing* to the virtual page `0x6245` in the hope that the program will execute it, how would the system respond? (2 marks)

ANSWER:

A segmentation fault will be raised as the page is not writable.

**(6c)** A file consists of 8 bytes only. The following shows its content in hexadecimal.

| File byte position | Byte content |
|---|---|
| 0 | 0xD0 |
| 1 | 0xE1 |
| 2 | 0xA2 |
| 3 | 0xD3 |
| 4 | 0xB4 |
| 5 | 0xE5 |
| 6 | 0xE6 |
| 7 | 0xF0 |

Show how this file would be stored on 5 disk RAID level 3 system employing *odd* parity. "bit[*i*][*j*]" refers to "bit *j* of byte *i*" (with '0' being the least significant bit/byte). You need to write down the position as well as the value in the answer template. (6 marks)

ANSWER:

| **Disk 0** | **Disk 1** | **Disk 2** | **Disk 3** | **Disk 4** |
|---|---|---|---|---|
| bit[0][0] = 0 | bit[0][1] = 0 | bit[0][2] = 0 | bit[0][3] = 0 | bit = 1 |
| bit[0][4] = 1 | bit[0][5] = 0 | bit[0][6] = 1 | bit[0][7] = 1 | bit = 0 |
| bit[1][0] = 1 | bit[1][1] = 0 | bit[1][2] = 0 | bit[1][3] = 0 | bit = 0 |
| bit[1][4] = 1 | bit[1][5] = 1 | bit[1][6] = 1 | bit[1][7] = 1 | bit = 1 |
| bit[2][0] = 0 | bit[2][1] = 1 | bit[2][2] = 0 | bit[2][3] = 0 | bit = 0 |
| bit[2][4] = 0 | bit[2][5] = 1 | bit[2][6] = 0 | bit[2][7] = 1 | bit = 1 |
| bit[3][0] = 1 | bit[3][1] = 1 | bit[3][2] = 0 | bit[3][3] = 0 | bit = 1 |
| bit[3][4] = 1 | bit[3][5] = 0 | bit[3][6] = 1 | bit[3][7] = 1 | bit = 0 |
| bit[4][0] = 0 | bit[4][1] = 0 | bit[4][2] = 1 | bit[4][3] = 0 | bit = 0 |
| bit[4][4] = 1 | bit[4][5] = 1 | bit[4][6] = 0 | bit[4][7] = 1 | bit = 0 |
| bit[5][0] = 1 | bit[5][1] = 0 | bit[5][2] = 1 | bit[5][3] = 0 | bit = 1 |
| bit[5][4] = 0 | bit[5][5] = 1 | bit[5][6] = 1 | bit[5][7] = 1 | bit = 0 |
| bit[6][0] = 0 | bit[6][1] = 1 | bit[6][2] = 1 | bit[6][3] = 0 | bit = 1 |
| bit[6][4] = 0 | bit[6][5] = 1 | bit[6][6] = 1 | bit[6][7] = 1 | bit = 0 |
| bit[7][0] = 1 | bit[7][1] = 1 | bit[7][2] = 1 | bit[7][3] = 0 | bit = 0 |
| bit[7][4] = 1 | bit[7][5] = 1 | bit[7][6] = 1 | bit[7][7] = 1 | bit = 1 |



=== END OF PAPER ===