Université d'Ottawa · University of Ottawa

SCHOOL OF INFORMATION TECHNOLOGY AND ENGINEERING

Préparation de l'examen de mi-session d'automne 2022

COURSE: CEG3136/CEG3536 **PROFESSOR:** Mohamed Ali Ibrahim, ing., Ph.D.

SEMESTER: **DATE:**

TIME: **80 minutes**

MIDTERM EXAMINATION

SOLUTION

NAME and STUDENT NUMBER:______________________________/_________________

Instructions:

- Answer ALL questions on the questionnaire.
- This is a close-book examination.
- Use the provided space to answer the following questions. If more space is needed, use the back of the page.
- Show all your calculations to obtain full marks.
- Calculators are allowed.
- Read all the questions carefully before you start.
- You may detach pages 8 to 12.

1. There are three (3) parts in this examination.

Part 1	**Short Answer**	**20 marks**	
Part 2	**Theory**	**10 marks**	
Part 3	**Application**	**25 marks**	
Total		**55 marks**	

Part 1a – Short Answer Questions (2 points per question – total 10 points)

Answer questions 1 to 5 given that the state of the memory and CPU contents shown on page 7. The diagram represents the initial conditions for each question.

1) What is the content of index register X after the execution of the following instructions?

```
LDY 0,X
LDX 0,Y                         X = …C161……
```

2) Identify all the changes to memory made by the following sequence of instructions.

```
ROR 0,X
ROR 0,X                 At address location $1A04, memory changed from $1A to $41
ROR 0,X
ROR 0,X
```

Rotates to the right with the carry bit

C	**Byte**	
0	**0001 1010**	**Initial**
0	**0000 1101**	
1	**0000 0110**	
0	**1000 0011**	
1	**0100 0001**	

3) Identify the destination of the data operand and the value of the data operand in the last instruction of the following sequence.

```
PULA
INCA
STAA -6,SP
```

Destination : **1A07**

Data value : **FD**

4) What is the content of accumulator A and the condition code bits NZVC after the execution of the following instructions.

```
LDAA      $1A02
PULB
SUBA      0,SP          A = .....$33......... NZVC = .0001
```

A contains $10, Pulb increments SP, so that 0,SP points to $DD, so that $10 - $DD = $33
Carry is set to 1, no 2's complement overflow, result not zero, result not negative

5) Give the instruction that loads the value at address $1A0A using indirect-indexed addressing into accumulator B.

LDAB [0,X]   also LDAB [-8,SP]

Part 1b – Short Answer Questions (total 10 points)

Complete the translation of the C function to Assembler code.
Integers are two bytes long

```
/*------------------------------
Function: duplExist
Parameters: intPt – pointer to array
Description: Scans the array to see
             if duplicate values
             exist in the array.
Returns:  0 of no duplicates and
          1 otherwise.
Assumption: Array is terminated
            with value -1 ($FFFF).
            Values are signed, but
            only positive values
            exist in the array.
            Integers are 2 bytes
            long.
------------------------------*/
int duplExist(unsigned int *intPt)
{
   int retVal = 0; // return value
   int *pt1, *pt1; // to scan array
   // only need to go to second
   // last value in array with pt1
   pt1 = intPt;
   while(*(pt1+1) != 0xFFFF &&
          !retval)
   {
      pt2 = pt1+1;
      while(*pt2 != 0xFFFF &&
             !retval)
      {
         if(*pt1 == *pt2) retval=1;
         pt2++;
      }
      pt1++;
   }
   return(retVal);
}

Hints:
   Recall that in C, the value 0
   is treated as false and any
   other value as true.

   Note that pt1+1 adds 2 to the
   value of pt1 to produce the
   address value since integers
   are 2 bytes long and pt1
   references integer values
```

```
;-------------Assembler Code-------------
; Subroutine - duplExist
; Parameters - intPt - stored on stack.
; Results:     returned in register B
; Local variables:
;     retVal - on stack
;     pt1, pt2 – Register X and Y respectively
;                pointers to array
; Description: See C function.

   OFFSET 0
DEX_RETVAL DS.B 1 ; retVal local variable
           DS.B 1 ; preserve A
           DS.W 1 ; preserve Y
           DS.W 1 ; preserve X
           DS.W 1 ; return address
DEX_INTPT DS.W 1 ; intPT parameter

duplExist: PSHX ; preserve register
           PSHY
           PSHA
           LEAS -1,SP  ; make room for retVal
           clr DEX_RETVAL,SP ; retVal = 0;
           ldx DEX_INPT,SP  ; pt1 = intPT;
DEX_WHILE1 ldd 2,x  ; while(*(pt1+1) != 0xFFFF
           cpd #$FFFF ;
           beq DEX_ENDWHILE1
           tst DEX_RETVAL,SP  ; && !retval)
           bne DEX_ENDWHILE1; {
                      ; {
            leay 2,x; ; pt2 = pt1+1;
DEX_WHILE2 ldd 0,y      ; while(*pt2 != 0xFFFF
            cpd #$FFFF
            beq DEX_ENDWHILE2
            tst DEX_RETVAL,SP   ;   && !retval)
            bne DEX_ENDWHILE2; {
                             ; {
DEX_IF      ldd 0,x          ;   if(*pt1 == *pt2)
            cpd 0,y
            bne DEX_ENDIF
            movb #01,DEX_RETVAL,SP   ; retVal=1;
DEX_ENDIF
            iny              ;  pt2++
            iny
            bra DEX_WHILE2
DEX_ENDWHILE2                ; }

           inx        ;    pt1++
           inx
           bra DEX_WHILE1
DEX_ENDWHILE1        ; }
           lda DEX_RETVAL,SP ; return(retVal);
           leas 1,SP  ; remove retVal from stack
           pulx            ; restore registers
           puly
           pula
           rts
```

Part 2 Theory (total 10 points)

(10 points) Describe the CPU12 indirect indexed addressing modes. Illustrate with detailed examples of all addressing modes.

Explanation

The CPU12 offers 2 indirect indexed addressing modes, one that uses a 16-bit constant offset from registers X, Y, SP, or PC, and a second that used accumulator offset, D, from the registers X, Y, SP, or PC.

Indexed indirect addressing consists of adding the offset (16-bit constant or contents of accumulator D) to the index register (X, Y, SP, or PC) to provide an address of the memory location where the effective address (used to address the data) required by the instruction.

Examples:

Example of the constant offset indirect indexed addressing mode:

Consider the following:

- **Register X contains the value $2000**
- **Memory location $3000/$3001 contains $1500**
- **Memory location $1500 contains the value $F0**

The instructions ldaa [$1000, X] is executed as follows:

- **The offset $1000 is added to the contents of X ($2000) to give $3000**
- **The address at $3000 is read to give the effective address $1500.**
- **The effective address is used to get the data $F0 that is loaded into accumulator A.**

Example of the accumulator offset indirect indexed addressing mode:

Consider the following:

- **Register X contains the value $2000**
- **Register D contains the value $1000**
- **Register Y contains $F0F0**
- **Memory location $3000/$3001 contains $1500**

The instructions sty [$1000, X] is executed as follows:

- **The value contained D ($1000) is added to the contents of X ($2000) to give $3000**
- **The address at $3000 is read to give the effective address $1500.**
- **The contents of register Y, $F0F0 is stored at the effective address $1500/$1501.**

Part 3 – Application Question (total 22 points)

The C standard library provides a function to compare two strings such as:

```
short strcmp(char *str1, char *str2)
```

A *string* of characters terminated with a null character is stored in the memory starting at address found in the pointer variable *str1* and a second string at address found in the pointer variable *str2*. Develop a structured assembly subroutine that compares the string pointed to by *str1* to the string pointed to by string *str2*. The subroutine returns

– a postive value if the string referenced by str1 is alphabetically smaller than the string referenced by str2,
– 0 if both strings are the same, and
– a negative if the string is referenced by str1 alphabetically larger than the string reference by str2.

Assume single byte ASCII characters.

1. First provide a C function that illustrates the design of the subroutine.
2. Then translate the C function to assembler code to subroutine. Do not forget to comment your code.

Hints:

- Compare strings character by character.
- Strings are equal when the end of both strings is reached at the same time.
- As soon as a difference between 2 characters is encountered, strings are not equal, and the return value can be obtained by subtracting the character of the second string (reference by str2) from the character of the first string (referenced by str1). For example if str1 points to the string “abcde” and str1 points to the string “abedc”, then the return value shall be: ‘c’ – ‘e’ = -2.
- The type `short` is a one byte signed integer.

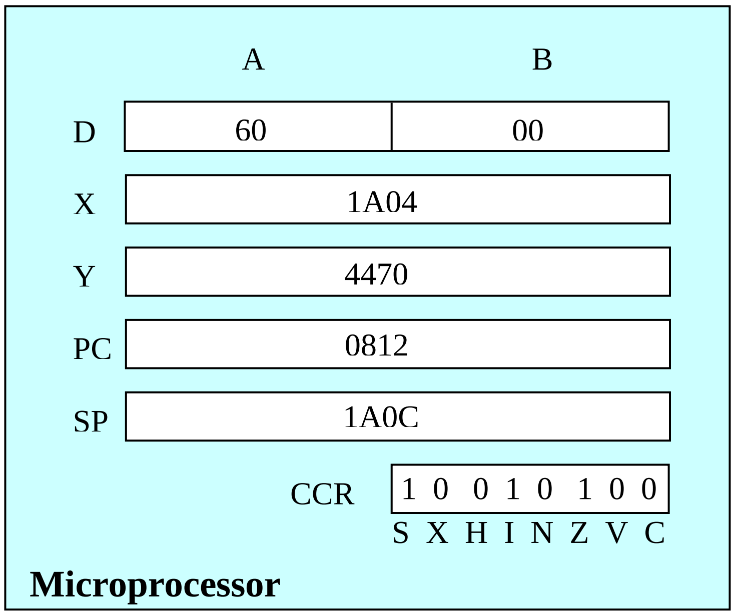
Design (C function):

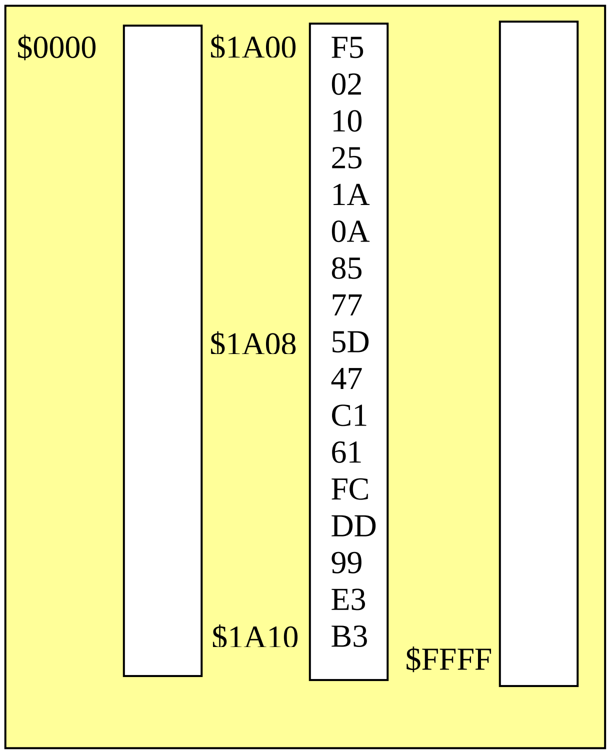
```
short strcmp(char *str1, char *str2)
{
        // use str1 and str2 pointers
        short retval = 0;

        while(*str1 != '\0' || *str2 != '\0')  //loop until end of equal strings
        {
           if(*str1 != *str2)
           {
               retval = *str1 - *str2;
               break;
           }
           else
           {
               str1++;
               str2++;
           }
        }
        return(retval);
}
```

Assembler Source Code:

```
; Subroutine: short strcmp(char *str1, char *str2)
; Parameters
;       str1 - address of first string - on stack and register x
;       str2 - address of second string - on stack and register y
; Returns in register B
;       int  -ve value: str1 less than str2
;             +ve value: str1 greater than str2
;             0: strings are the same.
; Local Variables
;       retval - return value in register B
; Description: Compares the strings and returns a velue that reflects
;              the results of the comparison.
;
; Stack Usage:
        OFFSET 0
SCMP_RETVAL  DS.B 1  ; return value
SCMP_VARSIZE
SCMP_PRY   DS.W 1   ; preserve B
SCMP_PRX   DS.W 1   ; preserve X
SCMP_RA    DS.W 1   ; return address
SCMP_STR1 DS.W 1   ; address of first string
SCMP_STR2 DS.W 1   ; address of second string

strcmp: pshx    ; preserve registers
        pshy
        leas -SCMP_VARSIZE,sp
        clr SCMP_RETVAL,sp
        ldx SCMP_STR1,sp ; get address of first string
        ldy SCMP_STR2,sp  ; get address of second string
        clra
scmp_while:     ; while(*str1 != 0 && *str2 != 0)
        tst 0,x
        bne scmp_if
        tst 0,y
        beq scmp_endwhile
scmp_if:
        ldab 0,x        ; if(*str1 ¡= *str2)
        cmpb 0,y
        beq scmp_else
        subb 0,y
        stab SCMP_RETVAL,SP
        bra scmp_endwhile
scmp_else
        inx
        iny
scmp_endif
         bra scmp_while
scmp_endwhile:
        ldab SCMP_RETVAL,SP
        leas SCMP_VARSIZE,SP ; restore stack pointer
        puly    ; restore registers
        pulx
      rts
```

CPU and Memory for Part 1





Memory

68HCS12 INSTRUCTION LIST (reduced)

Loads, Stores, and Transfers

Function	Mnemonic	IMM	DIR	EXT	IDX	[IDX]	INH	Operation	N Z V C
Clear Memory Byte	CLR			X	X	X		m(ea) <= 0	0 1 0 0
Clear Accumulator A (B)	CLRA (B)						X	A <= 0	0 1 0 0
Load Accumulator A (B)	LDAA (B)	X	X	X	X	X		A <= [m(ea)]	Δ Δ 0 -
Load Double Accumulator D (S, X or Y)	LDD	X	X	X	X	X		D <= [m(ea, ea+1)]	Δ Δ 0 –
Load Effective Address into SP (X or Y)	LEAS (A,B)							SP <= ea	– – – –
Store Accumulator A (B)	STAA (B)	X	X	X	X	X		m(ea) <= (A)	Δ Δ 0 -
Store Double Accumulator D (S, X or Y)	STD	X	X	X	X	X		m(ea, ea+1) <= D	Δ Δ 0 -
Transfer A to B	TAB						X	B <= (A)	Δ Δ 0 -
Transfer A to CCR	TAP						X	CCR <= (A)	Δ Δ Δ Δ
Transfer B to A	TBA						X	A <= B	Δ Δ 0 -
Transfer CCR to A	TPA						X	A <= (CCR)	– – – –
Exchange D with X (Y)	XGDX						X	D <=> (X)	– – – –
Pull A (B) from Stack	PULA(B)						X	A <= [m(SP)], SP <= (SP)+1	– – – –
Push A (B) onto Stack	PSHA(B)						X	SP <= (SP)-1, m(SP) <= A	– – – –
Move byte(word) in memory	MOVB(W)	IMM/EXT/IDX, EXT/IDX						M_1 => M_2 (M:M+1_1 => M:M+1_2)	– – – –

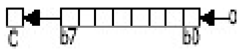
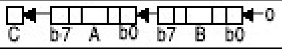
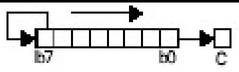
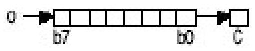
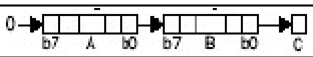
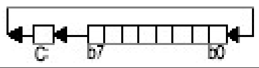
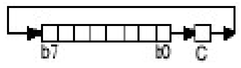
Arithmetic Operations

Function	Mnemonic	IMM	DIR	EXT	IDX	[IDX]	INH	Operation	N Z V C
Add Accumulators	ABA						X	A <= (A) + (B)	Δ Δ Δ Δ
Add with Carry to A (B)	ADCA (B)	X	X	X	X	X		A <= (A) + [m(ea)]+(C)	Δ Δ Δ Δ
Add Memory to A (B)	ADDA (B)	X	X	X	X	X		A <= (A) + [m(ea)]	Δ Δ Δ Δ
Add Memory to D (16 Bit)	ADDD	X	X	X	X	X		D <= (D) + [m(ea,ea+1)]	Δ Δ Δ Δ
Decrement Memory Byte	DEC			X	X	X		m(ea) <= [m(ea)] – 1	Δ Δ Δ –
Decrement Accumulator A (B)	DECA (B)						X	A <= (A) – 1	Δ Δ Δ –
Increment Memory Byte	INC			X	X	X		m(ea) <= [m(ea)] + 1	Δ Δ Δ –
Increment Accumulator A (B)	INCA (B)						X	A <= (A) + 1	Δ Δ Δ –
Subtract with Carry from A (B)	SBCA (B)	X	X	X	X	X		A <= (A) – [m(ea)] – C	Δ Δ Δ Δ
Subtract Memory from A (B)	SUBA (B)	X	X	X	X	X		A <= (A) – [m(ea)]	Δ Δ Δ Δ
Subtract Memory from D (16 Bit)	SUBD	X	X	X	X	X		D <= (D) – [m(ea,ea+1)]	Δ Δ Δ Δ
Multiply (byte, unsigned)	MUL						X	D <= (A) x (B)	– – – Δ
Multiply word, unsigned (signed)	EMUL(S)						X	Y:D <= (D) x (Y)	Δ Δ – Δ
Unsigned (signed) 32 by 16 divide	EDIV(S)						X	X <= (Y:D) /.(X),Y <= quotient, D <= remainder	Δ Δ Δ Δ
Fractional Divide (D < X)	FDIV						X	X <= (D) /.(X), D <= remainder	– Δ Δ Δ
Integer Divide (unsigned)	IDIV						X	X <= (D) /.(X), D <= remainder	– Δ 0 Δ

Logical Operations

Function	Mnemonic	IMM	DIR	EXT	IDX	[IDX]	INH	Operation	N Z V C
AND A (B) with Memory	ANDA (B)	X	X	X	X	X		A <= A • [m(ea)]	Δ Δ 0 –
Bit(s) Test A (B) with Memory	BITA (B)	X	X	X	X	X		A • [m(ea)]	Δ Δ 0 –
One’s Complement Memory Byte	COM			X	X	X		m(ea) <= [/m(ea)]	Δ Δ 0 1
One’s Complement A (B)	COMA (B)						X	A <= /A	Δ Δ 0 1
OR A (B) with Memory (Exclusive)	EORA (B)	X	X	X	X	X		A <= A ⊕ [m(ea)]	Δ Δ 0 –
OR A (B) with Memory (Inclusive)	ORAA (B)	X	X	X	X	X		A <= A + [m(ea)]	Δ Δ 0 –

Shift and Rotate

Function	Mnemonic	IMM	DIR	EXT	IDX	[IDX]	INH	Operation	N Z V C
Arithmetic/Logical Shift Left Memory	ASL/LSL			X	X	X			Δ Δ Δ Δ
Arithmetic/Logical Shift Left A (B)	ASLA(B)						X		Δ Δ Δ Δ
Arithmetic/Logical Shift Left Double	ASLD/LSLD						X		Δ Δ Δ Δ
Arithmetic Shift Right Memory	ASR			X	X	X			Δ Δ Δ Δ
Arithmetic Shift Right A (B)	ASRA(B)						X		Δ Δ Δ Δ
Logical Shift Right A (B)	LSRA(B)						X		0 Δ Δ Δ
Logical Shift Right Memory	LSR			X	X	X			0 Δ Δ Δ
Logical Shift Right D	LSRD						X		0 Δ Δ Δ
Rotate Left Memory	ROL			X	X	X			Δ Δ Δ Δ
Rotate Left A (B)	ROLA(B)						X		Δ Δ Δ Δ
Rotate Right A (B)	RORA(B)						X		Δ Δ Δ Δ
Rotate Right Memory	ROR			X	X	X			Δ Δ Δ Δ

Compare & Test

Function	Mnemonic	IMM	DIR	EXT	IDX	[IDX]	INH	Operation	N Z V C
Compare A to B	CBA						X	(A)-(B)	Δ Δ Δ Δ
Compare A (B) to Memory	CMPA (B)	X	X	X	X	X		(A) - [m(ea)]	Δ Δ Δ Δ
Compare D to Memory (16 Bit)	CPD	X	X	X	X	X		(D) - [m(ea,ea+1)]	Δ Δ Δ Δ
Compare SP to Memory (16 Bit)	CPS	X	X	X	X	X		(SP) - [m(ea,ea+1)]	Δ Δ Δ Δ
Compare X (Y) to Memory (16 Bit)	CPX	X	X	X	X	X		(X) - [m(ea,ea+1)]	Δ Δ Δ Δ
Test memory for 0 or minus	TST			X	X	X		m(ea) - 0	Δ Δ Δ Δ
Test A (B) for 0 or minus	TSTA (B)						X	(A)-0	Δ Δ Δ Δ

Short Branches

Function	Mnemonic	REL	DIR	IDX	[IDX]	PC <= ea if	N Z V C
Branch ALWAYS	BRA	X					– – – –
Branch if Carry Clear	BCC	X				C = 0 ?	– – – –
Branch if Carry Set	BCS	X				C = 1 ?	– – – –
Branch if Equal Zero	BEQ	X				Z = 1 ?	– – – –
Branch if Not Equal	BNE	X				Z = 0 ?	– – – –
Branch if Minus	BMI	X				N = 1 ?	– – – –
Branch if Plus	BPL	X				N = 0 ?	– – – –
Branch if Bit(s) Clear in Memory Byte	BRCLR		X	X		[m(ea)]•mask=0	– – – –
Branch if Bit(s) Set in Memory Byte	BRSET		X	X		[/m(ea)]•mask=0	– – – –
Branch if Overflow Clear	BVC	X				V = 0 ?	– – – –
Branch if Overflow Set	BVS	X				V = 1 ?	– – – –
Branch if Greater Than	BGT	X				Signed >	– – – –
Branch if Greater Than or Equal	BGE	X				Signed ≥	– – – –
Branch if Less Than or Equal	BLE	X				Signed ≤	– – – –
Branch if Less Than	BLT	X				Signed <	– – – –
Branch if Higher	BHI	X				Unsigned >	– – – –
Branch if Higher or Same (same as BCC)	BHS	X				Unsigned ≥	– – – –
Branch if Lower or Same	BLS	X				Unsigned ≤	– – – –
Branch if Lower (same as BCS)	BLO	X				Unsigned <	– – – –
Branch Never	BRN	X				3-cycle NOP	– – – –

Long branch mnemonic = **L** + Short branch mnemonic, e.g.: BRA → **L**BRA

Loop Primitive Instructions (counter ctr = A, B, or D)

Function	Mnemonic	REL	DIR	EXT	IDX	[IDX]	INH	Operation	N Z V C
Decrement counter & branch if =0	DBEQ	X						ctr <= (ctr)-1, if (ctr)=0 => PC <= ea	– – – –
Decrement counter & branch if ≠0	DBNE	X						ctr <= (ctr)-1, if (ctr) ≠0 => PC <= ea	– – – –
Increment counter & branch if =0	IBEQ	X						ctr <= (ctr)+1, if (ctr)=0 => PC <= ea	– – – –
Increment counter & branch if ≠0	IBNE	X						ctr <= (ctr)+1, if (ctr) ≠0 => PC <= ea	– – – –
Test counter & branch if =0	TBEQ	X						if (ctr)=0 => PC <= ea	– – – –

Subroutine Calls and Returns

Function	Mnemonic	REL	DIR	EXT	IDX	[IDX]	INH	Operation	N Z V C
Branch to Subroutine	BSR	X						SP <= (SP)-2, m(SP) <= (PC), PC <= ea	– – – –
Jump to Subroutine	JSR		X	X	X	X		SP <= (SP)-2, m(SP) <= (PC), PC <= ea	– – – –
CALL a Subroutine (expanded memory)	CALL		X	X	X	X		SP <= (SP)-2, m(SP) <= (PC), PC <= ea SP <= (SP)-1, m(SP) <= (PPG), PC <= pg	– – – –
Return from Subroutine	RTS						X	PC <= [m(SP)], SP <= (SP)+2	– – – –
Return from call	RTC						X	PPG <= [m(SP)], SP <= (SP)+1, PC <= [m(SP)], SP <= (SP)+2	– – – –

Function	Mnemonic	DIR	EXT	IDX	[IDX]	INH	Operation	N Z V C
Jump	JMP	X	X	X	X		PC <= ea	– – – –

The **jump** instruction allows control to be passed to any address in the 64-Kbyte memory map.

Stack and Index Register Instructions

Function	Mnemonic	IMM	DIR	EXT	IDX	[IDX]	INH	Operation	N Z V C
Decrement Index Register X (Y)	DEX (Y)						X	X <= (X) - 1	– Δ – –
Increment Index Register X (Y)	INX (Y)						X	X <= (X) + 1	– Δ – –
Load Index Register X (Y)	LDX(Y)	X	X	X	X	X		X <= [m(ea,ea+1)]	Δ Δ 0 –
Pull X (Y) from Stack	PULX						X	X <= [m(SP,SP+1)] SP <= (SP) + 2	– – – –
Push X (Y) onto Stack	PSHX (Y)						X	m(SP,SP+1) <= (X) SP <= (SP) - 2	– – – –
Store Index Register X (Y)	STX (X)	X	X	X	X	X		m(ea,ea+1) <= X	Δ Δ 0 –
Add Accumulator B to X (Y)	ABX (Y)						X	X <= (X) + (B)	– – – –
Decrement Stack Pointer	DES						X	SP <= (SP) - 1	– – – –
Increment Stack Pointer	INS						X	SP <= (SP) + 1	– – – –
Load Stack Pointer	LDS	X	X	X	X	X		SP <= [m(ea,ea+1)]	Δ Δ 0 –
Store Stack Pointer	STS	X	X	X	X	X		m(ea,ea+1) <= (SP)	Δ Δ 0 –
Transfer SP to X (Y)	TSX (Y)						X	X <= (SP)	– – – –
Transfer X (Y) to SP	TXS (Y)						X	SP <= (X)	– – – –
Exchange D with X (Y)	XGDX (Y)						X	(D) <=> (X)	– – – –

Function	Mnemonic	INH	Operation
Return from Interrupt	RTI	X	$(M_{(SP)}) \Rightarrow CCR; (SP) + \$0001 \Rightarrow SP$ $(M_{(SP)} : M_{(SP+1)}) \Rightarrow B : A; (SP) + \$0002 \Rightarrow S$ $(M_{(SP)} : M_{(SP+1)}) \Rightarrow X_H : X_L; (SP) + \$0004 \Rightarrow$ $(M_{(SP)} : M_{(SP+1)}) \Rightarrow PC_H : PC_L; (SP) + \$0002$ $(M_{(SP)} : M_{(SP+1)}) \Rightarrow Y_H : Y_L; (SP) + \$0004 \Rightarrow$
Software Interrupt	SWI	X	
Wait for Interrupt	WAI	X	

Interrupt Handling

The software interrupt (SWI) instruction is similar to a JSR instruction, except the contents of all working CPU registers are saved on the stack rather than just the return address. SWI is unusual in that it is requested by the software program as opposed to other interrupts that are requested asynchronously to the executing program.