



Université d'Ottawa • University of Ottawa

SCHOOL OF INFORMATION TECHNOLOGY AND ENGINEERING

COURSE: CEG3136

SEMESTER: Fall 2017

PROFESSOR: Rami Abielmona

DATE: October 19, 2017

TIME: 17h30 – 19h00

**MIDTERM
EXAMINATION
SOLUTIONS**

NAME and STUDENT NUMBER: _____ / _____

Instructions:

- Answer ALL questions on the questionnaire.
- This is a closed-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 9 to 13.

There are three (3) parts in this examination.

Part 1	Short Answer	20 marks	
Part 2	Theory	10 marks	
Part 3	Application	20 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) Show how memory changes after the execution of **each** of the following instructions?

	Byte at \$0904, Initial value	
LSL 0,X	0000 1001, \$09	
LSL 0,X	0001 0010, \$12	
LSL 0,X	001 00100, \$24	
LSL 0,X	01 001000, \$48	
LSL 0,X	1 0010000, \$90	

2) What is the content of accumulator D (hexadecimal value) after the execution of the following instructions?

VAR1 EQU \$0902

LDAA #16
LDAB VAR1

D = ...\$1010.....

3) Will the branch in the following instructions execute (i.e. will the program branch)? Give the contents of Accumulator A after the execution of the CMPA instruction.

VAR3 EQU \$0900
LDAA 0,SP
CMPA VAR3
BLT next

Loads value \$FC into A from address \$090C
Compares to value \$F5 at address \$0900.

NEXT ldab \$CC

Branch (Y or N) Y? Contents of the A register \$FC?

BLT treats numbers as signed numbers and thus \$FC is smaller (more negative) than \$F5.

4) Give an instruction that loads into accumulator D the value at address \$47C1 (found at address \$0908) using indirect-indexed addressing.

LDD [4,X]
or
LDD [-3,SP]

5) What are the contents of index registers X and Y after the execution of the following instructions?

LDY 1,X+
LDX 0,Y

X = \$47C1 ... Y = \$0909

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

- 1) (5 points) Translate the following short C program into assembler. Use the stack to exchange ALL parameters (assume *int*'s take 2 bytes of storage, and *long*'s take 4 bytes of storage). Return the result in register D. Ensure that the registers, other than D, 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: intDivide
Description: Divide unsigned 32-bit integer by unsigned 16-bit integer.
             Value returned is the quotient of the division, the remainder
             is discarded.
-----*/
unsigned int intDivide(unsigned long val1, unsigned int val2)
{
    unsigned long quotient;
    quotient = val1 / val2;
    return(quotient);
}
```

```
;-----Assembler Code-----
; Subroutine - intDivide
; Parameters - val1 - on stack
;              val2 - on stack
; Results quotient - returned in register D
; Description: Divides unsigned 32-bit integer by unsigned 16-bit.
; Stack usage:
;               OFFSET 0
;               DS.W 1 ; preserve Register X
;               DS.W 1 ; preserve Register Y
;               DS.W 1 ; return address
IDV_VAL1 DS.L 1 ; long val1
IDV_VAL2 DS.W 1 ; int val2

intDivide: PSHY ; preserve Y
          PSHX ; preserve X
          LDY IDV_VAL1,SP
          LDD IDV_VAL1+2,SP
          LDX IDV_VAL2,SP
          EDIV
          TFR Y,D
          PULX
          PULY
          RTS
```

2) (5 points) Complete the translation of the C function to Assembler code. Integers are two bytes long

```

/*-----
Function: countNum
Parameters: intPt - pointer to array
            num - number to count
Description: Counts the number of
            times a value (num)
            occurs in an integer
            array referenced
            by intPt. Array
            contains only positive
            values.
Assumption: Array is terminated
            with value -1 (0xFFFF).
-----*/
int countNum(unsigned int *intPt,
            unsigned int num)
{
    int retVal = 0;
    while(*intPt != 0xFFFF)
    {
        if(num == *intPt) retVal++;
        intPt++;
    }
    return(retVal);
}

```

```

;-----Assembler Code-----
; Subroutine - countNum
; Parameters - intPt - register D,
;               transferred to X
;               num - On the stack
; Results:      returned in register D
; Local variables: retVal - register Y
; Description: Counts the number of times
;               a value (num) occurs in an
;               integer array referenced by
;               intPt. Array contains only
;               positive values.
            OFFSET 0
            DS.W 1 ; preserve Y
            DS.W 1 ; preserve X
            DS.W 1 ; return address
CN_NUM      DS.W 1 ;

countNum:   PSHX ; preserve register
            PSHY
            LDY #$0          ; retVal = 0;
            TFR D,X          ; set up X as intPt

CN_WHILE    ldd 0,x          ; while(*intPt != 0xFFFF)
            cpd #$FFFF      ; {
            beq CN_ENDWHILE
CN_IF        cpd CN_NUM,SP    ; if(num == *intPt)
            bne CN_ENDIF     ;         retVal++;
            iny
CN_ENDIF
            inx              ; intPt++
            inx
            bra CN_WHILE     ; }
CN_ENDWHILE

            tfr y,d          ; return(retVal);
            pulx             ; restore registers
            puly
            rts

```

Part 2 Theory (total 10 points)

(10 points) The HCS12 CPU uses the stack to support subroutines.

- Describe how the stack is used when calling and returning from subroutines. Please be complete.
- Provide a concrete example to illustrate.
- How would unbalanced PSH/PUL instructions in a subroutine affect the execution of the subroutine?

a) Description:

The stack is used to store the return address so that the CPU can find the return address when it reaches the end of the sub-routine.

The return address is found in the PC when either of these instructions are executed, that is, the PC points to the instruction following the JSR or BSR instruction. Thus the content of the PC is pushed onto the stack when a JSR or BSR instruction is executed.

When the RTS at the end of the subroutine is encountered, the PC is changed by pulling a 16-bit value from the stack. Thus the execution will continue from the instruction following the JSR or BSR used to jump into the subroutine.

b) Example:

Consider the following snippet of code:

```

$0810  xx xxxx JSR $0900
$0813  yy yy yy OPCODE OPERAND
.
.
.
.
$0900  OPCODE OPERAND
.
RTS

```

When JSR is executed, the return address \$0813 is pushed onto the stack. The PC will be updated with \$0900 to continue executing instructions from that point.

When RTS is executed, the return address \$0813 pulled from the stack and loaded into the PC. The CPU starts executing instructions from that point, that is, back to the instruction that follows the JSR used to call the subroutine.

c) Unbalanced PSH/PUL instructions:

Unbalanced psh/pul instructions leave the stack pointer pointing to an address location that does not contain the return address. Thus when the rts instruction is executed, the PC is not loaded with the appropriate address.

Part 3 – Application Question (total 20 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 positive 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.
- When characters are different, strings are not equal; 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*).
- 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 value 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

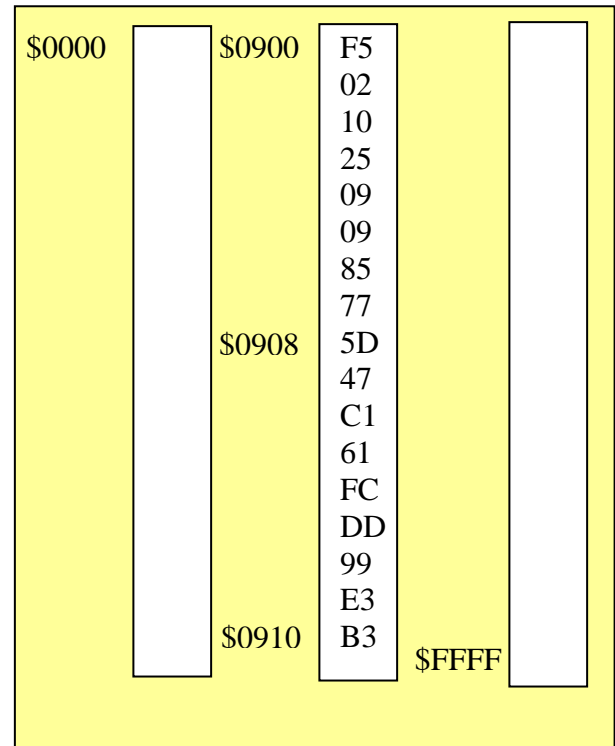
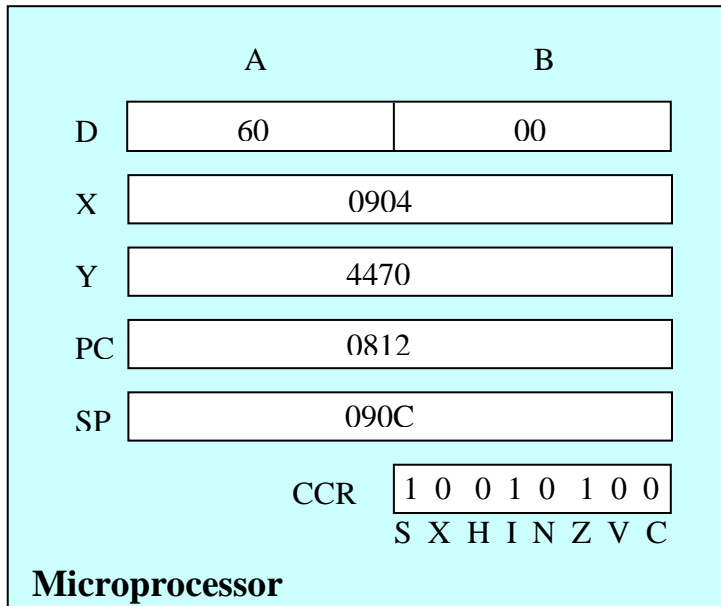
scmp_if:
        ldab 0, x ; if(*str1 != *str2)
        cmpb 0, y
        beq scmp_else
        subb 0, y
        stab SCMP_RETVAL, SP
        bra scmp_endwhile

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
```

Part 3 – continued

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		$m(ea) \leftarrow 0$
Clear Accumulator A (B)	CLRA (B)						X	$A \leftarrow 0$
Load Accumulator A (B)	LDAA (B)	X	X	X	X	X		$A \leftarrow [m(ea)]$
Load Double Accumulator D	LDD	X	X	X	X	X		$D \leftarrow [m(ea, ea+1)]$
Load Effective Address into SP (X or Y)	LEAS (A,B)							$SP \leftarrow ea$
Store Accumulator A (B)	STAA (B)	X	X	X	X	X		$m(ea) \leftarrow (A)$
Store Double Accumulator D	STD	X	X	X	X	X		$m(ea, ea+1) \leftarrow D$
Transfer A to B	TAB						X	$B \leftarrow (A)$
Transfer A to CCR	TAP						X	$CCR \leftarrow (A)$
Transfer B to A	TBA						X	$A \leftarrow B$
Transfer CCR to A	TPA						X	$A \leftarrow (CCR)$
Exchange D with X (Y)	XGDX						X	$D \leftrightarrow (X)$
Pull A (B) from Stack	PULA(B)						X	$A \leftarrow [m(SP)], SP \leftarrow (SP)+1$
Push A (B) onto Stack	PSHA(B)						X	$SP \leftarrow (SP)-1, m(SP) \leftarrow A$

Arithmetic Operations

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

Logical Operations

Function	Mnemonic	IMM	DIR	EXT	IDX	[IDX]	INH	Operation
AND A (B) with Memory	ANDA (B)	X	X	X	X	X		$A \leftarrow A \bullet [m(ea)]$
Bit(s) Test A (B) with Memory	BITA (B)	X	X	X	X	X		$A \bullet [m(ea)]$
One's Complement Memory Byte	COM			X	X	X		$m(ea) \leftarrow \sim [m(ea)]$
One's Complement A (B)	COMA (B)						X	$A \leftarrow \sim A$
OR A (B) with Memory (Exclusive)	EORA (B)	X	X	X	X	X		$A \leftarrow A \oplus [m(ea)]$
OR A (B) with Memory (Inclusive)	ORAA (B)	X	X	X	X	X		$A \leftarrow A + [m(ea)]$

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 → LBRA

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)) => CCR; (SP) + \$0001 => SP (M _(SP) : M _(SP+1)) => B : A; (SP) + \$0002 => SP (M _(SP) : M _(SP+1)) => X _H : X _L ; (SP) + \$0004 => SP (M _(SP) : M _(SP+1)) => PC _H : PC _L ; (SP) + \$0002 => SP (M _(SP) : M _(SP+1)) => Y _H : Y _L ; (SP) + \$0004 => SP
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.

Reference Guide for TST Instruction

TST <i>opr16a</i>	(M) – 0 Test Memory for Zero or Minus	EXT	F7 hh 11	rPO	rOP	----	ΔΔ00
TST <i>opr16_0_xysp</i>		IDX	B7 xb	rPf	rEP		
TST <i>opr16_9_xysp</i>		IDX1	B7 xb ff	rPO	rPO		
TST <i>opr16_16_xysp</i>		IDX2	B7 xb ee ff	frPP	frPP		
TST [D, <i>xysp</i>]		[D, IDX]	B7 xb	fIfxPf	fIfxEP		
TST [<i>opr16_16_xysp</i>]		[IDX2]	B7 xb ee ff	fIPxPf	fIPxEP		
TSTA	(A) – 0 Test A for Zero or Minus	INH	97	0	0		
TSTB	(B) – 0 Test B for Zero or Minus	INH	D7	0	0		

Reference Guide for JSR Instruction

Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail	HC12	S X H I	N Z V C
JSR <i>opr8a</i>	(SP) – 2 ⇒ SP; RTN _H , RTN _L ⇒ M _(SP) , M _(SP+1) ; Subroutine address ⇒ PC	DIR	17 dd	SPPP	PPPS	----	----
JSR <i>opr16a</i>		EXT	16 hh 11	SPPP	PPPS		
JSR <i>opr16_0_xysp</i>		IDX	15 xb	PPPS	PPPS		
JSR <i>opr16_9_xysp</i>		IDX1	15 xb ff	PPPS	PPPS		
JSR <i>opr16_16_xysp</i>		IDX2	15 xb ee ff	fPPPS	fPPPS		
JSR [D, <i>xysp</i>]		[D, IDX]	15 xb	fIfPPPS	fIfPPPS		
JSR [<i>opr16_16_xysp</i>]		[IDX2]	15 xb ee ff	fIfPPPS	fIfPPPS		

Post Byte Encoding, xb, for Indexed Addressing

Indexed Addressing Mode Postbyte Encoding (xb)

60	1, +Y pre-inc	70	1, Y+ post-inc	A0	1, +SP pre-inc	B0	1, SP+ post-inc	C0	0, PC 5b const	D0	-16, PC 5b const	E0	n, X 9b const	F0	n, SP 9b const
61	2, +Y pre-inc	71	2, Y+ post-inc	A1	2, +SP pre-inc	B1	2, SP+ post-inc	C1	1, PC 5b const	D1	-15, PC 5b const	E1	-n, X 9b const	F1	-n, SP 9b const
62	3, +Y pre-inc	72	3, Y+ post-inc	A2	3, +SP pre-inc	B2	3, SP+ post-inc	C2	2, PC 5b const	D2	-14, PC 5b const	E2	n, X 16b const	F2	n, SP 16b const
63	4, +Y pre-inc	73	4, Y+ post-inc	A3	4, +SP pre-inc	B3	4, SP+ post-inc	C3	3, PC 5b const	D3	-13, PC 5b const	E3	[n, X] 16b indir	F3	[n, SP] 16b indir
64	5, +Y pre-inc	74	5, Y+ post-inc	A4	5, +SP pre-inc	B4	5, SP+ post-inc	C4	4, PC 5b const	D4	-12, PC 5b const	E4	A, X A offset	F4	A, SP A offset
65	6, +Y pre-inc	75	6, Y+ post-inc	A5	6, +SP pre-inc	B5	6, SP+ post-inc	C5	5, PC 5b const	D5	-11, PC 5b const	E5	B, X B offset	F5	B, SP B offset
66	7, +Y pre-inc	76	7, Y+ post-inc	A6	7, +SP pre-inc	B6	7, SP+ post-inc	C6	6, PC 5b const	D6	-10, PC 5b const	E6	D, X D offset	F6	D, SP D offset
67	8, +Y pre-inc	77	8, Y+ post-inc	A7	8, +SP pre-inc	B7	8, SP+ post-inc	C7	7, PC 5b const	D7	-9, PC 5b const	E7	[D, X] D indirect	F7	[D, SP] D indirect
68	8, -Y pre-dec	78	8, Y- post-dec	A8	8, -SP pre-dec	B8	8, SP- post-dec	C8	8, PC 5b const	D8	-8, PC 5b const	E8	n, Y 9b const	F8	n, PC 9b const
69	7, -Y pre-dec	79	7, Y- post-dec	A9	7, -SP pre-dec	B9	7, SP- post-dec	C9	9, PC 5b const	D9	-7, PC 5b const	E9	-n, Y 9b const	F9	-n, PC 9b const
6A	6, -Y pre-dec	7A	6, Y- post-dec	AA	6, -SP pre-dec	BA	6, SP- post-dec	CA	CA 5b const	DA	-6, PC 5b const	EA	n, Y 16b const	FA	n, PC 16b const
6B	5, -Y pre-dec	7B	5, Y- post-dec	AB	5, -SP pre-dec	BB	5, SP- post-dec	CB	11, PC 5b const	DB	-5, PC 5b const	EB	[n, Y] 16b indir	FB	[n, PC] 16b indir
6C	4, -Y pre-dec	7C	4, Y- post-dec	AC	4, -SP pre-dec	BC	4, SP- post-dec	CC	12, PC 5b const	DC	-4, PC 5b const	EC	A, Y A offset	FC	A, PC A offset
6D	3, -Y pre-dec	7D	3, Y- post-dec	AD	3, -SP pre-dec	BD	3, SP- post-dec	CD	13, PC 5b const	DD	-3, PC 5b const	ED	B, Y B offset	FD	B, PC B offset
6E	2, -Y pre-dec	7E	2, Y- post-dec	AE	2, -SP pre-dec	BE	2, SP- post-dec	CE	14, PC 5b const	DE	-2, PC 5b const	EE	D, Y D offset	FE	D, PC D offset
6F	1, -Y pre-dec	7F	1, Y- post-dec	AF	1, -SP pre-dec	BF	1, SP- post-dec	CF	15, PC 5b const	DF	-1, PC 5b const	EF	[D, Y] D indirect	FF	[D, PC] D indirect

le 1

