UNIVERSITY OF VICTORIA

EXAMINATIONS SPRING 2008

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	*4*		8	*6*	
2	*3*		9	*12*	
3	*5*		10	*13*	
4	*6*		11	*10*	
5	*5*		12	*10*	
6	*8*		13	*10*	
7	*8*		**TOTAL**	**100**	

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 13 pages including this cover page plus 5 pages of Appendix.
3. No aids are permitted. However, an Appendix 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. [4]

(a) The range of decimal values that can be represented as *unsigned* 16-bit integers is:

(_____ , _____)

(b) The range of decimal values that can be represented as a *signed* 8-bit 2's complement integers is:

(_____ _____ , _____ _____)

Question 2. [3] Suppose that execution time for a program is directly proportional to instruction access time. Access time to an instruction is *2 ns* from the cache and *20 ns* from memory. The probability of a cache hit is 96%. In the case of a cache miss, the instruction is fetched from main memory and copied into the cache, and then a second access must take place to copy the instruction from the cache (this time it will be a hit).

(a) [1] Compute the execution time of a program with 100 instructions without the cache.

_____

(b) [1] Compute the execution time of a program with 100 instructions with the cache.

_____

(c) [1] If the cache size is doubled, the probability of not finding an instruction is cut in half. Compute the execution time of a program with 100 instructions with the larger cache.

_____

Question 3. [5] Figure 1.4 in your textbook is reproduced below in Figure 1. It shows the execution portion for a program "P" which has an "*input*" phase from t_0 to t_1, a "*compute*" phase from t_1 to t_4, and an "*output*" phase from t_4 to t_5. During the *input* phase, an OS routine loads the program from disk to memory and then execution control is passed to the application. From t_2 to t_3 the OS transfers data from disk, and from t_4 to t_5 it prints results. If several applications need to be processed, resources can be better allocated as, for example, the OS can be loading the *input* phase of program "P+1" while the *output* phase is taking place for program "P". Furthermore during the *input* phase of program "P" it could have been processing the *output* phase of program "P-1". This pattern of concurrent execution is often called "*multiprogramming*" and it is based on overlapping the *input* and *output* phases of a collection of programs to reduce the total time needed to execute them.

Let each of the six OS routine execution intervals be 1 unit of time, with each of the two disk operations requiring 3 units, printing requiring 3 units, and each of the two program compute intervals requiring 2 units of time (as labelled in figure 1). Figure 2 shows a diagram of the overlapping of three successive programs P1, P2 and P3. You are asked to compute the ratio of best overlapped time to non overlapped time for a sequence of programs. It may help to follow the steps below.

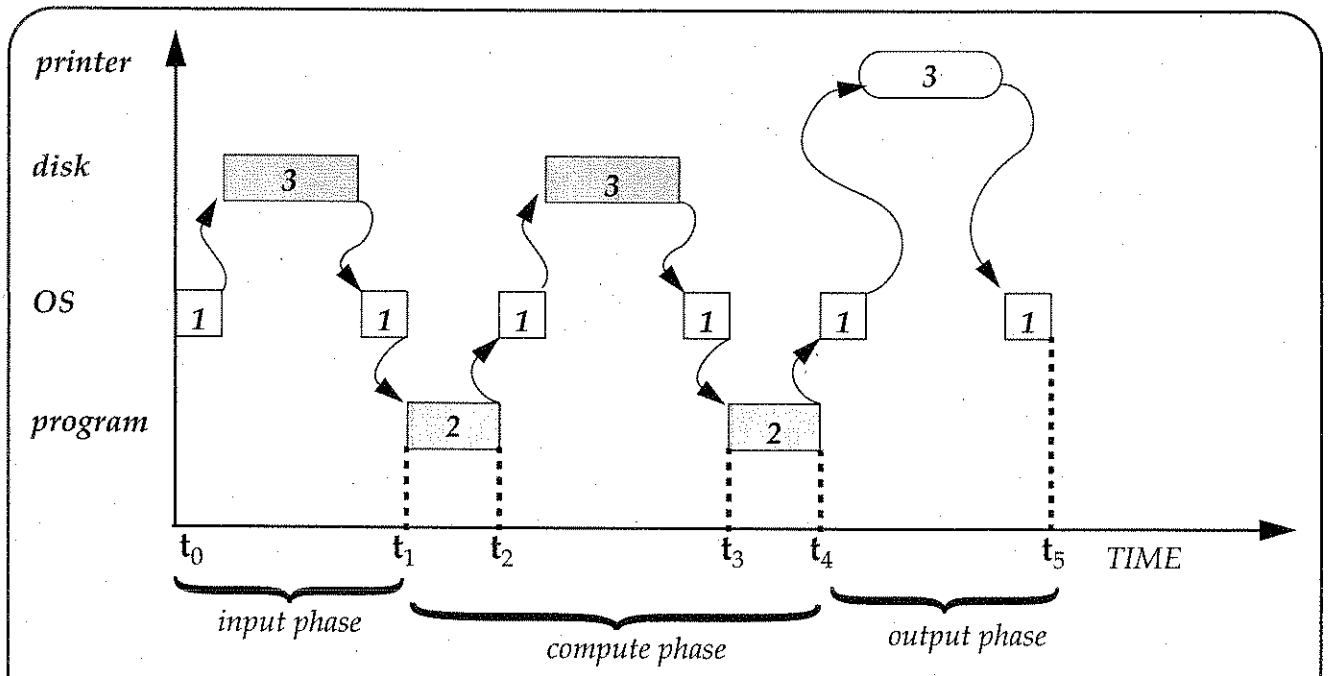


FIGURE 1. Sharing the processor time between OS routines and user program

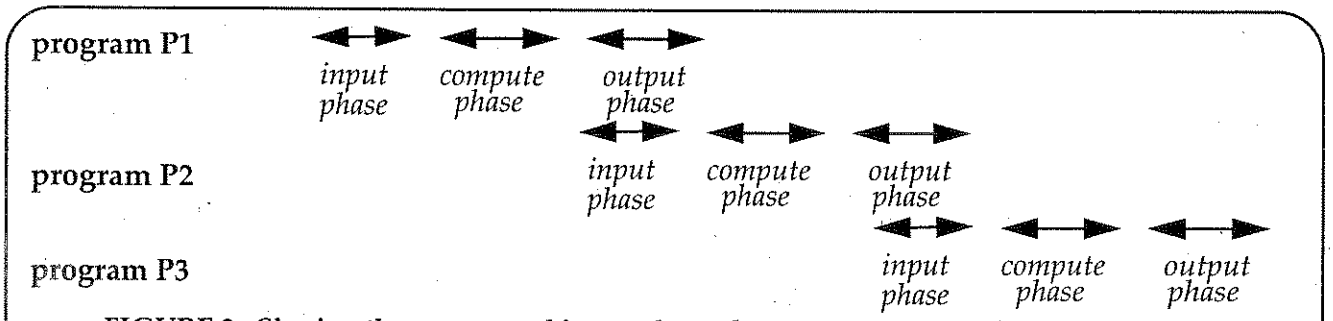


FIGURE 2. Sharing the output and input phases between two successive programs

(a) [1] How many units of time in total are required for each single program, assuming they are composed of the same phases? Draw a diagram showing the three phases (input, computing, output) and the intervals within each, then compute the total.

(b) **[2]** In Figure 2 three programs are shown with overlapping. How many additional time units are required for the second program P2 to finish execution? How many additional time units are required for the third program P3 to finish execution?

(c) **[1]** You have calculated the time between completion of successive programs in (b) with overlapping and the time to completion without overlapping in (a). Now compute the ratio of best overlapped time to non overlapped time for a sequence of programs as asked initially.

(d) **[1]** The overlap considered so far has been only between the *output* phase of program P with the *input* phase of program P+1. Now ignore the short time needed for OS routines and assume that each program has an equal balance among the *input*, *compute* and *output* phases, that is, each phase takes 1/3 of the total elapsed time to execution. If the three activities of printing, processing and disk access can be seen as independent, such that all three could take place at the same time in the steady state, what could the theoretical ratio be between non overlapped and overlapped time?

Question 4. **[6]** Program execution time, T, is defined in section 1.6.2 of the textbook as:

$$T = \frac{N \times S}{R} \qquad \text{where}$$

- T is the total elapsed time,
- N is the number of machine language instructions used during the execution (not necessarily the number of machine instructions in the object code),
- S is the number of basic steps needed to execute one machine instruction (where each basic step is assumed to take 1 clock cycle),
- R is the clock rate.

You are asked to examine T for a certain high-level language program. The program can be run on a RISC or a CISC computer. Both computers use pipelined instruction execution and have the same clock rate R. However pipelining in the RISC machine is more effective than in the CISC machine, such that the effective value of Sr for the RISC machine is 1.2, but it is $Sc = 1.5$ for the CISC machine. You are told that the program will have the same total execution time T on both machines if Nc, the number of machine language instructions for CISC, is 80% or 4/5 of Nr, the number of machine lan-

guage instructions for RISC. Is this correct? Show your work and your reasoning in order to justify your answer in various steps, as below..

(a) [1] *Elapsed time in RISC:* Tr =

(b) [1] *Elapsed time in CISC:* Tc =

(c) [4] *Given* $Sc = 1.5$ *and* $Sr = 1.2$, *the conjecture is that* $Nc = 4/5\ Nr$ *will give* $Tr = Tc$. *Why?*

Question 5. [5] You are involved in the design of a system which needs to have a 16-bit address bus and a 16-bit data bus; it is expected to be byte-addressable for everything and a word is defined to be 16 bits (2 bytes). There are both peripherals and memory units to be connected to it and it is expected that the whole address space will be used.

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

_____

(b) [1] The idea is to use about 3/4 of the addressable space for the memory requirements of RAM and ROM, and 1/4 of the addressable space for the peripherals. What will be the number of addresses available for memory?

_____

(c) [1]] What will the number of available addresses be for peripherals?

_____

(d) [1] The addressable space for peripherals is further subdivided into 1/8 of it for input components, 1/8 of it for output components and the rest (6/8) for disk addressing. The chosen +disk drives each require 4Kb of address space. How many can one fit into the system?

_____

(e) [1] The ROM has already been bought and its size is 16Kb. The cheapest RAM chips contain 8K words. How many should you buy to build one system?

_____

Question 6. **[8]** When talking about a compiler the following words are used: front end, back end. Gives a terse yet complete explanation of what is meant by these terms in the context of how a compiler works. In an interview situation, you should not take more than 1 minute orally.

Question 7. **[8]** Describe at least one cache mapping strategy in detail and give an example.

Question 8. **[6]** Assume that the processor has a pipeline with 4 stages, namely for "Fetching instructions", for "Decoding instructions and fetching operands", for "Executing operations" and for "Writing results" with 3 interstage buffers. The two instructions below are found in that order in a segment of code to be executed with this processor and it is possible that they may cause a delay in the normal performance.

```
MUL   R1,R2,R3        @ R1 = R2*R3
STR   R4,[R5,R1]      @ store R4 to memory at address (R5)+(R1)
```

(a) **[2]** Why might these instructions cause a delay?

(b) **[1]** What is this situation called?

(c) **[2]** Draw a timing diagram for a general pipeline of this type which shows this situation.

(d) **[1]** What solutions are possible either in software or in hardware to avoid this kind of delay? State one.

Question 9. **[12]** Describe the steps which occur when the ARM instruction:

```
LDR R4,[R2,R3, LSL #2]
```

is fetched, decoded and executed in the context of the example structure of the internal datapath of a processor of Figure 3. 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.

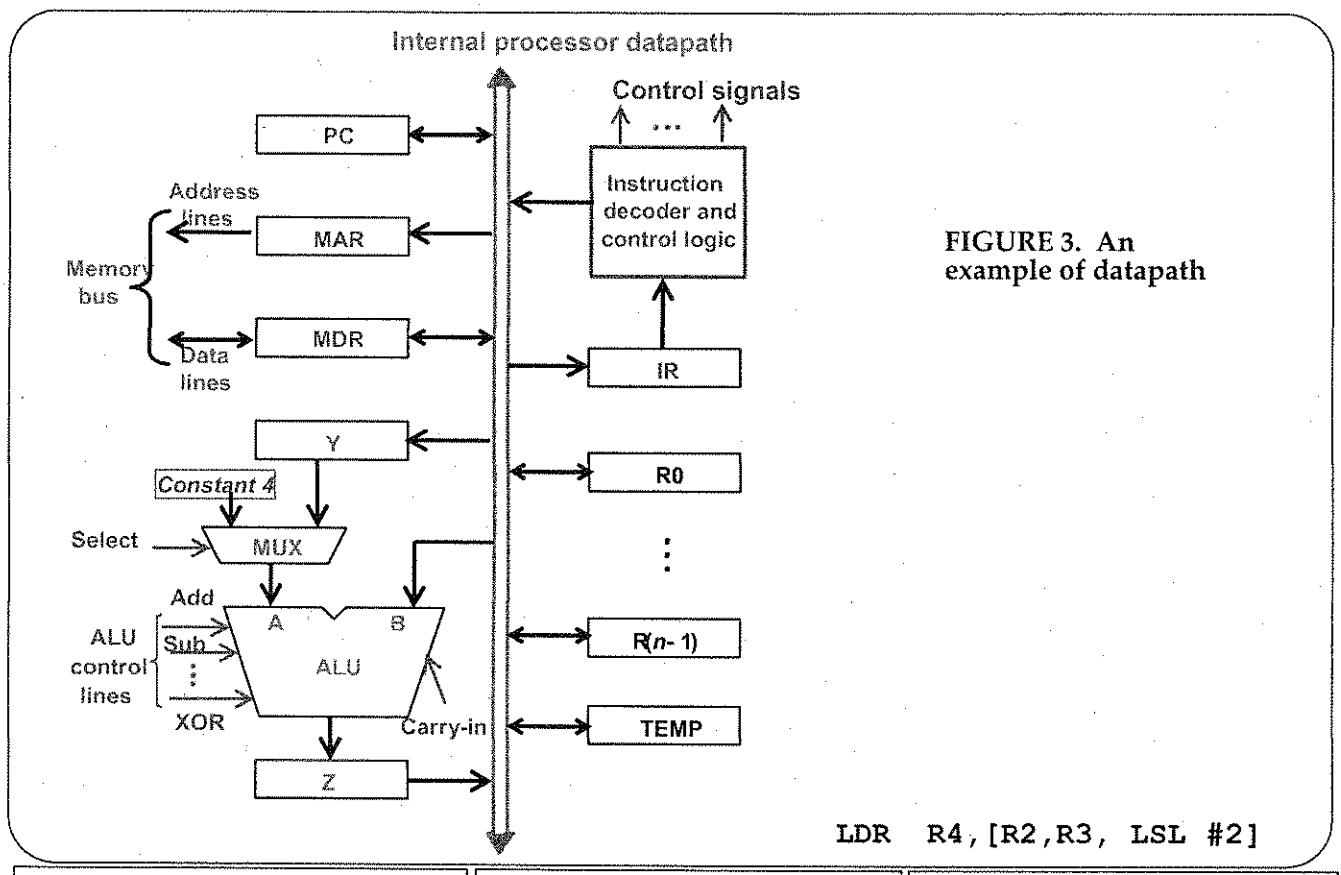


FIGURE 3. An example of datapath

```
LDR  R4,[R2,R3, LSL #2]
```

FETCH	DECODE	EXECUTE

Question 10. [13] Assume you have a system which contains almost all the elements which have been discussed in the course, namely Virtual Memory, Cache, Pipelining, MMU (Memory Management Unit), TLB (Translation Lookaside Buffer), Page Table, DMA (Direct Memory Access). You need to explain how all these function together when the processor needs to read data and produces a virtual address which needs to be translated into a physical address. Answer the following questions:

(a) [1] Where is the Page Table stored?

_____

(b) [1] Where is the TLB stored?

_____

(c) [3] Describe the steps which take place when there is a cache hit (best scenario).

(d) [3] Describe the steps which take place when the needed data is only on disk (worst scenario).

(e) [3] Describe the steps which take place when the needed data is in memory, but not in cache.

(f) [2] You may or may not have mentioned the DMA in the steps above. If you included the DMA, give a definition of what it is and what function it had in the previous 3 scenarios. If you did not include the DMA, state why not and still give a definition of what it is.

Question 11. [10] The most difficult coding segment in one of your assignments was the function "void Delay(N:R0)" which causes a delay of N milliseconds. The first difficulty was translating from a 32-bit timer in ARMSim to a 15-bit timer on the Embest board. The second difficulty was having to take care of the rolling around of the timer. You must now show that you truly have understood the details of the code by rewriting perfect code following the strict specifications here.

(a) [1] A call to the `swi 0x6d` instruction returns in R0 the current number of ticks (milliseconds) as a 32-bit quantity. Show ARM code to transform this *32-bit value in R0* into a *15-bit value in R1*.

(b) [4] The 15-bit timer has a range starting at time "`0x0000`" and ending at time "`0x7FFF`" = "$32,767_{10}$". After reaching the value "`0x7FFF`=$32,767_{10}$" it rolls over to "`0x0000`". The delay function calls the `swi 0x6d` instruction two times, saving the results in T1 and T2. It then computes the difference between T1 and T2 and compares it to the desired delay value passed as parameter. In its simplest form, the pseudo code could be as follows:

DELAY= desired delay (from input parameter R0)
T1 = get time with swi 0x6d
T1 = adjusted time to 15-bit
Repeat: {
    T2 = get time with swi 0x6d
    T2 = adjusted time to 15-bit
    TIME= T2-T1
    If TIME < DELAY go to Repeat
}

There are cases to be considered because of the rollover. If both T1 and T2 fall within the same period of the timer, that is 0 < T1 < T2 < 32,767, with T1 and T2 as *unsigned* values, then all is well (e.g. T1=1,000 and T2=15,000). But if T1 has a value close to 32,767 and T2 has a value which has

rolled over after 0, then the case becomes 0 < T2 < T1 < 32,767, with T1 and T2 as *unsigned* values (e.g. T1=30,000 and T2=2,000). Rewrite the pseudo code to take care of this situation properly.

(c) [5] Write the complete, precise, documented and correct *code in C*, based on your pseudo code from above, for the function "`void Delay(int N)`". Assume you have already a function "`TIME=SWI06D()`" which returns the current number of milliseconds as a 32-bit value.

Question 12. [10] You used the `swi 0x203` instruction to check whether one of the 16 blue buttons had been pressed (see the Appendix for documentation). Somebody tells you that there might be some errors in the documentation and you should test the functionality before using the instruction. For example, you are told that if one pushes the button "13" it is not clear if the result in R0 is the value "0x00002000" corresponding to the bit in position 13 having been set, or the value "0x0000000D" corresponding to the number "13" itself, or the value "0x00040000" corresponding to the bit in position 13 having been set when bits are labelled from left to right. You have now the job of testing the code against the specifications in the documentation. Write the few lines of ARM code needed to complete the code segment below to accomplish the task of checking how the `swi 0x203` instruction works exactly. Check, after pressing each button, what result appears in register R0 and compare it against the expected value in the documentation. Document your code.

```
. . . . . . . .
MLOOP:     LDR  R0,=PROMPTNUM        @prompt tester to enter a number
           SWI  0x07              @and save it in R2. Blue button pressed
           MOV  R2,R0             @should be the one numbered as in R2



LOOP:      SWI  0x203             @keep checking for a blue button
           BEQ  LOOP
                                  @blue button with number=R2 should
                                  @have been pressed - now check here
                                  @that the pattern in R0 corresponds
                                  @to the expectation of the button
                                  @number stated in R2
                                  @for example, if R2 = 5, check that
                                  @the bit in position 5 (counting from
                                  @right) in R0 is indeed set to 1 and
                                  @all other bits are equal to 0







ERROR:     LDR  R0,=NOTOKAY
           SWI  0x02              @print error message
           BAL  MLOOP             @go test next button
NOERROR:   LDR  R0,=OKAY
           SWI  0x02              @print ok message
           BAL  MLOOP             @go test next button
. . . . . . . . . .
. . . . . . . . . .
     .data
PROMPTNUM:.asciz    "enter a number 0-16 and then press the
                    corresponding blue button\n"
NOTOKAY:  .asciz    "problem with this button\n"
OKAY:     .asciz    "no problem with this button\n"
```

Question 13. [10] In the assignments you used a segment of code for the 8-segment display and you were expected to understand it fully (see the Appendix for the SWI instructions). Now pretend that you are teaching a new student and you must explain extremely clearly exactly what happens in this code. Explain for each line [3] through [5] what happens and exactly how the instruction works (that is, be precise about the addressing mode and its semantics). The code is shown in Figure 4, while the data and the EQU declarations are in Figure 5. To make the explanation really clear, use an example of how the code works, line by line, when it is called with parameters R0=5 and R1=1. Every register must be fully explained in its content and its meaning.

```
@ *** Display8Segment (Number:R0; Point:R1) ***
@ Displays the number 0-9 in R0 on the
@ LED 8-segment display
@ If R1 = 1, the point is also shown
[1]  Display8Segment:    stmfd     sp!,{r0-r2,lr}
[2]        ldr       r2,=Digits
[3]        ldr       r0,[r2,r0,lsl#2]
[4]        tst       r1,#0x01
[5]        orrne     r0,r0,#SEG_P
[6]        swi       0x200
[7]        ldmfd     sp!,{r0-r2,pc}
```

FIGURE 4. The subroutine for the 8-segment display

```
Digits:
    .word  SEG_A|SEG_B|SEG_C|SEG_D|SEG_E|SEG_G      @0
    .word  SEG_B|SEG_C                              @1
    .word  SEG_A|SEG_B|SEG_F|SEG_E|SEG_D            @2
    .word  SEG_A|SEG_B|SEG_F|SEG_C|SEG_D            @3
    .word  SEG_G|SEG_F|SEG_B|SEG_C                  @4
    .word  SEG_A|SEG_G|SEG_F|SEG_C|SEG_D            @5
    .word  SEG_A|SEG_G|SEG_F|SEG_E|SEG_D|SEG_C      @6
    .word  SEG_A|SEG_B|SEG_C                        @7
    .word  SEG_A|SEG_B|SEG_C|SEG_D|SEG_E|SEG_F|SEG_G  @8
    .word  SEG_A|SEG_B|SEG_F|SEG_G|SEG_C            @9
       .word  0                  @Blank display
```

```
.equ   SEG_A,0x80
.equ   SEG_B,0x40
.equ   SEG_C,0x20
.equ   SEG_D,0x08
.equ   SEG_E,0x04
.equ   SEG_F,0x02
.equ   SEG_G,0x01
.equ   SEG_P,0x10
```

FIGURE 5. Data and EQU for 8-segment display

```
     Display8Segment:
[1]  stmfd     sp!,{r0-r2,lr}
[2]  ldr  r2,=Digits
[3]  ldr  r0,[r2,r0,lsl#2]



[4]  tst  r1,#0x01



[5]  orrne     r0,r0,#SEG_P



[6]  swi  0x200
[7]  ldmfd     sp!,{r0-r2,pc}
```

EXPLAIN HERE

Appendix to Final Exam for CSC 230 - Spring 2008

1. Basic SWI Operations.

Table 1: SWI operations (0x00 - 0xFF)

Opcode	Description and Action	Inputs	Outputs
`swi 0x00`	Display Character on Console	r0: the character	
`swi 0x02`	Display String on Console	r0: address of a null terminated ASCII string	
`swi 0x07`	Prompt User for an Integer	r0: address of a null terminated ASCII string	r0: the integer
`swi 0x6d`	Get the current time (ticks)		r0: the number of ticks (milliseconds)

Display Character on Screen: swi 0x00

Displays one character in the output window.

```
    mov     r0,#'X
    swi     0x00
```

Display String on Screen: swi 0x02

Displays a string in the output window.

```
    ldr     r0,=MyString
    swi     0x02
    ...
MyString: .asciz "Hello There\n"
```

Prompt User for an Integer: swi 0x07

Creates a pop-up window which displays the supplied message and waits until the user types a number into that window.

```
    ldr     r0,=Prompt
    swi     0x07
    ldr     r1,=Number
    str     r0,[r1]
    ...
Number:     .word  0
Prompt:     .asciz "Enter a number"
```

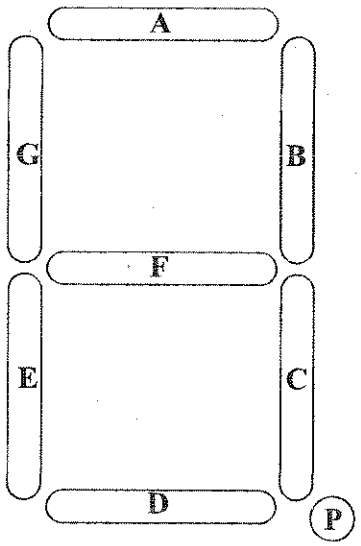
Table 2: SWI operations > 0xFF

Opcode	Description and Action	Inputs	Outputs
`swi 0x200`	Set the 8-Segment Display to light up.	r0: the 8-segment Pattern (see below in Figure 1)	The appropriate segments light up to display a number or a character
`swi 0x203`	Check if one of the Blue Buttons has been pressed.	None (see below in Figure 2)	r0 = the Blue Button Pattern (see below in Figure 2).

Set the 8-Segment Display to light up: swi 0x200

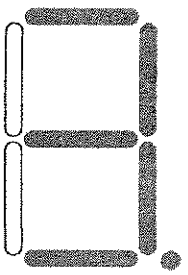
The appropriate segments light up to display a number or a character. The pattern of segments to be lit up is assigned to register `r0` before the call to `swi 0x200`. Figure 1 shows the arrangements of segments.

Figure 1: The Pattern for the 8-Segment Display



Display byte values	
A	0x80
B	0x40
C	0x20
P	0x10
D	0x08
E	0x04
F	0x02
G	0x01

Example: number "3"



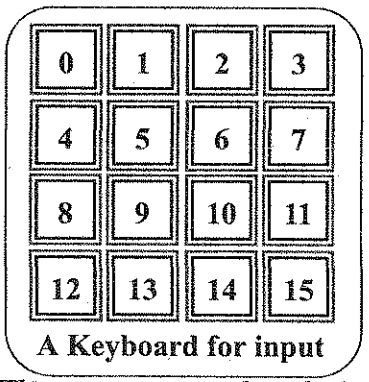
For example, the number 3 plus the right hand dot would have a pattern value computed as the logical OR of the values of the segments "A,B,C,D,F,P" to form the integer: `0x80 | 0x40 | 0x20 | 0x08 | 0x02 | 0x10 = 0xFA`, to be assigned to r0.

Check if one of the Blue Buttons has been pressed: swi 0x203

After the call with `swi 0x203`, test the content of `r0`. The number in `r0` corresponds to the position of the blue button as depicted in Figure 2. For example, if `r0=2` then the blue button in position 2 was pressed.

```
swi     0x203
cmp     r0,#1
cmp     r0,#2
cmp     r0,#3
. . . . . .
```

Figure 2: The Pattern for the Blue Buttons



The keypad with 16 blue buttons as depicted in the board view.

They are arranged such that each button has a corresponding bit position in the 16-bit lower portion of a word in a register. The *"number labels"* placed in the figure, which do not appear on the real keypad, also represent the corresponding bit position, as in:

POSITION	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ON KEYPAD	↕	↕	↕	↕	↕	↕	↕	↕	↕	↕	↕	↕	↕	↕	↕	↕
BLUE BUTTON:	*15*	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*

When a button is pressed, the corresponding bit is set. For example, when the button in position "1" is pressed in the cellular phone program, the `swi 0x203` instruction returns `r0 = 0x02`, that is,

`r0 = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0`

in binary, where the bit in position "1" has been set.

Operand 2	
Immediate value	#<immed_8>
Logical shift left immediate	Rm, LSL #<immed_5>
Logical shift right immediate	Rm, LSR #<immed_5>
Arithmetic shift right immediate	Rm, ASR #<immed_5>
Rotate right immediate	Rm, ROR #<immed_5>
Register	[Rm]
Rotate right extended	Rm, RRX
Logical shift left register	Rm, LSL Rs
Logical shift right register	Rm, LSR Rs
Arithmetic shift right register	Rm, ASR Rs
Rotate right register	Rm, ROR Rs

Condition Field	
EQ	Equal
NE	Not equal
CS	Carry Set
CC	Carry clear
MI	Negative
PL	Positive or zero
VS	Overflow
VC	No overflow
HI	Unsigned higher
LS	Unsigned lower or same
GE	Signed greater or equal
LT	Signed less than
GT	Signed greater than
LE	Signed less than or equal
AL	Always

Dec	Bin	Hex
0	00000000	00
1	00000001	01
2	00000010	02
3	00000011	03
4	00000100	04
5	00000101	05
6	00000110	06
7	00000111	07
8	00001000	08
9	00001001	09
10	00001010	0A
11	00001011	0B
12	00001100	0C
13	00001101	0D
14	00001110	0E
15	00001111	0F

Addressing Mode 4 - Multiple Data Transfer			
Block load		**Stack pop**	
IA	Increment After	FD	Full Descending
IB	Increment Before	ED	Empty Descending
DA	Decrement After	FA	Full Ascending
DB	Decrement Before	EA	Empty Ascending
Block store		**Stack push**	
IA	Increment After	EA	Empty Ascending
IB	Increment Before	FA	Full Ascending
DA	Decrement After	ED	Empty Descending
DB	Decrement Before	FD	Full Descending

D	BIN	H	D	BIN	H	D	BIN	H	D	BIN	H
0	00000000	00	4	00000100	04	8	00001000	08	12	00001100	0C
1	00000001	01	5	00000101	05	9	00001001	09	13	00001101	0D
2	00000010	02	6	00000110	06	10	00001010	0A	14	00001110	0E
3	00000011	03	7	00000111	07	11	00001011	0B	15	00001111	0F

Operation	Assembler	Action
Move	MOV{S} Rd, <Oprnd2>	Rd := Oprnd2 {CPSR}
	MVN{S} Rd, <Oprnd2>	Rd := NOT Oprnd2 {CPSR}
Arithmetic	ADD{S} Rd, Rn, <Oprnd2>	Rd := Rn + Oprnd2 {CPSR}
	ADC{S} Rd, Rn, <Oprnd2>	Rd := Rn + Oprnd2 + Carry {CPSR}
	SUB{S} Rd, Rn, <Oprnd2>	Rd := Rn - Oprnd2 {CPSR}
	SBC{S} Rd, Rn, <Oprnd2>	Rd := Rn + Oprnd2 + Carry {CPSR}
	RSB{S} Rd, Rn, <Oprnd2>	Rd := Oprnd2 - Rn {CPSR}
	RSC{S} Rd, Rn, <Oprnd2>	Rd := Oprnd2 - Rn - NOTCarry {CPSR}
	MUL{S} Rd, Rm, Rs	Rd := Rm * Rs {CPSR}
	MLA{S} Rd, Rm, Rs, Rn	Rd := Rm * Rs + Rn {CPSR}
	CLZ Rd, Rm	Rd := # leading zero in Rm
Logical	AND{S} Rd, Rn, <Oprnd2>	Rd := Rn AND Oprnd2 {CPSR}
	EOR{S} Rd, Rn, <Oprnd2>	Rd := Rn EXOR Oprnd2 {CPSR}
	ORR{S} Rd, Rn, <Oprnd2>	Rd := Rn OR Oprnd2 {CPSR}
	TST Rn, <Oprnd2>	Update CPSR on Rn AND Oprnd2
	TEQ Rn, <Oprnd2>	Update CPSR on Rn EOR Oprnd2
	BIC{S} Rd, Rn, <Oprnd2>	Rd := Rn AND NOT Oprnd2 {CPSR}
	NOP	R0 := R0
Compare	CMP Rd, <Oprnd2>	Update CPSR on Rn - Oprnd2
Branch	B{cond} label	R15 := label
	BL{cond} label	R14 := R14-4; R15 := label
Swap	SWP Rd, Rm	temp := Rn; Rn := Rm; Rd := temp
Load	LDR Rd, <a_mode2>	Rd := address
	LDM <a_mode4L> Rd{!}, <reglist>	Load list of registers from [Rd]
Store	STR Rd, <a_mode2>	[address]:= Rd
	STM <a_mode4S> Rd{!}, <reglist>	Store list of registers to [Rd]
SWI	SWI <immed_24>	Software Interrupt

Addressing Mode 2 - Data Transfer		
Pre-indexed	Immediate offset	[Rn, #+/-<immed_12>]{!}
	Zero offset	[Rn]
	Register offset	[Rn, +/-Rm]{!}
	Scaled register offset	[Rn, +/-Rm, LSL #<immed_5>]{!}
		[Rn, +/-Rm, LSR#<immed_5>]{!}
		[Rn, +/-Rm, ASR #<immed_5>]{!}
		[Rn, +/-Rm, ROR #<immed_5>]{!}
		[Rn, +/-Rm, RRX]{!}
Post-indexed	Immediate offset	[Rn], #+/-<immed_12>
	Register offset	[Rn], +/-Rm
	Zero offset	[Rn]
	Scaled register offset	[Rn], +/-Rm, LSL #<immed_5>
		[Rn], +/-Rm, LSR #<immed_5>
		[Rn], +/-Rm, ASR #<immed_5>
		[Rn], +/-Rm, ROR #<immed_5>
		[Rn], +/-Rm, RRX

Key to tables	
{cond}	See Condition Field
<Oprnd2>	See Operand 2
{S}	Updates CPSR if present
<immed>	Constant
<a_mode2>	See Addressing Mode 2
<a_mode4>	See Addressing Mode 4
<reglist>	List of registers with commas
{!}	Updates base register if present