Université d'Ottawa · University of Ottawa

SCHOOL OF INFORMATION TECHNOLOGY AND ENGINEERING

COURSE: CEG3131
SEMESTER: Fall 2009

PROFESSOR: Gilbert Arbez
DATE: October 24, 2009
TIME: 13h00 – 14h30

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**	**18 marks**	
Part 2	**Theory**	**10 marks**	
Part 3	**Application**	**22 marks**	
Total		**50 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 8. The diagram represents the initial conditions for each question.

1) Give the address of the memory location affected by the following instructions. __**$2509**___
Give the contents of the location after the instructions have been executed. __**$9A**_____.

LDAA 1,X+	**loads from $2508 into A $5D, %0101 1101**
ANDA 0,X	**AND $47 from $2509 $5D = $45, %0100 0101**
ORAA #%01100000	**changes contents of A to %0110 0101**
COMA	**complements contents of A to produce %1001 1010**
STAA 0,X	**Stores contents of A, $9A into address $2509**

2) Give the contents of D register after the execution of the following 2 instructions. ___**8547**__

LDAA 2,-X	**pre-decrements X by 2 to $2506, loads $85 into A (from address $2506 in register X),**
LDAB 3, X	**computes effective address as $2506+3=$2509, and loads $47 into B using this EA.**

3) Consider the following instructions:

LDAB $2500	**loads into B the value $F5 (from address $2500)**
CMPB 0,Y	**subtracts $B3 (found at address $2510 in Y) from $F5.**
BGT NEXT	
.	
.	
NEXT LDAA $CC	

Will the instruction BLT branch to NEXT? **Yes, since $F5 (-9) is larger (less negative) than $B3 (–61). BLT treats numbers as signed.** What will be the contents of the register B after the CMPB instruction? __**$F5**___

4) Given that 16-bit unsigned integers are stored at addresses NUMERATOR and DENOMINATOR, what sequence of instructions would you use to store NUMERATOR%DENOMINATOR at address MODULO, where % is the modulo operator.

LDD NUMERATOR
LDX DENOMINATOR
IDIV
STD MODULO

5) Consider the following instructions

OFFSET 0	
LCL_VAR1 DS.B 6	
LCL_VAR2 DS.B 1	
LEAS 2,X	**changes the value of the S to 2+$2508 (contents of S) to $250A**
LDD 0,X	**loads register D with contents at address $2508, which is $5D47**
STD LCL_VAR2,SP	**stores the contents of D at address $250A+6 = $2510**

The last instruction stores the contents of D into a location in memory. Give the address of this location __**$2510**___ and the value that gets stored there __**$5D47**___

Part 1b – Short Answer Questions (total 8 points)

1) (5 points) Translate the following short C program into assembler. Use the stack to exchange ALL parameters and the result (assume *int*'s take 2 bytes of storage, and *byte*'s take 1 byte of storage). Ensure that the registers used by the subroutine are not changed after the subroutine has executed. Define the stack usage using the OFFSET directive and labels as offsets into the stack.

```
/*-----------------------------
Function: addInts
Description: Adds two 8-bit integers.
-----------------------------*/
int addInts(byte val1, byte val2)
{
  int sum;
  sum = val1 + val2;
  return(sum);
}
```

```
;-------------Assembler Code-------------
; Subroutine – addInts
; Parameters – val1 – on stack
;              val2 – on stack
; Results sum – on stack
; Description: Adds two integers.
; Stack usage:
      OFFSET 0
         DS.W 1 ; preserve Register D
         DS.W 1 ; return address
ADI_SUM DS.W 1  ; sum and return value
ADI_VAL1 DS.B 1 ; byte val1
ADI_VAL2 DS.B 1 ; byte val2

addInts: PSHD ; preserve acc D
         CLRA
         LDAB ADI_VAL1,SP
         ADDB ADI_VAL2,SP
         ADCA #00
         STD ADI_SUM,SP
         PULD
         RTS
```

2) (3 points) Complete the translation of the C function to Assembler code.

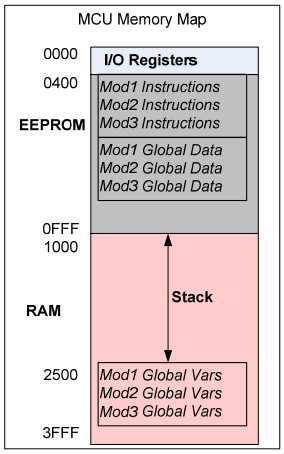
```
/*-----------------------------
Function: addByteArray
Parameters: arrPt – pointer to array
            num – number of elements
                  to sum
Description: Adds the contents of
             an integer array.
Assumption: array contains at
            least one element.
-----------------------------*/
int addByteArray(byte *arrPt,
                 byte num)
{
  int sum;
  sum=0;
  do
  {
     sum=sum+*arrPt++;
     num--;
  }while(num != 0);
  return(sum);
}
```

```
;-------------Assembler Code-------------
; Subroutine – addArr
; Parameters – arrPt – register X
;              num – register Y
; Results: sum – register D
; Description: Adds contents of an integer
;              array.

addByteArray: PSHX
              PSHY ; preserve Registers
              LDD #0 ; sum=0;
loop          ADDB 1,X+ ; sum=sum+*arrPt++;
              ADCA #0
              DEY
              BNE loop









              PULY ; restore registers
              PULX
              RTS
```

Part 2 Theory (total 10 points)

(10 points) Modular design provides the means of separating the tasks of any project into manageable pieces. Each of the modules in a project is typically stored in a separate assembler file. Although this does help facilitate the development of a software project, it produces a challenge – how to collect the executable code, constant global data and variable global data from each module into separate sections of software where code and constant data are store in Read Only Memory (these sections follow one another) and the variable global data is stored in RAM.

Describe how assembler directives, such as ORG, SECTION, SWITCH available in MiniIDE, can be used to perform this organization of software sections.




```
Module 1 (module1.asm)

code      SECTION
          ORG $0400
 ;---Assembler Instructions

dataEE    SECTION
          ORG ENDCODE
 ;---Constant global data

dataRAM SECTION
          ORG $2500
 ;---Global variables (modifiable)

;--------Add other Modules
  INCLUDE module2.asm
  INCLUDE module3.asm
;--Defines start of dataEE section
  SWITCH code
ENDCODE
```

```
Module 2 (module2.asm)
    SWITCH code
;---Assembler Instructions

    SWITCH dataEE
;---Constant global data

    SWITCH dataRAM
;---Global variables (modifiable)
```

```
Module 3 (module3.asm)
    SWITCH code
;---Assembler Instructions

    SWITCH dataEE
;---Constant global data

    SWITCH dataRAM
;---Global variables (modifiable)
```

Directive SECTION – defines the start of a section.
Directive ORG – sets the location counter of a section.
The above two directives can be used to define sections and their locations in memory.
Directive SWITCH – changes section, such that any subsequent assembly instructions (including storage instruction such as DS) are placed in the section.

Note: Absolute addresses are defined only on the second pass and thus ENDCODE which defines the location of the global data is place after all code assembly and consequently places the global constant data section after the code section.

- ENDCODE value can change when code changes when source code is re-assembled.

Part 3 – Application Question (total 22 points)

The C standard library provides a function to concantenate two strings:

```
char *strcat(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 concantenates the contents pointed to by *str2* to the end of the contents in string *str1*. The function returns the address of string *str1* (that is the address received by the *str1* variable). Assume single byte ASCII characters.

1. First provide a C function that illustrates the design of the subroutine (or functions/subroutines – you may use additional functions/subroutines to provide a solution).
2. Then translate the C functions to assembler code to subroutine(s). Do not forget to comment your code.

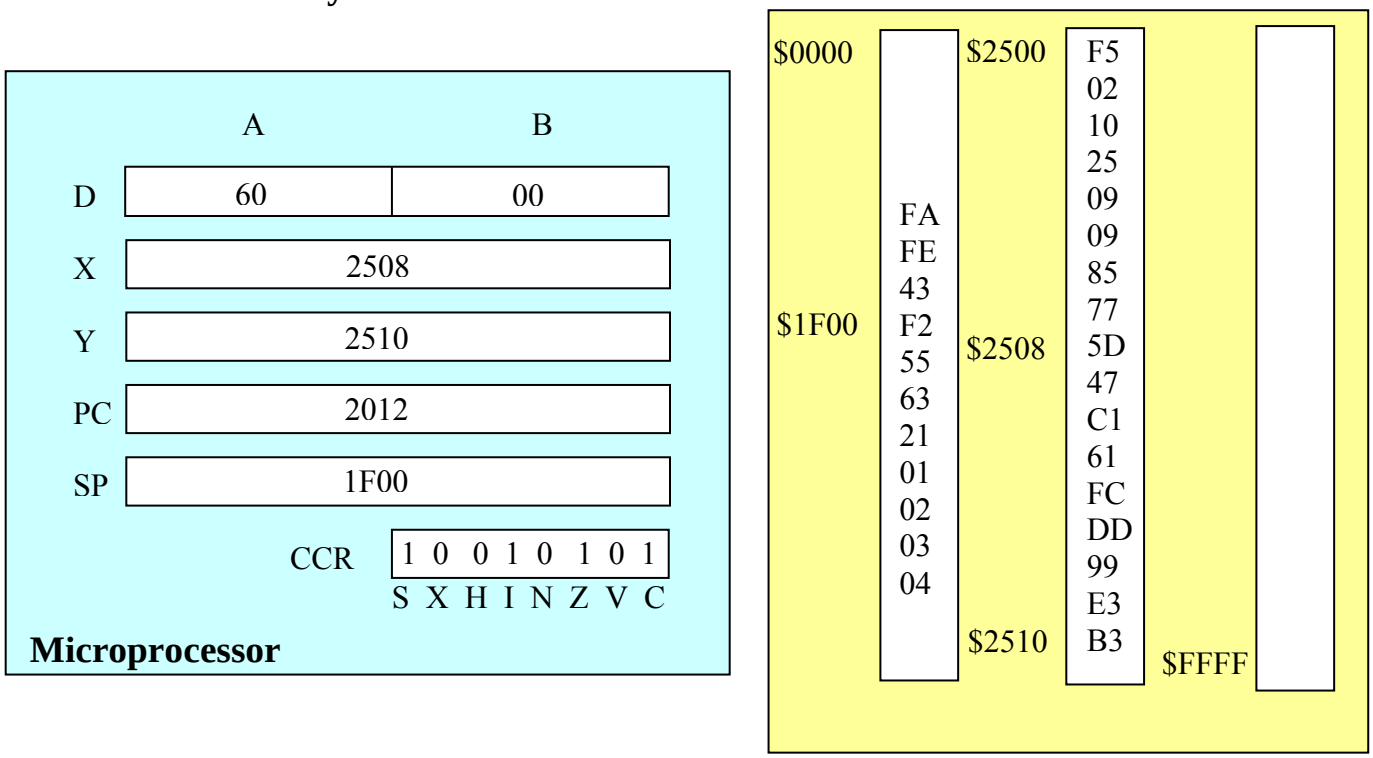
Design (C program and description):

```
char *strcat(char *str1, char *str2)
{
      char *pt;  // working pointer

      pt = findEnd(str1);  // find end of string, i.e. nul character
      while(*str2 != 0)
      {
         *pt++ = *str2++;
      }
      *pt = *str2;
      return(str1);
}

char *findEnd(char *str)
{
      while(*str != 0) str++;
      return(str);
}
```

Assembler Source Code:

```
; Subroutine: char* strcat(char *str1, char *str2)
; Parameters
;       str1 - address of first string - on stack
;       str2 - address of second string to append to first string
;              on stack and reg Y
; Returns
;       str1 - pointer to concatenated string, str1 - on stack
; Local Variables
;       pt - register X
; Description: appends the second string to the first string.
;
; Stack Usage:
        OFFSET 0
SCAT_PRB  DS.B 1  // preserve B
SCAT_PRX  DS.W 1  // preserve X
SCAT_RA   DS.W 1  // return address
SCAT_STR1 DS.W 1  // address of first string
SCAT_STR2 DS.W 1  // address of second string
SCAT_RET  DS.W 1  // address of return value

strcat: pshx    ; preserve registers
        pshy
        ldx SCAT_STR1,SP ; get address of first string
        jsr $findEnd  ; x points to end, i.e. pt
        ldy SCAT_STR2,SP  ; y points to str2
scat_loop:
        tst 0,Y ; (*str2 == 0)
        beq scat_endwhile ; exit loop if true
        movb 1,Y+,1,X+  ; *pt++ = *str2++;
        bra scat_loop
scat_endwhile:
        movb 0,Y,0,X ; *pt = *str2 // nul char
        puly    ; restore registers
        pulx
        rts

; Subroutine: char *findEnd(str)
; Parameters
;       str - address to a string - x
; Returns
;       adr - points to end of string in x
; Description: Finds the end of the string (nul char) and returns its address.
;
; Stack usage:
FEND_PRX DS.2   ; preserve X
FEND_RA  DS.W 1 ; 0 return address

findEnd: pshx           ; preserve b
fend_loop:
        tst 0,x         ; (*pt == 0)
        beq fend_while  ; yes exit loop

        inx             ; pt++
        bra fend_loop
fend_endwhile:
        pulx            ; restore b
        rts
```

CPU and Memory for Part 1



All values shown are hexadecimal.

68HC12 INSTRUCTION LIST (reduced)

Loads, Stores, and Transfers

Function	Mnemonic	IMM	DIR	EXT	IDX	[IDX]	INH	Operation
Clear Memory Byte	CLR			X	X	X		m(ea) <= 0
Clear Accumulator A (B)	CLRA (B)						X	A <= 0
Load Accumulator A (B)	LDAA (B)	X	X	X	X	X		A <= [m(ea)]
Load Double Accumulator D	LDD	X	X	X	X	X		D <= [m(ea, ea+1)]
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)
Store Double Accumulator D	STD	X	X	X	X	X		m(ea, ea+1) <= D
Transfer A to B	TAB						X	B <= (A)
Transfer A to CCR	TAP						X	CCR <= (A)
Transfer B to A	TBA						X	A <= B
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

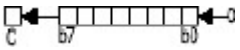
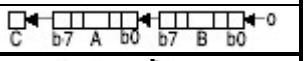
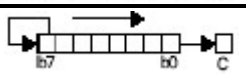
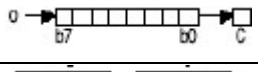
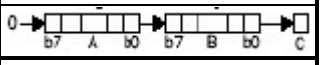
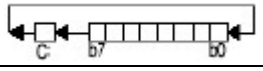
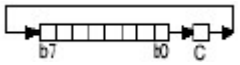
Arithmetic Operations

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

Logical Operations

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

Shift and Rotate

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

Compare & Test

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

Short Branches

Function	Mnemonic	REL	DIR	IDX	[IDX]	PC <= ea if
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
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	DBEQ	X						if (ctr)=0 => PC <= ea

Subroutine Calls and Returns

Function	Mnemonic	REL	DIR	EXT	IDX	[IDX]	INH	Operation
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
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
Decrement Index Register X (Y)	DEX (Y)						X	X <= (X) - 1
Increment Index Register X (Y)	INX (Y)						X	X <= (X) + 1
Load Index Register X (Y)	LDX(Y)	X	X	X	X	X		X <= [m(ea,ea+1)]
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
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)]
Store Stack Pointer	STS	X	X	X	X	X		m(ea,ea+1) <= (SP)
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	$(M_{(SP)}) \Rightarrow CCR; (SP) + \$0001 \Rightarrow SP$ $(M_{(SP)} : M_{(SP+1)}) \Rightarrow B : A; (SP) + \$0002 \Rightarrow S$ $(M_{(SP)} : M_{(SP+1)}) \Rightarrow X_H : X_L; (SP) + \$0004 \Rightarrow S$ $(M_{(SP)} : M_{(SP+1)}) \Rightarrow PC_H : PC_L; (SP) + \$0002 \Rightarrow$ $(M_{(SP)} : M_{(SP+1)}) \Rightarrow Y_H : Y_L; (SP) + \$0004 \Rightarrow S$
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.