# *UNIVERSITY OF VICTORIA*

# *EXAMINATIONS SPRING 2007*

# *C SC 230 - Introduction to Computer Architecture and Assembly Language*

**STUDENT NUMBER:** _____

**TIME:** 3 hours

**INSTRUCTOR:** M. Serra

**TOTAL MARKS:** 100

**TO BE ANSWERED ON THE PAPER**

| Question No. | Value | Mark | Question No. | Value | Mark |
|---|---|---|---|---|---|
| 1 | *7* | | 8 | *12* | |
| 2 | *4* | | 9 | *2* | |
| 3 | *10* | | 10 | *8* | |
| 4 | *14* | | 11 | *7* | |
| 5 | *9* | | 12 | *12* | |
| 6 | *5* | | **TOTAL** | **100** | |
| 7 | *10* | | | | |

**INSTRUCTIONS:**

1. STUDENTS MUST COUNT THE NUMBER OF PAGES IN THIS EXAMINATION PAPER BEFORE BEGINNING TO WRITE, AND REPORT ANY DISCREPANCY IMMEDIATELY TO THE INVIGILATOR
2. This examination paper consists of 17 pages including this cover page.
3. No aids are permitted. However, a handout describing the ARM instruction set is provided for your use.
4. The marks assigned to each question are shown within square brackets. Partial marks are available for all questions
5. Please be precise but brief, and use point form where appropriate.
6. It is strongly recommended that you read the entire exam through from beginning to end before beginning to answer the questions.

**Question 1.** [7] Fill in the most appropriate ARM assembly language instructions to accomplish the tasks stated. Some tasks may require more than one instruction.

**(a)** [1] `r1 = FL1`, where `FL1` is a variable declared in the "data" section.

**(b)** [1] `r2 = address of FL2`, where `FL2` is a variable in the "data" section.

**(c)** [1] `r3 = r1 - r2` and the condition codes in the CPSR are set.

**(d)** [2] `r4 = r3 * 5` in 1 instruction only!

**(e)** [2] You are given that:

- R1 contains the current value of variable `"i"`;
- R2 contains the address of an array of integers declared as `"MyList"`;
- Each integer is 4 bytes in size.
- R3 contains the current value of variable `"Temp"`.

Give the *single* ARM instruction equivalent to:

```
MyList[i] = Temp;
```

**Question 2.** [4] **(a)** The range of decimal values that can be represented as unsigned 8 bit integers is:

( _____ , _____ )

**(b)** The range of decimal values that can be represented as signed 16 bit 2's complement integers is:

( _____ _____ , _____ _____ )

**Question 3.** [10] A subroutine is recursive if its execution involves calling itself. Many interesting problems are inherently recursive (e.g. tree traversal). When a recursive routine calls itself, it must use the stack for parameters and a stack frame for the local environment. In a high level language, the implementation can be logically straightforward. In a low level language, one must do explicit stack manipulation. Here you will explore a recursive implementation of the computation for factorial, one of the easiest, both using the C language and ARM Assembly language.

The normal definition of factorial is: $N! = N \times (N-1) \times (N-2) \times \ldots \times 3 \times 2 \times 1$, for $(N \geq 1)$ .

The recursive definition can be given as:

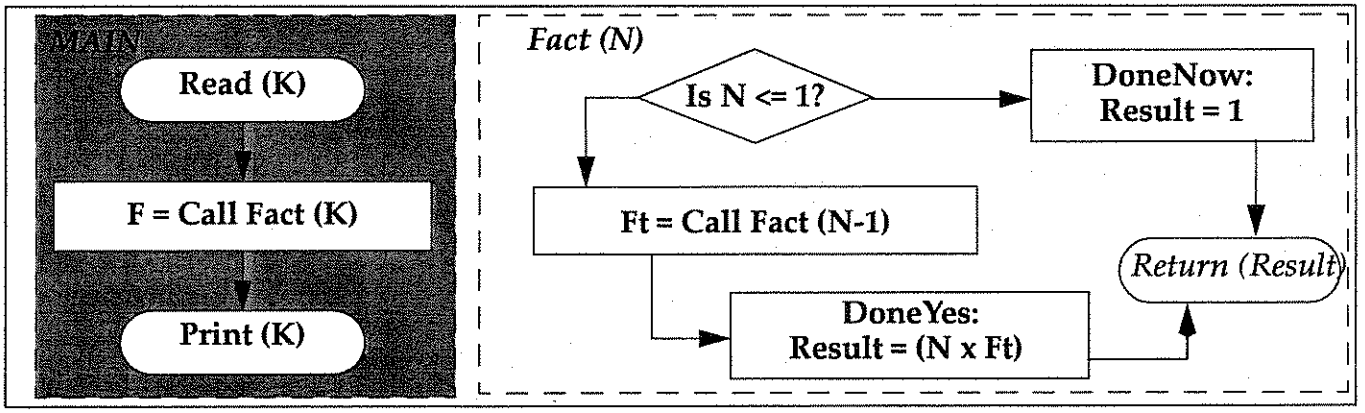
Base conditions: $0! = 1$
$1! = 1$

Recursive step: $N! = N \times (N-1)!$ for $N \geq 2$ .

Figure 1 shows a possible flowchart to guide the implementation.

**(a)** [6] Write the precise C code for this recursive implementation of the factorial computation. The program should be called "`RF.c`", the function is called "`Fact`" with its signature being "`int Fact (int N)`". *Use the variable names as specified.*

**FIGURE 1. Flowchart for a recursive implementation of Factorial**




**Code for `RF.c`**

**(b)** [4] Now examine the ARM code already written for you for this implementation in Figure 2. Something is wrong with the code. State what is incorrect [2 marks] and give the instructions to correct the code [2 marks] at the bottom of Figure 2.

**FIGURE 2. Recursive ARM implementation of Factorial**

```
_start:
[1]   LDR   r1,=K
[2]   LDR   r1,[r1]          @ r1 = K
[3]   STR   r1,[sp,#-4]!     @ push K on stack
[4]   BL    Fact             @ r0 = Fact(K)
[5]   BL    print_int        @ print result from r0
DoneRF:
[6]   swi       0x11
Fact:
[7]   STMFD     sp!,{r1-r3,fp,lr}   @ save registers
[8]   ADD   fp,sp,#12        @ set frame pointer
[9]   LDR   r2,[fp,#8]       @ r2 = parameter N
[10]  CMP   r2,#1            @ if N <= 1, done
[11]  BLE   DoneNow
[12]  MOV   r3,r2            @ else copy N to r3
[13]  SUB   r3,r3,#1         @ r3 = N-1
[14]  STR   r3,[sp,#-4]!     @ N-1 on stack
[15]  BL    Fact
[16]  MUL   r3,r2,r0         @ r3 = N * (N-1)!
[17]  MOV   r0,r3            @ r0 = return value
[18]  BAL   DoneFact
DoneNow:
[19]  MOV   r0,#1            @ return value
DoneFact:
[20]  LDMFD     sp!,{r1-r3,fp,pc}
```

**Question 4.** **[14]** The ARM program in Figure 3 should be vaguely familiar to you, as it was used in the lectures as an examples of nested function calls, with parameters passed through the stack. The labels and names have been changed, however. You are expected to trace the execution of this code (some unnecessary segments are missing) and answer the questions below, all relating to the content of the stack, memory and registers at various times

**FIGURE 3. Nested Function Calls**

```
[1]   LDR  R1,=X
[2]   LDR  R1,[R1]          @R1 = 1st parameter
[3]   LDR  R2,=Y
[4]   LDR  R2,[R2]          @R2 = 2nd parameter
[5]   STR  R1,[SP,#-4]!     @push parameters on stack
[6]   STR  R2,[SP,#-4]!
[7]   BL   FT1          @call R0 = FT(X,Y)
[8]   ADD  SP,SP,#8     @clean the stack
[9]   LDR  R1,=RES      @R1 = address of result
[10]  STR  R0,[R1]      @store result
[11]  SWI  0x11         @stop execution
@**************Function FT1 ********************
      FT1:                  @int FT1 (r int, s int) on stack
[12]   STMFD SP!,{R2-R3,R5,FP,LR}          @save registers
[13]   ADD FP,SP,#12                       @set frame pointer
[14]   LDR R2,[FP,#8]                      @copy parameter 1
[15]   LDR R3,[FP,#12]                     @copy parameter 2
          . . . more code for FT1 uses R2, R3, R5 . . . (omitted)
      @ get ready to call FT2 by placing its parameter on stack
      @assume that R2 contains the integer parameter
[20]  STR  R2,[SP,#-4]!           @push parameter on stack
[21]  BL   FT2                    @R0 = FT2 (i) on stack
[22]  ADD  SP,SP,#4               @clean the stack
          . . . more code for FT1 uses R2, R3, R5 . . . (omitted)
      @ready to return from FT1 - set the return value in R0
[30]  MOV  R0, R5
[31]  LDMFD SP!,{R2-R3,R5,FP,PC} @
@******Function FT2 *****
FT2:
[32]   STMFD SP!,{R1-R2,FP,LR}
[33]   ADD FP,SP,#8
[34]   LDR R1,[FP,#8]
          . . . more code for FT2 uses R1, R2, . . . (omitted)
      @ready to return from FT2 - set the return value in R0
[40]  MOV  R0, R2
[41]  LDMFD SP!,{R1-R2,FP,LR}
```

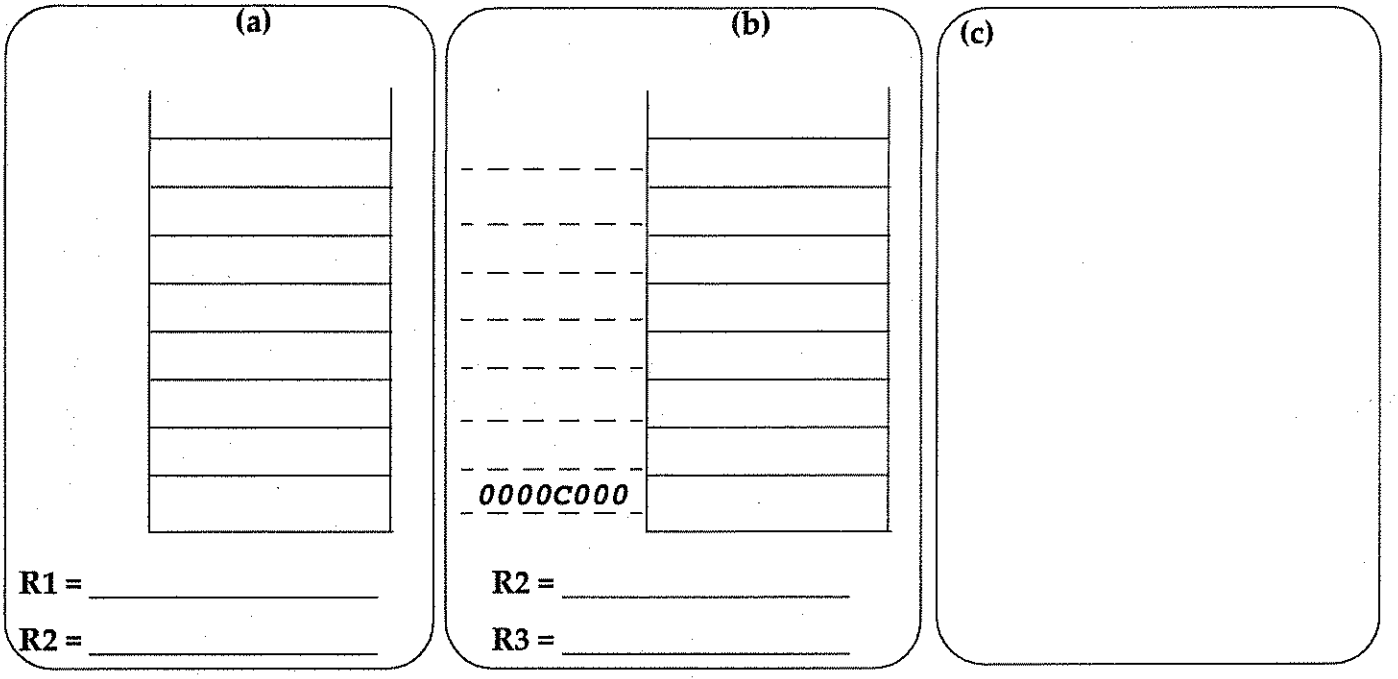
**(a)** [3] The program starts by executing lines [1] - [7]. Assume label "X" is at address "0x0000 C000" and its content is "0x0000 0003"; also label "Y" is at address "0x0000 C004" and its content is "0x0000 000C". Show the contents of the stack and of registers R1 and R2 *after* the execution of [7] and *before* any execution of line [12] using Figure 4 (a).

**(b)** [3] Execution continues through lines [12] - [15]. Show the contents of the stack and of registers R2 and R3 immediately after [15] using Figure 4 (b).

**(c)** [1] Label the lines on the left of the stacks in Figure 4(b) with the correct addresses, given that the initial address of the empty stack is given to you there.

**(d)** [1] Why is the constant "8" used in line [14]? Explain briefly using Figure 4(c).

**FIGURE 4. Tracing Nested Function Calls - part 1**




**(e)** [6] Execution continues. Trace the code and show the contents of the stack immediately after *every time* you happen to execute any of the following lines: [10], [21],[22], [31], [34],[41]. Use the diagrams provided for you in Figures 5 and 6. There may be more than you need. Label carefully each diagram or you will not get full marks. Labels must include: line number, all pertinent pointers (SP, FP), addresses - basically anything that shows your understanding!

**FIGURE 5. Tracing Nested Function Calls - part 2.**

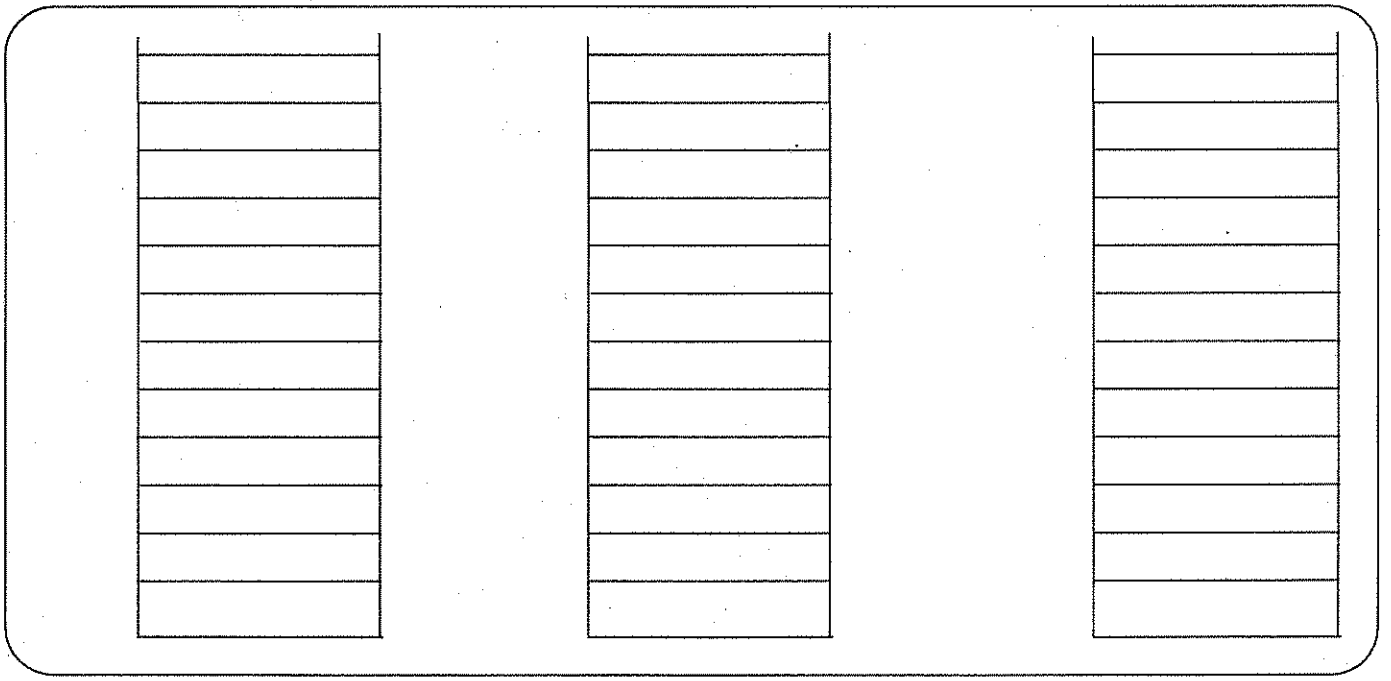
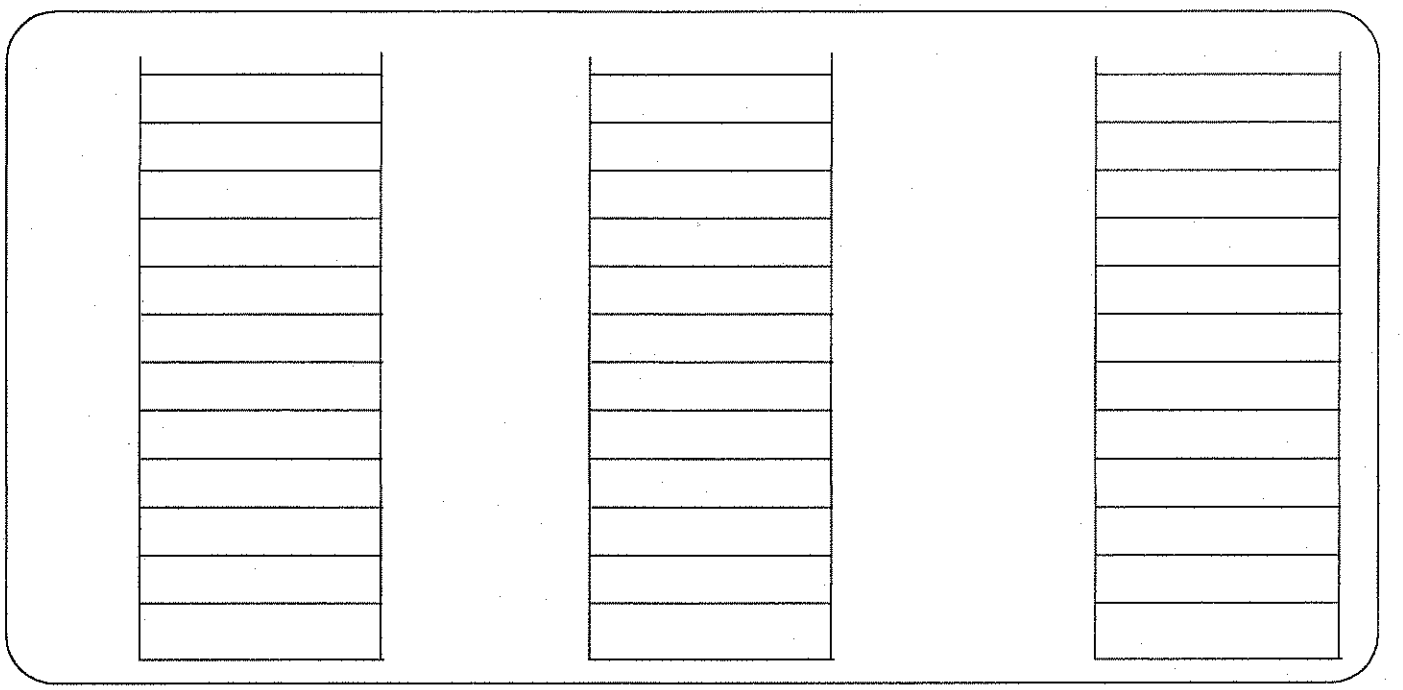


**FIGURE 6. Tracing Nested Function Calls - part 3.**

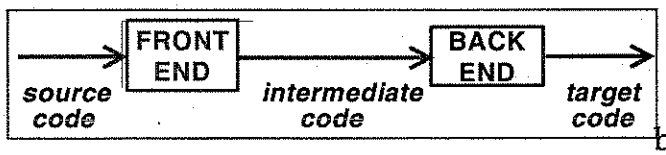


**Question 5. [9]** The design of a system has been started, with a 16-bit address bus and a 16-bit data bus; it is expected to be word-addressable for the memory interface, where a word is defined to be 16 bits (2 bytes). Four peripherals are certainly going to be connected to it: a keyboard for input and a screen for output, each requiring $2^{10}$ bytes of addressing space to communicate with the processor; a controller for the disk drive requiring 1,024 bytes of addressing space; a network communication card which needs $2^9$ words of addressing space. All 4 peripherals are *byte-addressable* in their interfacing with the processor. It is time to purchase memory. What is the maximum size of memory *in bytes* which can be placed into this system? (State the answers using powers of 2 notation as much as possible). Show your work in detail with explanations. You must convince the instructor that you understand the process; a correct number answer only gives you half the marks. To help you, start by answering the following question:

**(a)** [1] what is the total number of addressable locations for this system?

**(b)** [7] Now explain the rest in detail.

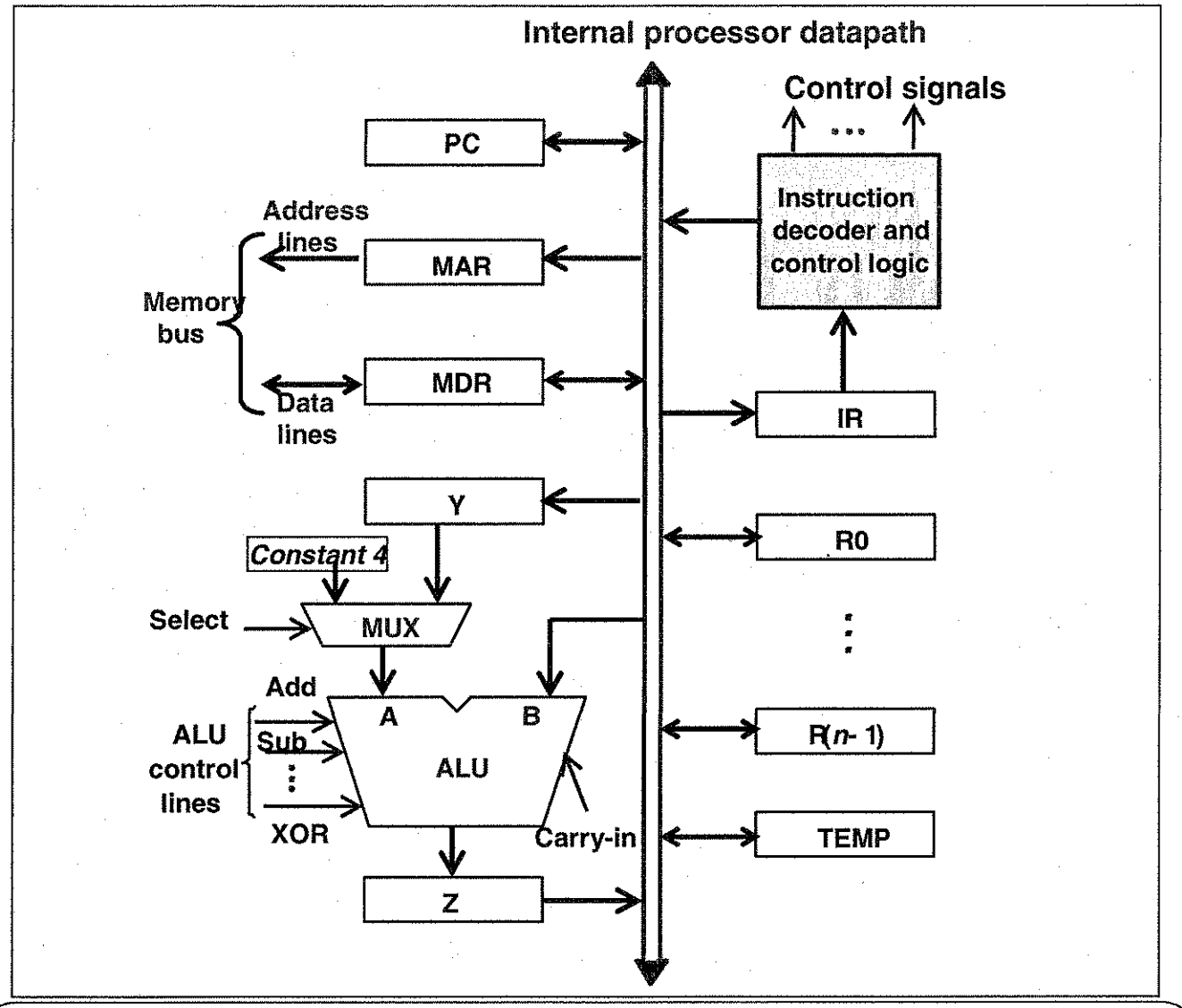
(c) [1] If memory were changed from being *word-addressable* to be *byte-addressable*, there would be considerable cost savings. True or false?" Justify your answer.




**Question 6. [5]** The diagram shows the structure of a compiler decomposed into a "Front End" and a "Back End" with Intermediate Code generated in between. Explain briefly in point form what the units do, how, what the features and dependencies are; in summary, anything which gives a terse yet complete explanation. In an interview situation, you should not take more than 2 minutes; be similarly concise and precise.

**Question 7. [10]** Consider Figure 7 showing the main components of a CPU and its interface to memory (from the lectures). Describe the steps which occur when the ARM instruction "`STR R1,[R3,R4]`" is fetched, decoded and executed in the context of Figure 7. Give precise details of everything that happens, which registers, buses, control lines, etc. are used, how each element is addressed and describe the purpose of each micro-operation. Be brief yet precise.

FIGURE 7. An example of datapath




```
STR  R1,[R3,R4]
```

**Question 8.** [12] You have learned about many aspects of architecture and organization which affect what happens when the processor needs to fetch instructions or data. All items are somewhat related: cache, VM (Virtual Memory), DMA, disks, MMU (Memory Management Unit), pages, Page Table, the TLB (Table Lookaside Buffer) - to name a few. Now summarize what you know by stating what happens step by step in each of the situations below. List which units are involved, why and how, and what the outcome should be. The tables below are to help contain the answers in an easy format. Tick in the appropriate column only if you believe that the items listed there is involved at all in the process. Leave it blank otherwise.

**[ a]: A block of data is requested and it is in the cache**

| Cache | TLB | Page Table | Memory | Disk (VM) | *Explain here:* |
|---|---|---|---|---|---|
| | | | | | |

**[ b]: A block of data is requested and it is in memory**

| Cache | TLB | Page Table | Memory | Disk (VM) | *Explain here:* |
|---|---|---|---|---|---|
| | | | | | |

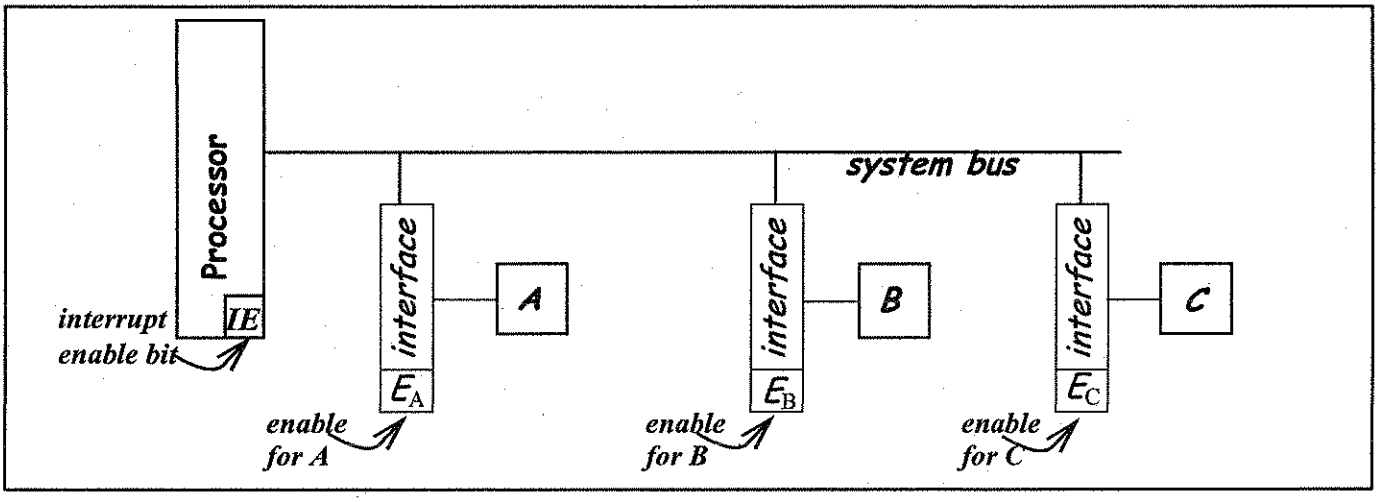
**[ c]: A block of data is requested and it is on disk**

| Cache | TLB | Page Table | Memory | Disk (VM) | *Explain here:* |
|---|---|---|---|---|---|
| | | | | | |

**Question 9.** [2] An addressing mode containing an absolute address for a memory location is not provided in the ARM architecture. Thus an ARM program cannot contain an instruction like "`LDR R2,[0x00C0A4]`" to load the content of memory at address "`0x00C0A4`" into `R2`. Can you explain why the designers of the ARM architecture could not or did not want to provide an addressing mode for absolute memory locations?

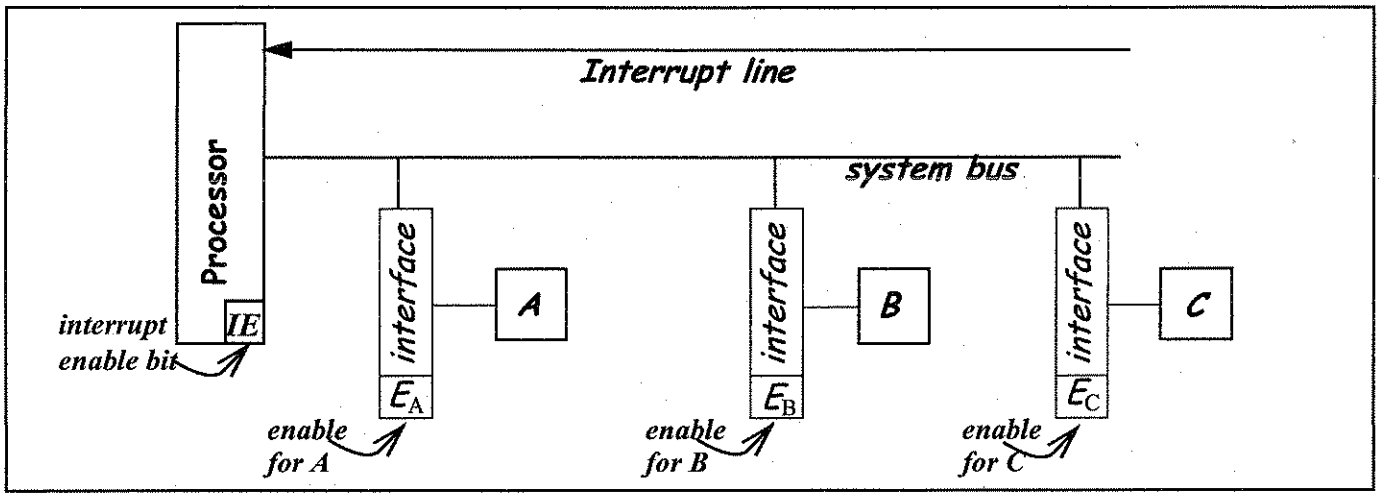
**Question 10.** [8] Three devices, A, B, and C, are directly connected via an interface and the bus to the processor in a system. Transfers for I/O between the processor and the three devices use interrupt control. The processor has an Interrupt Enable bit in its Status Register, shown as "IE" in the figure below. Each device interface also has an "enable" flag, labelled as "$E_A$", "$E_B$" and "$E_C$" as shown in Figure 8.

**FIGURE 8. The initial connections for three peripherals**




Interrupt nesting for devices A and B is not allowed, that is, neither A nor B can interrupt the other or interrupt C. However, interrupt requests from C may be accepted while either A or B is being serviced. You are to explore, explain and test different ways in which this can be accomplished correctly for two different scenarios described below. Specify when and how interrupts are enabled and disabled in each solution, by sending the appropriate signals and setting/resetting the flag bits available. Moreover, for each scenario, show in a table what should happen for each possible test case according to the requirements, and what will happen within your solution, assuming a maximum of only two interrupt requests in a time period.

*Scenario A:* the processor has one interrupt-request line.

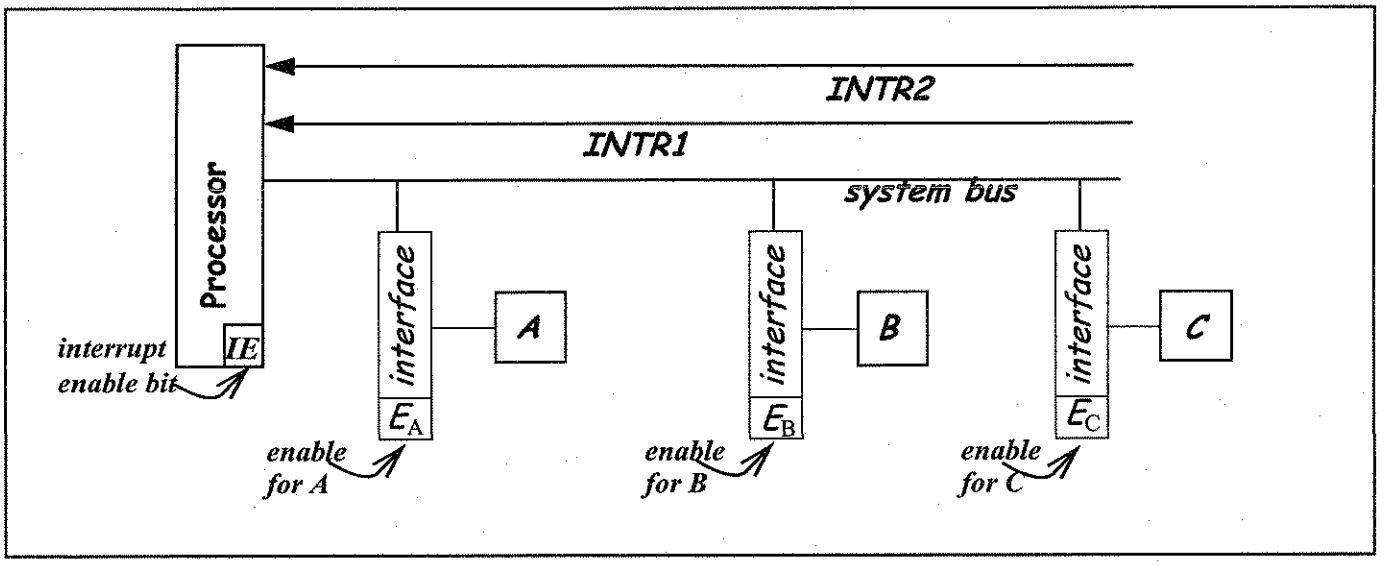



**(a)** [1] Show on the figure how the interfaces and devices should connect to the *newly placed* Interrupt line.

**(b)** [3] State how the overall process will work. State how interrupts are enabled or disabled, which signals are sent and which bits are set and by which component. Be brief and use the table below to explain the various cases. If you believe there should be more cases to be examined, feel free to add them. One sample answer is already given to you, to give you an idea of the expectations.

| | | **What should happen? (circle one or more)** | **Give details of signals, events, reasons.** |
|---|---|---|---|
| **1** | A is being serviced and B sends an interrupt request | *A continues, B ignored forever.*<br>*A continues, B ignored until A is finished.*<br>*A stops, B is done, A restarts.*<br>*A stops, B is done, A does not restart.* | *IE = 1, set by processor, to enable further interrupts.*<br>$E_A$ *= 0, set by A's Interface (request was accepted).*<br>$E_B$ *= 1, set by B's Interface (B is sending a request).*<br>$E_C$ *= 0, set by C's Interface (no request from C).* |
| **2** | A is being serviced and C sends interrupt request | *A continues, C ignored forever.*<br>*A continues, C ignored until A is finished.*<br>*A stops, C is done, A restarts.*<br>*A stops, C is done, A does not restart.* | *IE =*<br>$E_A$ =<br>$E_B$ =<br>$E_C$ = |
| **3** | B is being serviced and A sends interrupt request | *B continues, A ignored forever.*<br>*B continues, A ignored until B is finished.*<br>*B stops, A is done, B restarts.*<br>*B stops, A is done, B does not restart.* | *IE =*<br>$E_A$ =<br>$E_B$ =<br>$E_C$ = |
| **4** | C is being serviced and A sends interrupt request | *C continues, A ignored forever.*<br>*C continues, A ignored until C is finished.*<br>*C stops, A is done, C restarts.*<br>*C stops, A is done, C does not restart.* | *IE =*<br>$E_A$ =<br>$E_B$ =<br>$E_C$ = |

*Scenario B:* two interrupt-request lines, INTR1 and INTR2, are available, with INTR1 having higher priority.



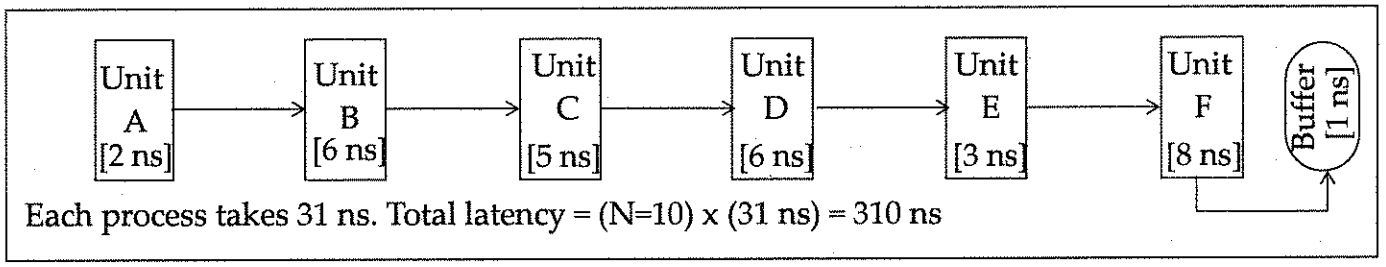

**(c)** [1] Show on the figure how the interfaces and devices should connect to the newly placed INTR1 and INTR2.

**(d)** [3] State how the overall process will work. State how interrupts are enabled or disabled, which signals are sent and which bits are set and by which component. Be brief and use the table below to explain the various cases. If you believe there should be more cases to be examined, feel free to add them. One sample answer is already given to you, to give you an idea of the expectations.

| | | **What should happen? (circle one or more)** | **Give details of signals, events, reasons.** |
|---|---|---|---|
| **1** | A is being serviced and B sends interrupt request | *A continues, B ignored forever.*<br>*A continues, B ignored until A is finished.*<br>*A stops, B is done, A restarts.*<br>*A stops, B is done, A does not restart.* | *IE = 1, set by processor, to enable further interrupts.*<br>$E_A$ *= 0, set by A's Interface (request was accepted).*<br>$E_B$ *= 1, set by B's Interface (B is sending a request).*<br>$E_C$ *= 0, set by C's Interface (no request from C).* |
| **2** | B is being serviced and A sends interrupt request | *B continues, A ignored forever.*<br>*B continues, A ignored until B is finished.*<br>*B stops, A is done, B restarts.*<br>*B stops, A is done, B does not restart.* | *IE =*<br>$E_A$ =<br>$E_B$ =<br>$E_C$ = |
| **3** | B is being serviced and C sends interrupt request | *B continues, C ignored forever.*<br>*B continues, C ignored until B is finished.*<br>*B stops, C is done, B restarts.*<br>*B stops, C is done, B does not restart.* | *IE =*<br>$E_A$ =<br>$E_B$ =<br>$E_C$ = |
| **4** | C is being serviced and B sends interrupt request | *C continues, B ignored forever.*<br>*C continues, B ignored until C is finished.*<br>*C stops, B is done, C restarts.*<br>*C stops, B is done, C does not restart.* | *IE =*<br>$E_A$ =<br>$E_B$ =<br>$E_C$ = |

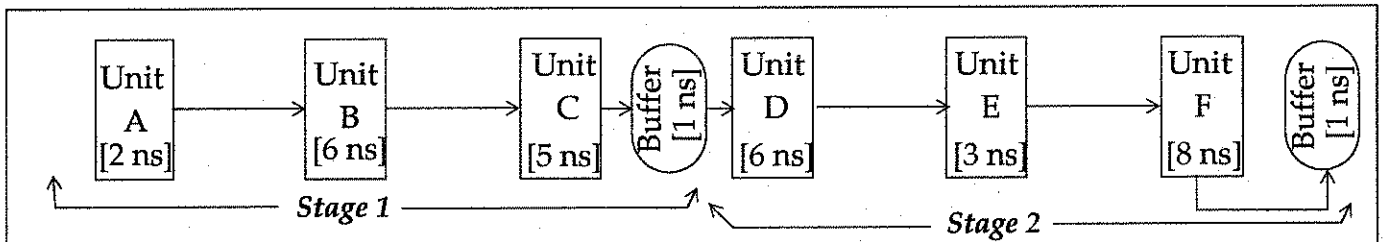
**Question 11.** [7] +[2] **(e)** A linear data path being designed consists of parts with the following latencies, in order from left to right, as shown in Figure 9 : 2 ns, 6 ns, 5 ns, 6 ns, 3 ns, 8 ns. These parts are indivisible, so that any pipeline buffers (storages/delay) must be inserted only between consecutive components. Insertion of such buffers introduces a stage overhead of 1 ns. In Figure 9 the latency for N processes, with N = 10 (for ease of computation, however unrealistic), without any pipelining and only one buffer at the output is shown to be 310 ns. The introduction of one more buffer in the middle, making it into a 2-stage pipeline is shown in Figure10 to have a latency of 194 ns.

**FIGURE 9. Serial datapath with no pipelining**




Each process takes 31 ns. Total latency = (N=10) x (31 ns) = 310 ns

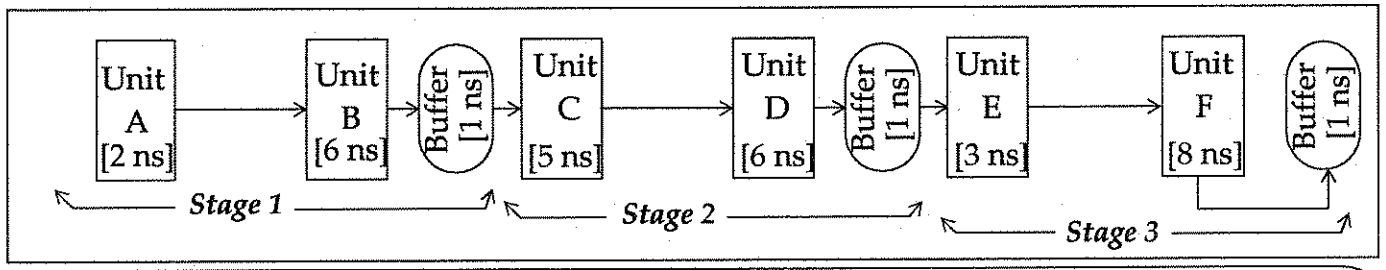
**FIGURE 10. Datapath with 2-stage pipelining**




The first process takes 32 ns. The next 9 processes take (6 + 3 + 8) ns + 1ns = 18 ns.
Total latency = 32 ns + (N=9) x (18 ns) = 194 ns

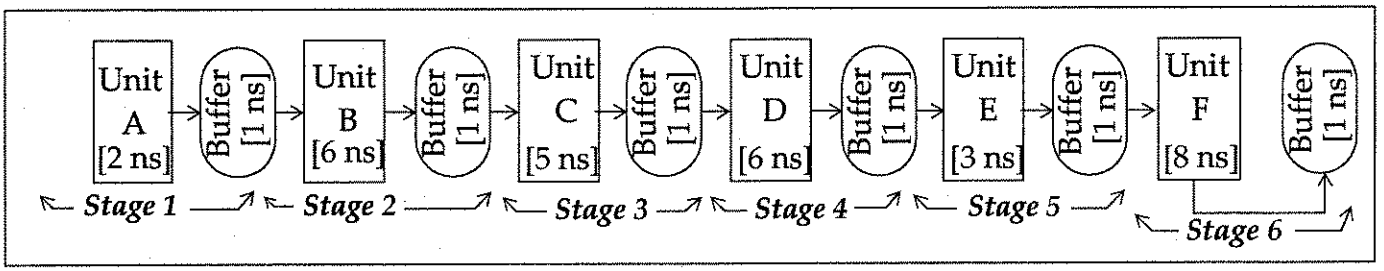
**(a)** [3] Following the same patterns of calculation, determine the latency when changing the datapath to be a 3-stage pipeline with buffers inserted as shown in Figure 11.

**FIGURE 11. Datapath with 3-stage pipelining**



**(b)** [6] Following the same patterns of calculation, determine the latency when changing the datapath to be a 6-stage pipeline with buffers inserted in between every pair of units as shown in Figure 12 .

**FIGURE 12. Datapath with 6-stage pipelining**




**(c)** [1] Draw a conclusion about an optimal design among the ones explored so far.

**(d) BONUS** [2] Redo the calculations using N = 1,000 for the number of processes. Do you reach the same conclusions?

**Question 12. [16] (a)** A memory has 256K words (= $2^{18}$ words), addressable by 32-bit addresses, to be viewed as 16K blocks (= $2^{14}$ blocks) of 16 words each, ($2^{14}$ blocks x $2^4$ words =$2^{18}$ words). Draw a schematic picture of this organizational view on the left side below. Be precise, i.e. label and number the blocks, etc. [2 marks]

**(b)** You are constructing a cache and considering three mapping strategies. Your only constraint is the size of the cache, which must be 128 blocks (lines) of 16 words each. Explore at least two strategies you know, explain each both in words and with a diagram [4 marks]. For each of the two strategies, show precisly the cache block (line) address where memory blocks are mapped into the cache [10 marks], for the following memory blocks numbered: [0], [ $2^{14}$ -1], [128], [ $2^{12}$ ], [3,256] .

**(c)** If you wish to repeat the answer above for a third cache mapping strategy, you may receive up to 8 bonus marks for a complete and correct answer.

**THE END**