

**PROBLEM 1** (4 points)

Consider a device for non-volatile storage in flash memory. It has a 64 GB flash memory that contains 16 M blocks of 4 KB each ( $2^{24}$  blocks  $\times$   $2^{12}$  bytes). Figure 1 shows the details of the device adapter.

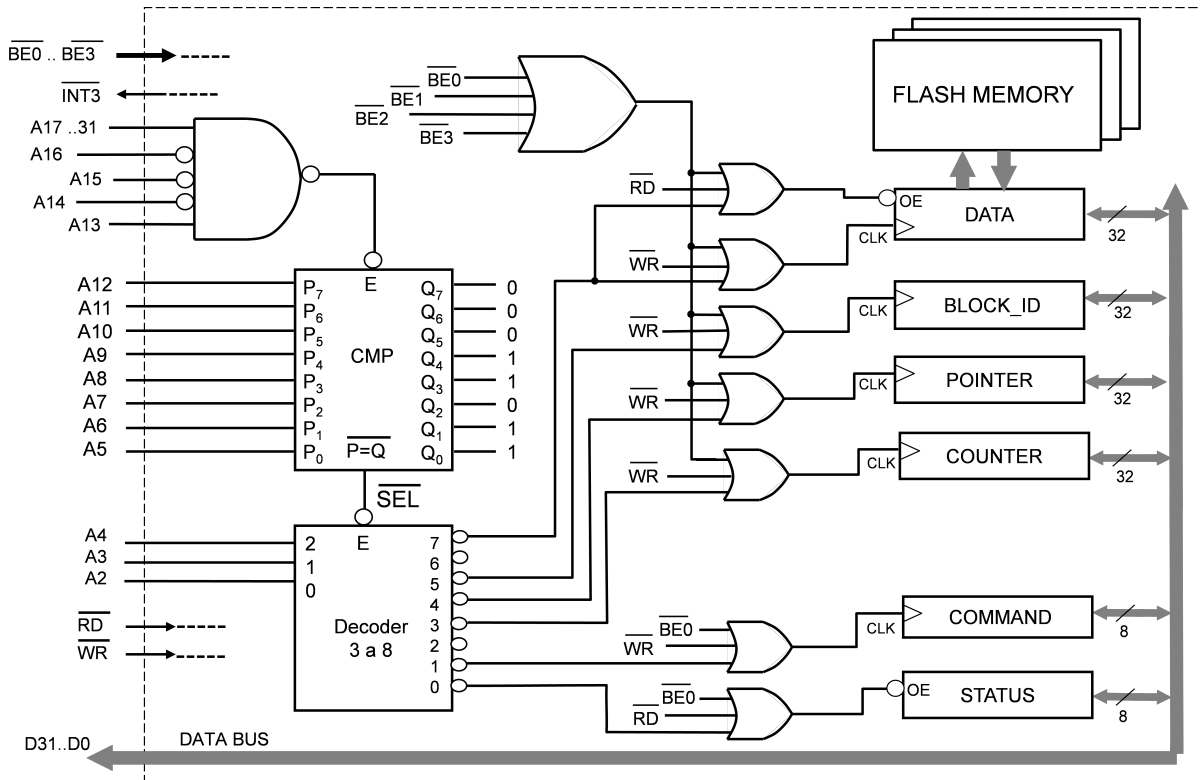


Figure 1: Adapter of the flash storage device.

1. (0.5 points) Identify the device's base address.

Base Address (BA) =

2. (0.5 points) Identify the addresses of the 6 registers in the adapter. Give also the type of operation permitted for each register, by specifying RO (read-only), WO (write-only), or RW (read-write).

Register	Address	Operation
STATUS	BA +	
COMMAND	BA +	
COUNTER	BA +	
POINTER	BA +	
BLOCK_ID	BA +	
DATA	BA +	

3. (0.5 points) Assume the label `Dir_Data` represents the address of the DATA register. What is the effect of executing the following piece of code?

```
.kdata
buffer: .space 512

.ktext
...
la $t0, Dir_Data
lb $t1, 0($t0)
sb $t1, buffer
...
```



4. (0.5 points) We want to build a new version of this adapter that will be connected to a processor that is exactly like the MIPS R2000 in all aspects but one: it has separate address maps for memory and I/O. We want the new adapter be mapped only in the I/O address space of this processor, and not in its memory map. Indicate the modifications needed in the adapter to build this new I/O-map-only version. Draw your proposed changes directly on figure 1.
5. (0.5 points) Consider now the details of the adapter registers:

**STATUS** Reflects the status of the peripheral. The only meaningful bit in this register is:

- Bit 4: Ready (R) – Indicates readiness of the device for a new operation. For programmed I/O transfers, R becomes 1 every time there is a new data item in the DATA register. For DMA transfers, R becomes 1 every time a block transfer terminates.

**COMMAND** Serves to program the peripheral. Contains the following meaningful bits:

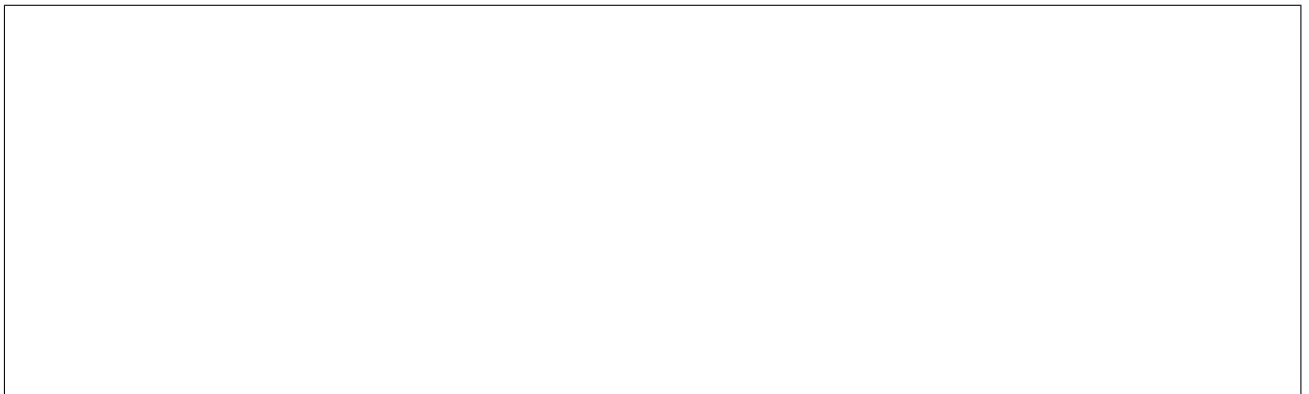
- Bits 0..3: Command. Commands issued to the adapter are encoded as follows:
  - 0x0: No operation
  - 0x1: Read block in DMA mode
  - 0x2: Write block in DMA mode
  - There are other codes are not listed here
- Bit 7: Enable (E) – Enables interrupt requests via `intr3*`. When  $E = 1$  an interrupt 3 is requested every time the Ready bit becomes 1. Setting  $E = 0$  disables interrupt requests from the adapter.
- Bit 5: Clear (C) – Setting this bit to 1 clears (sets to 0) the Ready bit, thereby canceling the corresponding interrupt request (when applicable).

**BLOCK\_ID** Identifier of the memory block to be transferred.

**POINTER** For DMA operation, this register holds the address of the memory block to read or write.

**COUNTER** Number of 32-bit words involved in a block transfer.

Write a sequence of code that reads one word from the DATA register, in programmed I/O mode. You must first poll the Ready bit in the STATUS register until DATA contains fresh data, and then transfer the read word to the memory variable `New_Data`.



6. (1 point) The adapter is supplied with a library of system functions, including:

Function	Code	Arguments	Result
Read_DMA	\$v0 = 900	\$a0: Pointer to memory buffer \$a1: Block number (BLOCK_ID) \$a2: Number of words to transfer	None
Write_DMA	\$v0 = 901	Same as Read_DMA	None

Write the code for the system function `Read_DMA`. This function must program a DMA block read from the flash memory. Termination of the full block transfer will be indicated by interrupt `intr3*`.

`Read_DMA:`

```
jal suspend_this_process
b retexc
```

7. (0.5 points) Write the code for `intr3*` for the handling of DMA transfers. This means the interrupt request is only issued when a full block has been transferred. Note also that we want the interrupts disabled in the adapter once `intr3*` has been processed.

`intr3:`

```
jal activate_waiting_procs
b retexc
```

**SOLUTION:**

1. By examining how address lines are decoded in the adapter, the base address is  $0xFFFE2360$ .
2. Offsets relative to the base address can be deduced by examining the decoding of lines  $A_4$ ,  $A_3$  and  $A_2$ . The mode of operation of the different registers can be deduced by looking at the connections of lines  $\overline{RD}$  and  $\overline{WR}$ .

Register	Address	Operation
STATUS	BA + 0	RO
COMMAND	BA + 4	WO
COUNTER	BA + 12	WO
POINTER	BA + 16	WO
BLOCK_ID	BA + 20	WO
DATA	BA + 28	RW

3. The code proposed has no effect since it accesses the DATA register as a byte, by using `lb` and `sb` instructions. This register can only be accessed as a word, since all  $\overline{BE}_i$  lines take part in the selection logic of the register.
4. Having two separate maps implies that the processor has an output (eg.  $M/\overline{IO}$ ) to signal which map an address refers to. The  $M/\overline{IO}$  line should be connected to an additional inverting input of the NAND gate.
5. Code to read a word by programmed I/O with polling synchronization:

```

...
    la $t0, 0xFFFE2360
poll: lb $t1, 0($t0)
    andi $t1, $t1, 0x10
    bne $t1, $zero, poll
    lw $t1, 28($t0)
    sw $t1, New_Data
...

```

6. Code for the Read\_DMA system function:

```

Read_DMA:
    la $t0, 0xFFFE2360 # Base Address
    sw $a1, 20($t0)     # Block_ID
    sw $a0, 16($t0)     # Pointer
    sw $a2, 12($t0)     # Counter
    li $t1, 0x81        # Command: Read_DMA and E=1
    sb $t1, 4($t0)

    jal suspend_this_process
    b retexc

```

7. Code for the interrupt handler. Note that it only needs to cancel the interrupt and disable further interrupts from the adapter.

```

intr3:
    la $t0, 0xFFFE2360 # Base Address
    li $t1, 0x20        # Command: No operation, plus Clear=1 and E=0
    sb $t1, 4($t0)

    jal activate_waiting_procs
    b retexc

```

**PROBLEM 2 (4 points)**

The left hand side of figure 2 shows schematically the installation of a freight elevator in a two-floor building. The two sensors, TOP and BOTTOM, detect the arrival of the elevator in the respective floors. The motor just below the roof moves the elevator up and down as required. Operation is controlled with the three-position handle at the bottom floor. Moving the handle to the UP and DOWN positions makes the elevator move accordingly. Moving the handle to the STOP position stops the motor if it is moving, and has no effect otherwise. By construction, the handle is always in one (and only one) of the three positions.

The right hand side of figure 2 shows the elevator interface, that is connected to a computer based on a MIPS R2000 processor. It contains two 8-bit registers, STATUS and CONTROL, that are mapped and structured as follows:

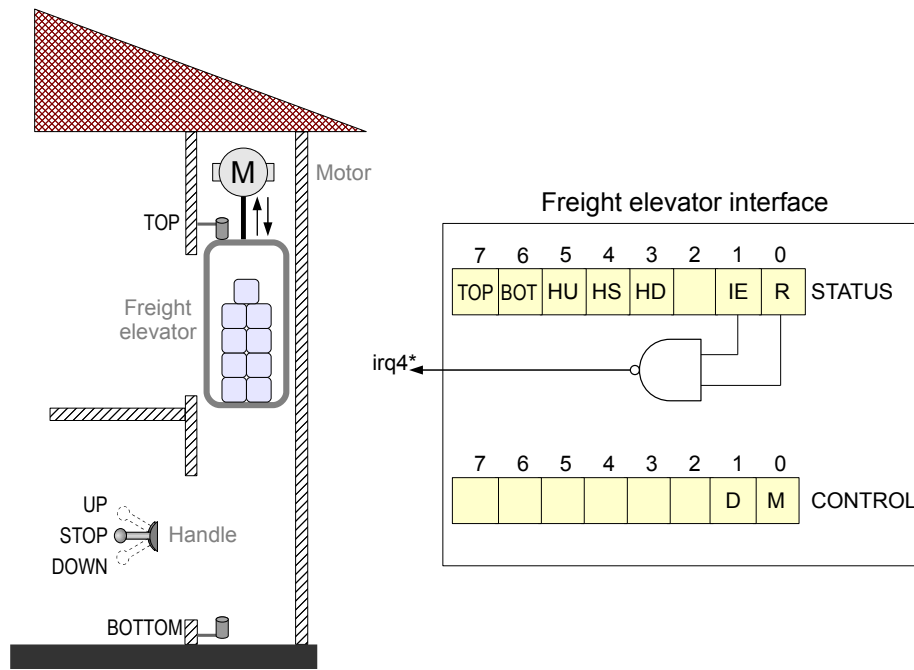


Figure 2: (Left) Freight elevator installation. (Right) Elevator interface.

**STATUS register** (read/write, base address 0xFFFF0010)

**TOP:** Set to 1 by the hardware while the elevator is in the top floor; 0 otherwise.

**BOT:** Set to 1 by the hardware while the elevator is in the bottom floor; 0 otherwise.

**HU:** Set to 1 by the hardware when the handle is in the UP position; 0 otherwise.

**HS:** Set to 1 by the hardware when the handle is in the STOP position; 0 otherwise.

**HD:** Set to 1 by the hardware when the handle is in the DOWN position; 0 otherwise.

**IE:** Setting this bit to 1 (by software) enables the device to request interrupts to the CPU. Conversely, setting the IE bit to 0 by software, disables the interrupt request mechanism.

**R:** The Ready bit. It is set to one by the hardware whenever there is a change **from 0 to 1** in any sensor bits (bits 3 to 7 in the STATUS register). This implies that the TOP and BOT sensors activate the Ready bit only when the elevator reaches the corresponding floor, but not when it leaves a floor. Note that when IE=1, the setting of the Ready bit produces an interrupt request connected to the CPU's irq4\* input. The Ready bit must be set to 0 by software once a change is detected and processed. When interrupts are enabled, resetting the Ready bit effectively cancels the interrupt.

Special care must be taken when writing bits IE and R, so that the rest of bits in this register remain unchanged.

**CONTROL register** (write only, base address + 4)

**M:** This bit controls the motor operation, and hence the elevator's motion. It must be set to 1 to make the motor move. The motor remains stopped while the value of this bit is 0.

**D:** Sets the direction of movement for the elevator. D=1 for upward, and D=0 for downward.

1. (0.4 points) From the description of the adapter given above, explain and justify whether synchronization with the elevator must be only via interrupts, or only via polling, or with either of the two techniques.

2. (0.4 points) At the final part of the system startup phase, the following initialization code is executed just before jumping to the user program:

```

...
1  la $t0, 0xFFFF0010
2  lb $t1, 0($t0)
3  ori $t1, $t1, 0x2
4  sb $t1, 0($t0)
5  li $t0, 0x3402
6  mtc0 $t0, $Status
7  jal main          # Jump to user program
...

```

Does this code enable the handling of elevator interrupts? Either justify why it does, or propose changes in the previous code if it does not. Figure 3 gives the details of the Cause and Status registers of the MIPS R2000.

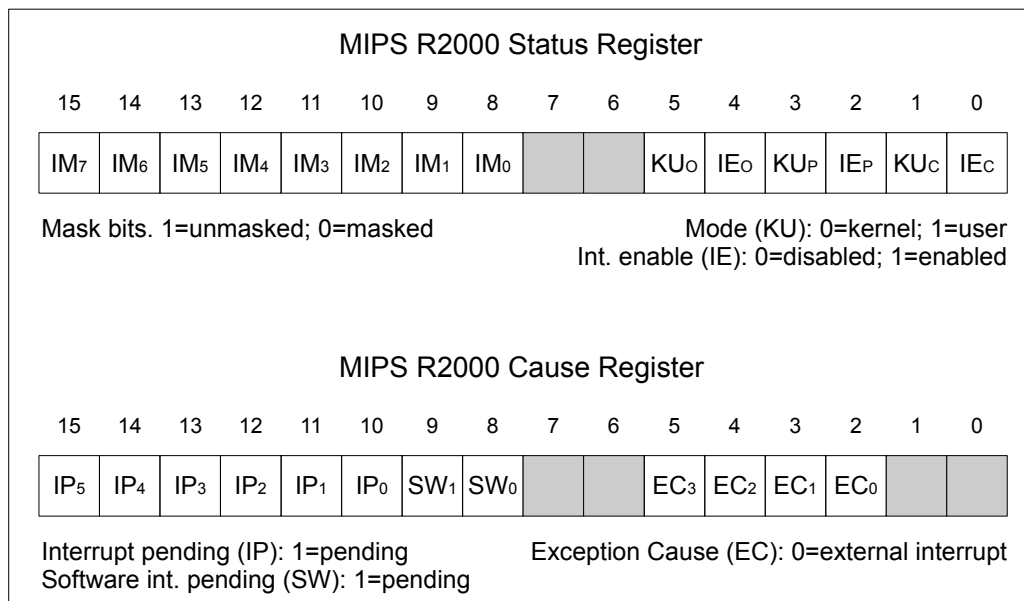


Figure 3: Details of the Cause and Status registers of the MIPS R2000.

3. (0.4 points) The following code is an excerpt from the system exception handler. In particular, it is a part of the code in charge of finding the source of an interrupt:

```
...
mfc0 $k0,$Cause
andi $t0,$k0,0x3C
beq $t0,$zero,int
...
int: ...
andi $t0, $k0, 0x1000
bne $t0, $zero, cod_intW
andi $t0, $k0, 0x2000
bne $t0, $zero, cod_intX
andi $t0, $k0, 0x4000
bne $t0, $zero, cod_intY
andi $t0, $k0, 0x8000
bne $t0, $zero, cod_intZ
...
```

By examining this code, justify which of the four labels (`cod_int{W|X|Y|Z}`) is the one corresponding to the code for handling elevator interrupts.

4. (0.4 points) Write the code for three routines, `move_up`, `move_down` and `stop` to make the elevator move up and down and to stop it, respectively. The functions will reside in the kernel space and will be called only while the CPU is in kernel mode. You are only allowed to use registers `$s0` and `$s1` from within these routines.

`move_up:`

`jr $ra`

`move_down:`

`jr $ra`

`stop:`

`jr $ra`

5. (2 points) Write the code for the elevator's interrupt service routine. The required behavior is that the system reacts properly to changes in the handle position and to the activation of the TOP and BOTTOM sensors. Moving the handle up makes the elevator move up to the top floor, unless the TOP bit is already 1. Similarly, moving the handle down makes the elevator move down to the bottom floor, unless the BOT bit is 1. Moving the handle to the stop position makes the motor stop. The reaction to activation of bits TOP and BOT is simply to stop the motor. You can use the routines `move_up`, `move_down` and `stop` from the previous question. Assume the system's exception handler preserves the values of registers `$t0...$t4` and `$s0...$s1`.

If you don't have a better idea, you can follow the algorithm proposed below in your design. If you chose another approach, give also the pseudocode of your algorithm:

```

elevator_int: ...
check_HU:      if HU=1 then          -- Detect handle up
                if TOP=1 then        -- Elevator already at top floor?
                    b stop_M          -- Yes, stop motor
                else
                    move elevator up  -- No, move elevator up
                    b continue
                end if
            end if
check_HD:      if HD=1 then          -- Detect handle down
                if BOT=1 then        -- Elevator already at bottom floor?
                    b s top_M         -- Yes, stop motor
                else
                    move elevator down -- No, move elevator down
                    b continue
                end if
            end if
stop_M:        stop motor            -- Rest of cases (TOP, BOT, HS), stop motor
continue:      ...

```

6. (0.4 points) Write the code for a system function with the following specification:

Name	Index	Arguments	Result
Elevator_Position	\$v0 = 333	—	\$v0 = Position code

The position code returned by the system function must indicate whether the elevator is at the bottom floor (value 1), or top floor (value 2) or it is somewhere in between both floors (value 0). As a hint, note that this code selection is somewhat inspired by the encoding of bits TOP and BOT in the elevator's status register. Note also that TOP and BOT can never be both 1 simultaneously.



## SOLUTION:

1. The adapter can be used either via interrupts or by polling. By interrupts because it has an interrupt request line with the appropriate functionality. And by polling because changes can be detected by inspection of the READY bit.
2. The code unmask `irq4*` and enables interrupts at the device level. However, instruction 5 uses a constant `0x3402` to write the processor's status register, so it does not enable interrupts globally because bit 0 is 0 in `0x3402`, whereas it should be 1 to set  $IE_C$ . Consequently, the only change needed is to replace the constant in instruction 5 with `0x3403`.
3. The label for handling interrupt request 4 is `cod_intY`, since the bit corresponding to a pending `irq4` in the cause register is bit 14, and  $0x4000 = 2^{14}$ .
4. Proposed code for the required routines:

```
move_up:    la $s0,0xFFFF0010
            li $s1,3
            sb $s1,4($s0)
            jr $ra

move_down:  la $s0,0xFFFF0010
            li $s1,1
            sb $s1,4($s0)
            jr $ra

stop:       la $s0,0xFFFF0010
            sb $zero,4($s0)
            jr $ra
```

5. Interrupt service routine for handling interrupt requests from the elevator, following the proposed algorithm.

```
elevator_int:
    la $t0,0xFFFF0010
    lb $t1,0($t0)          -- Read elevator's status register
    andi $t2,$t1,0x20      -- Mask out HU
    beq $t2,$zero,check_HD
    andi $t2,$t1,0x80      -- Mask out TOP
    bne $t2,$zero,stop_M
    jal move_up
    b continue

check_HD:
    andi $t2,$t1,0x08      -- Mask out HD
    beq $t2,$zero,continue
    andi $t2,$t1,0x40      -- Mask out BOT
    bne $t2,$zero,stop_M
    jal move_down
    b continue

stop_M:
    jal stop               -- Rest of cases (TOP, BOT, HS), stop motor

continue:
    andi $t2,$t1,0xFE      -- Reset Ready bit to cancel interrupt
    sb $t2,4($t0)
    b retexc
```

6. Implementation of the system function `Elevator.Position`. The value of bits TOP and BOT encodes the position as requested. We only need to shift these two bits 6 positions to the left, so that they take bits 0 and 1 of the result.

```
elevator_position:
    la $t0,0xFFFF0010
    lbu $t1,0($t0)         -- Read elevator's status register
    srl $v0,$t2,6          -- Shift right and place in result register
    b retexc
```

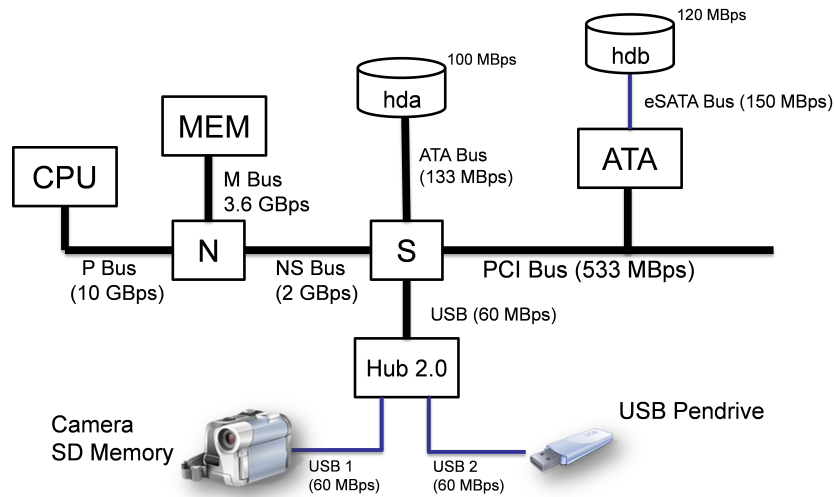


Figure 4: Problem context.

### PROBLEM 3 (2 points)

Figure 4 shows the bus structure and peripherals of a given computer. Consider the transfer of a file from the camera's SD memory to the hard disk hdb and to the pendrive. Transfers are in both cases of type DMA (camera to memory and memory to disk).

1. (0.75 points) Assume all transfers are **sequential** (first read full file from camera, then write full file to disk). Calculate the minimum time required for transferring a 1 GB file **from camera to disk hdb**.

2. (0.75 points) Now assume transfers are **concurrent** (blocks are written to the disk at the same time they are being read from the camera). Calculate the minimum time required for transferring a 1 GB file **from camera to the pendrive**.

3. (0.5 points) In the previous case (concurrent transfer from camera to pendrive), calculate the utilization of all busses involved, in percentage of their maximum bandwidth.

**SOLUTION:**

1. Transfer 1 GB from camera to hub: In the camera-to-memory transfer, the limiting bandwidth corresponds to the bus labelled USB1, at 60 MBps. In the memory-to-disk transfer, the limiting element is the disk, at 120 MBps. The first transfer occurs at 60 MBps, so it takes  $1 \text{ GB} / 60 \text{ MBps} = 16.7$  seconds. The second transfer takes  $1 \text{ GB} / 120 \text{ MBps} = 8.3$  seconds. In total, by adding these two times, the transfer takes 25 seconds.
2. Transferring from the camera to the pendrive, concurrently, implies that the USB bus (connecting the Hub 2.0 with the South bridge, must support both the traffic from the camera to memory and from memory to the pendrive. So the effective bandwidth of 60 MBps is equally shared at an effective 30 MBps. The total time required is hence  $1 \text{ GB} / 30 \text{ MBps} = 33.3$  seconds.
3. Utilization of the busses depends on the  $30 + 30 = 60$  MBps traffic and the maximum bandwidth of the bus in particular. Except only for the USB busses connecting the camera and the pendrive to the computer. In particular:

**M Bus:**  $(30+30) \text{ MBps} / 3600 \text{ MBps} = 1.67 \%$

**NS Bus:**  $(30+30) \text{ MBps} / 2000 \text{ MBps} = 3 \%$

**USB Bus:**  $(30+30) \text{ MBps} / 60 \text{ MBps} = 100 \%$

**HUB-Camera Bus:**  $30 \text{ MBps} / 60 \text{ MBps} = 50 \%$

**HUB-Pendrive Bus:**  $30 \text{ MBps} / 60 \text{ MBps} = 50 \%$