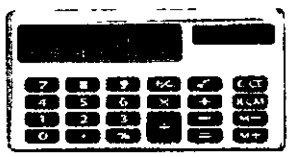
UNIVERSITY EXAMINATIONS



UNIVERSITEITSEKSAMENS

UNISA university of south africa

COS2621

May/June 2016

COMPUTER ORGANISATION

Duration 2 Hours

100 Marks

EXAMINERS
FIRST DR AP ABIDOYE
SECOND PROF C OMLIN

Use of a non-programmable pocket calculator is permissible.

Closed book examination

This examination question paper remains the property of the University of South Africa and may not be removed from the examination venue

This paper consists of the following

1 17 pages with questions (p 1-17)

2 An additional 3 empty pages (p 18–20 for additional space

3 An appendix (p i–v)

INSTRUCTIONS

1 Answer all questions in the space provided below the questions Use the additional blank pages at the back if necessary and write the question number in the space provided

2 All rough work must be done in this question and answer book

3 The mark for each question is given in brackets next to the question

4 The appendix contains a list of IBM PC Assembly Language instructions

GOOD LUCK!

[TURN OVER]

Question 1: General Knowledge **[30]**

Indicate which one of alternatives (1) to (5) you consider to be the correct one by putting *** (3 stars) next to the corresponding answer for each question

a) The processing required for a single instruction is called a(n) __________ cycle [2]

1 execute
2 fetch
3 instruction
4 packet
5 none of the above

b) Convert the following hexadecimal number to binary $C7AE_h$ [2]

1. 1100 0111 1011 1101
2 1100 0111 1010 1110
3 1010 0111 1010 1101
4 1100 0111 1011 1101

ROUGH WORK

c) Which one of the following instructions would you use if you want to branch to the label TEST if the sign flag is set? [2]

1 JN TEST
2 JZ TEST
3 JS TEST
4 JNS TEST
5 JNP TEST

d) The ____________ is an area in memory that is used to save data and addresses that need to be temporarily stored while the program is executing [2]

1 cache
2 execution memory
3 CPU memory
4 stack
5 none of the above

e) NASM is a(n) __________ while DEBUG is a(n) _______ [2]

1 compiler, linker and loader
2 assembler, assembler
3 assembler, utility to assemble and debug programs
4 translator, interpreter
5 compiler, assembler

f) A _________ operation removes a value from the stack and places it in a register or variable [2]

1 PUSH
2 MOV
3 INT
4 POP
5 none of the above

g) When booting the computer, Disk Operating System (DOS) looks for a file called ______ on the root directory of the boot disk [2]

1 MSDOS SYS
2 AUTOEXEC BAT
3 IO SYS
4 COMMAND COM
5 CONFIG SYS

h) A logical cache stores data using __________ [2]

1 physical addresses
2 virtual addresses
3. random addresses
4 both 1 and 3/
5 none of the above

i) What is the postfix version of the following infix expression? [2]

(A * B + C) / (E – F)

1 BC+A*/EF-
2 ABC+*/EF-
3 AB*C+/EF-
4 AB*C+EF-/
5 none of the above

j) Consider the following assembly language program fragment. [2]

```
        org 0x100
label1  dw  0x76EF
        nop
        mov ax,ax
        nop
```

Which one of the following statements is true?

1 76h is stored in position 101h and EFh is stored in position 100h
2 76h is stored in position 100h and EFh is stored in position 101h
3 67h is stored in position 101h and EFh is stored in position 100h
4 67h is stored in position 100h and EFh is stored in position 101h
5 None of the above

k) An unsigned byte uses __________ bits for its numeric value [2]

1 2
2. 4
3 6
4 8
5. 10

l) Which one of the following is NOT a reason for writing programs in assembly language rather than using a high-level programming language?

1. Code is often much faster
2 Insight into the problem often allows real improvement in performance
3 It allows complete access to hardware
4 The program logic is easy to follow
5 It can be used to write function libraries that are compatible with multiple operating systems

m) A segment may be located ______________ in the memory provided it starts on a paragraph boundary [2]

1 at the back
2 anywhere
3 at the front
4. 1. and 3.
5. none of the above

n) The __________ register holds the base location of all executable instructions (code) in a program [2]

1 data segment
2 code segment
3. stack segment
4 extra segment
5 accumulator segment

o) A CPU's ____________ is the set of machine instructions that the CPU is able to execute [2]

1 binary number
2 logical set
3 instruction set
4 variable number
5 microcode

Question 2: Computer Architecture **[16]**

a) What, in general term, is the distinction between computer architecture and computer organization? (2)

ii) Give examples of each (2)

i

ii

b) Briefly explain 4 registers in the structure of the IAS (Institute for Advanced Studies) computer (4)

i.

ii

iii

iv

c) What are the 4 basic functions that a computer can perform? (8)

i

ii

iii

iv

Question 3: Performance Assessment **[10]**

a) A benchmark program is run on a 400-MHz processor The executed program consists of 2 million instruction executions The cycles per instruction (CPI) and instruction mix are given in a table below

Instruction Type	CPI	Instruction Mix (%)
Arithmetic and logic	1	60
Load/store with cache hit	2	18
Branch	4	12
Memory reference with cache miss	8	10

Determine the effective CPI and millions of instructions per second (MIPS) of the processor (4)

b) Briefly describe 3 methods used to increase processor speed (6)

II

III

Question 4: Design for Performance **[10]**

a) The evolution of CPUs has led to integrated circuits of higher and higher density in the pursuit of speed However, as clock speed and logic density increase, a number of physical obstacles (heat generation, increased delays due to resistance and capacitance, memory latency) become more significant Thus, other approaches in CPU organization and architecture are needed to improve performance Briefly describe the following techniques built into modern processors designed to improve CPU performance

i Pipelining (2)

ii Branch prediction (2)

iii Data flow analysis (2)

iv Speculative execution (2)

v Multicore organization (2)

Question 5: Number Systems & Arithmetic **[12]**

a) Mention three (3) reasons for using hexadecimal number in a modern computer (3)

i

ii

iii

b) Convert the following binary numbers to their decimal equivalents (6)
i 11100 011 ii 110011 10011 iii 1010101010 101

i.

ii

iii

c) Convert the following hexadecimal numbers to their binary equivalents (3)
i. D52 ii 67E iii ABCD

i

ii

iii

Question 6: Memory Hierarchies **[10]**

In practice, a memory system is a hierarchy of storage devices with different capacities, costs, and access times

a) Give an example of a memory hierarchy with at least 4 levels (2)

b) What are the trade-offs with respect to the three key characteristics of memory? (2)

c) What is the design objective of cache memory? (2)

d) Describe what happens when a processor attempts to read a word of memory (2)

e) What is the basis for the performance advantage of multi-level memory over single-level memory? Explain (2)

Question 7: Assembly Programming **[12]**

a) Give the x86 assembler code to implement the following program structure (8)

```
For I = 1 to 10 do
{
If A = I or C ≠ I then
   Array(I) = A + 1
else
   if C = I and A ≠ B then
   Array(I) = B - 1
}
```

You may assume that all the variables used in the program have already been defined
Hint Use indexed addressing to access the elements in the array

b) How long will it take to execute the following code using the assumptions given below? (4)

```
mov  ax,2h
mov  bx,3h
mul  bx
add  ax,bx
```

Assume the following

(i) The machine on which the above program fragment is to be executed has a 250 MHz clock

(ii) Four cycles are needed to fetch an instruction from memory (Instructions are not prefetched)

(iii) Multiplication takes 5 clock cycles to execute

(iv) Addition takes 4 cycles to execute

(v) The mov instruction takes 2 cycles to execute

(vi) Fetch and execute cycles do not overlap.

Express your answer in microseconds (µsec), where 1 µsec = 1×10^{-6} sec

Question ____ (continued):

Question ____ (continued):

Question ____ (continued):

APPENDIX

DOS INTERRUPT 20h - terminate program		
DOS INTERRUPT 21h		
AH	**Purpose**	**Description**
0	Terminate program	Terminates execution of program
1	Keyboard input	Waits for keyboard input displays it and returns it in AL
2	Display output	Displays the character in DL on the screen
3	Auxiliary input	Waits for a character from the communications port and puts it in AL
4	Auxiliary output	Outputs the character in DL to the communications port
5	Printer output	Outputs the character in DL to the printer
6	Keyboard input Screen output	Performs both input and output It can also determine the input status Unlike the other input functions, it does not wait for an input character Also note that the input character is not automatically displayed and that the Ctrl-Break command does not terminate the operation As usual the DL register contains the output character and the AL register receives the input character You ask for input by placing the value FFh in the DL register On return from the function, the zero flag is set if no character is ready If the zero flag is clear, it means a character was read into the AL register If the DL register contains any value other than FFh that character is sent to the standard output device Also compare with functions 7 and 8
7	Keyboard input no echo	Waits for a character from the keyboard and puts it in AL The character is not displayed on the screen
8	Keyboard input no echo	Waits for a character from the keyboard returns it in AL and terminates when Ctrl-Break is pressed
9	Display string	We have seen that functions 2 and 6 can display single characters, but function 9 is much easier to use for more than one character Of course non-displayable characters (such as carriage return, line feed and Esc) can also be included in the string To use this function you place the string somewhere in memory and terminate it with a dollar sign The address of the string is placed in the DS DX registers Set AH register to 9 and execute the INT 21h instruction
0A	Buffered keyboard	Reads characters from the keyboard into memory buffer DS DX points to the input buffer 1st byte = max number of characters and 2nd byte = actual number of characters entered The string is stored from the third byte onwards
0B	Check keyboard	Checks to see if a character is available from the keyboard Sets AL=FFh if a Status character is ready Otherwise AL=0
0C	Clear keyboard	Clears keyboard buffer and performs the INT 21h keyboard input function
4C	Terminate program	Returns control to DOS

INSTRUCTIONS

(Note that the first operand is always the destination operand)

ADD. **ADD** binary numbers Adds the source operand to the destination operand and stores the result in the destination operand
Format: ADD operand1,operand2 **Flags:** OF, SF, ZF, AF, PF, CF

AND **AND**s the bits in the source and destination operands The result is stored in the destination
Format: AND operand1,operand2 **Flags.** OF SF, ZF, AF=?, PF, CF=0

[TURN OVER]

CALL CALLs a near or far procedure Decrements the SP (stack pointer) by 2 (or 4 for a far call) If the procedure is **NEAR** (in the same segment) Pushes the current location (only the offset) of the next instruction onto the stack and transfers control to the destination If the procedure is **FAR** (in a different segment) Pushes the current location (both segment and offset) of the next instruction onto the stack and transfers control to the destination
Format CALL operand **Flags:** None

CLC CLear Carry Flag
Format CLC **Flags.** CF=0

CMP: CoMPares the contents of operand1 and operand2 Compares the destination to the source by doing an implied subtraction of the source from the destination The values of the operands do not change
Format. CMP operand1,operand2 **Flags:** OF, SF, ZF, AF, PF, CF

DB/DW/DD: These operations are instructions to the assembler
DB Allocates one byte of storage
DW Allocates two bytes of storage
DD Allocates four bytes of storage

DEC. **DEC**rement Subtracts 1 from a byte or word
Format. DEC reg **Flags** OF, SF ZF, AF, PF
DEC mem

DIV **DIV**ide unsigned Divides an unsigned (the leftmost bit is treated as part of the data and not as a sign) dividend by an unsigned divisor If the divisor is 8 bits, the dividend is assumed to be in AX, the quotient is stored in AL and the remainder in AH If the divisor is 16 bits, the dividend is assumed to be in DX AX the quotient is stored in AX and the remainder in DX
Format: DIV reg **Flags:** all undefined
DIV mem

HLT: **HaLT** Stops the CPU until a hardware interrupt occurs
Format: HLT **Flags:** None

IDIV: Integer (signed) **DIV**ision Divides a signed dividend by a signed divisor Compare with DIV

Dividend (operand1)	Divisor (operand2)	Quotient	Remainder
AX (16 bits)	reg/mem (8 bits)	AL	AH
DX AX (32 bits)	reg/mem (16 bits)	AX	DX

Format. IDIV reg **Flags** AF, CF, OF, PF, SF, ZF (all undefined)
IDIV mem
(We need only specify operand2 as either AX or AL is used as operand1 by default)

IMUL: Integer (signed) **MUL**tiplication Performs a signed integer multiplication on either AL or AX Compare with MUL

Multiplicand (operand1)	Multiplier (operand2)	Product
AL (8 bits)	reg/mem (8 bits)	AX
AX (16 bits)	reg/mem(16 bits)	DX AX

Format: IMUL reg **Flags:** OF CF, SF, ZF, AF and PF (all undefined)
IMUL mem
(We need only specify operand2 as either AX or AL is used as operand1 by default)

INC. **INC**rement Adds 1 to a register or memory position The operation **does not affect the carry flag.**
Format: INC reg **Flags** OF, SF, ZF, AF, PF
INC mem

INT. **INT**errupt Generates a software interrupt, which in turn calls a routine in BIOS or DOS
Format: INT immed **Flags** TF, IF

JMP: Unconditional **JuMP** Jumps unconditionally to a label or memory position
Format: JMP operand **Flags:** None

Conditional Jumps Jcondition Jumps if a specific flag condition is true

JA/JNBE	Jump if Above or Jump if Not Below/Equal
JAE/JNB	Jump if Above/Equal or Jump if Not Below
JB/JNAE	Jump if Below or Jump if Not Above/Equal
JBE/JNA	Jump if Below/Equal or Jump if Not Above
JC	Jump if Carry
JCXZ/JECXZ	Jump if CX is Zero or if ECX is Zero
JE/JZ	Jump if Equal or Jump if Zero
JG/JNLE	Jump if Greater or Jump if Not Less/Equal
JGE/JNL	Jump if Greater/Equal or Jump if Not Less
JL/JNGE	Jump if Less or Jump if Not Greater/Equal
JLE/JNG	Jump if Less/Equal or Jump if Not Greater
JNC	Jump if No Carry
JNE/JNZ	Jump if Not Equal or Jump if Not Zero
JNO	Jump if No Overflow
JNP/JPO	Jump if No Parity or Jump if Parity Odd
JNS	Jump if No Sign - jumps if an operation set the sign to positive
JO	Jump if an operation caused an Overflow
JP/JPE	Jump on Parity or Jump if Parity Even
JS	Jump if an operation set the Sign to negative

Format Jump_condition short_label **Flags:** None in all cases
Jump_condition mem

LEA: Load Effective Address Calculates and loads the 16-bit effective address (offset) of a memory position into a register
Format: LEA reg,mem **Flags** None

LOOP: **LOOP** until complete Executes a routine a specified number of times The CX register must be loaded with the iteration count before the start of the loop **Loop** appears at the end of the loop it decrements CX and causes a jump if CX is not equal to zero If CX=0, the next instruction after LOOP is executed
Format: LOOP short_label **Flags:** None

LOOPE/LOOPZ: **LOOP** while Equal or while Zero These instructions are similar to **LOOP** except that they terminate if CX=0 **or** if the ZF flag is set to zero (i e a non-zero condition)

LOOPNE/ LOOPNZ: **LOOP** while Not Equal or loop while Not Zero Similar to **LOOP** except that they terminate if CX=0 or if the ZF flag is set to 1 (zero condition)

MOV **MOV**e data Moves (actually copies) a byte or word from a source (operand2) to a destination operand (operand1)
Format: MOV operand1, operand2 **Flags.** None

MUL **MUL**tiply unsigned integers Multiplies AX or AL by a source operand of 16 or 8 bytes respectively In the first case the product is stored in AX DX and in the second case the product is stored in AX

Multiplicand (operand1)	**Multiplier** (operand2)	**Product**
AX (16 bits)	reg/mem (16 bits)	DX AX
AL (8 bits)	reg/mem (8 bits)	AX

Format: MUL reg **Flags:** Affects CF and OF AF, PF, SF and ZF are undefined
MUL mem
(We need only specify operand2 as either AX or AL is used as operand1 by default)

NEG. **NEG**ate NEG calculates the two's complement of a binary value
Format: NEG operand **Flags:** OF, SF, ZF, AF, PF, CF

NOP NO OPeration Do nothing

[TURN OVER]

Format: NOP **Flags** None

NOT: Performs a logical **NOT** on an operand
Format: NOT operand **Flags:** None

OR: Performs a logical OR on two operands The result is stored in operand 1 (the destination operand)
Format OR operand1,operand2 **Flags:** OF=0, CF=0, SF, ZF, and PF are affected AF is undefined

POP **POPs** a word from the stack into the operand
Format POP operand **Flags:** None

PUSH. **PUSHes** a word onto the stack
Format PUSH operand **Flags:** None

RCL/RCR. Rotate bits **Left** or **Right** through the **Carry** flag
RCL rotates (shifts) the destination operand to the left a number of times as specified by the source operand The carry flag is copied into the least significant bit and the most significant bit is copied into the carry flag with each shift The CL register or a constant may be used to control the number of rotations
RCR rotates (shifts) the destination operand to the right a number of times as specified by the source operand The carry flag is copied into the most significant bit and the least significant bit is copied into the carry flag with each shift The CL register or a constant may be used to control the number of rotations
Format. RCL/RCR reg,1 RCL/RCR mem 1 **Flags:** OF, CF
RCL/RCR reg,CL RCR/RCR mem,CL

REP/REPE/REPZ/ **REP**eat string Used as a prefix to **MOVS/MOVSB/MOVSW/MOVSD/REPNE/REPNZ** operations

REP repeats a string operation a specified number of times CX is used as a counter and is decremented each time the instruction is repeated The operation repeats until CX=0
REPE/REPZ repeats a string operation until CX=0 or ZF=0
REPNZ/REPNE repeats a string operation until CX=0 or ZF=1
Format REP/REPE/REPZ/REPNE/REPNZ string_operation **Flags:** Depends on the associated string operation

RET/RETN/RETF **RET**urn from a procedure previously entered by a **NEAR** or **FAR** CALL
NEAR **RET/RETN** (return near) pops the word at the top of the stack into IP and increments SP by 2
FAR **RET/RETF** pops the 2 words at the top of the stack into IP and CS and increments SP by 4
An optional immediate 8_bit operand (called a pop_value) is added to the SP
Format RET **Flags:** None
RET operand

ROL/ROR: **Rotate** Left or **R**ight
ROL rotates (shifts) the destination operand to the left a number of times as specified by the source operand The leftmost (most significant) bit is rotated into the least significant position The most significant bit also moves into the carry flag for each shift The CL register or a constant may be used to control the number of rotations
ROR rotates (shifts) the destination operand to the right a number of times as specified by the source operand The rightmost (least significant) bit rotates into the most significant position The least significant bit also moves to the carry flag for each shift The CL register or a constant may be used to control the number of rotations
Format: ROL/ROR reg 1 ROL/ROR mem,1 **Flags** OF,CF
ROL/ROR reg CL ROL/ROR mem,CL

SAL/SAR: Shift Arithmetic Left or Right
SAL shifts the destination operand to the left a number of times as specified by the source operand Zero is shifted into the least significant bit and the most significant bit is shifted into the carry flag for each shift The CL register or a constant may be used as a source operand
SAR shifts the destination operand to the right a number of times as specified by the source operand The most significant bit retains its previous value which means that the sign bit is duplicated with each shift The least significant bit is shifted into the carry flag with each shift The CL register or a constant may be used as source operand
Format SAL/SAR reg,1 **Flags:** CF,OF,PF,SF,ZF (AF undefined)
SAL/SAR mem,1
SAL/SAR reg CL

SAL/SAR mem,CL

SBB: **SuB**tract with **B**orrow Subtracts the source operand from the destination operand and then subtracts the value of CF from the destination SBB is used in multiword subtraction to carry an overflowed 1 bit into the next stage of arithmetic

Format: SBB reg,reg SBB reg,immed
SBB reg,mem SBB mem,immed
SBB mem,reg SBB AL immed
SBB AX,immed

Flags: OF, SF ZF, AF, PF, CF

SCAS/SCASB/SCASW: **SCA**n String Scans a string in memory pointed to by ES DI for a value that matches a value in AL or AX

Format· SCAS operand SCASB (match with AL)
SCAS ES operand SCASW (match with AX)

Flags· OF, SF, ZF, AF PF, CF

SHL/SHR **SH**ift logical **L**eft or **SH**ift logical **R**ight
SHL is identical to SAL
SHR shifts bits to the right a specified number of times and fills the bits on the left with 0's
The CL register or a constant (for some assemblers the constant may only be a 1) may be used to control the number of shifts

Formats SHL/SHR reg,immed SHL/SHR reg CL
SHL/SHR mem,immed SHL/SHR mem CL

Flags· OF, SF, ZF AF (undefined), PF CF

STC: **S**e**T** **C**arry flag Sets the CF to 1
Format: STC **Flags** CF=1

STD· **S**e**T** **D**irection flag Sets the DF flag to 1 This causes SI and/or DI to be decremented by string operations which in turn causes string operations to process strings from right to left
Format STD **Flags** DF=1

STOS/STOSB/ Store string
STOSW/STOSD Stores either AL (byte) or AX (word) in the memory position addressed by ES DI DI is incremented if DF=0 and DI is decremented if DF=1 (compare with STD)
Format. STOS mem STOSB (stores AL) **Flags:** None
STOS ES mem STOSW (stores AX)

SUB **SUB**tract Subtracts the source operand from the destination operand
Format SUB operand1,operand2 **Flags** OF, SF, ZF, AF, PF, CF

TEST **TEST** bits Tests individual bits in the destination operand against those in the source operand TEST performs a logical AND, but the **destination operand is not affected**
Format TEST reg,reg TEST reg immed
TEST reg mem TEST mem,immed
TEST mem,reg TEST AL,immed
TEST AX,immed
Flags· OF=0, SF ZF, AF (undefined), PF, CF=0

XCHG **EXCH**an**G**es operands Exchanges the contents of the two operands
Format: XCHG operand1 operand2 **Flags** None

XOR EXclusive **OR** Performs a logical exclusive OR on the bits in two operands
Format XOR operand1 operand2 **Flags** OF=0, SF, ZF AF (undefined), PF, CF=0

©
UNISA 2016