This is a closed book exam. You must put your answers on pages 1, 2, 3, 4 only. You have 50 minutes, so allocate your time accordingly. Show your work, and put your answers in the boxes.

Please read the entire quiz before starting.

(5) Question 1. The format is 8-bit signed. What is the hexadecimal representation of the value -60?

(5) Question 2. When you add two 8-bit signed numbers an overflow error can occur. Which of the following techniques can be used to handle the problem of overflow? If there is more than one answer, give all answers that could work.
   A) Make it nonvolatile.
   B) Mask the data
   C) Make it friendly.
   D) Use interrupts.
   E) Implement ceiling and floor.
   F) Add drop out.
   G) Use promotion.
   H) Use demotion.
   I) Use unsigned math.

(5) Question 3. Consider the following two instructions

   ldab #250
   subb #-2

What will be the value of the overflow (V) bit?

What will be the value of the carry (C) bit?
(10) **Question 4.** Use a 7406 to interface an LED to PP5 of the 9S12. The desired operating point is 2.6V at 10 mA. At 10mA you can assume the $V_{OL}$ of the 7406 will be 0.4 V.

(10) **Question 5.** Consider the following piece of code that starts at `main`

```
$4000  CF4000           main  lds  #$4000
$4003  CE0008            ldx  #8
$4006  34                loop  pshx
$4007  0706              bsr  Sub1
$4009  31                puly
$400A  0435F9            dbne  x,loop
$400D  183E              stop
$400F  09                Sub1  dex
$4010  3D                rts
$FFFE  4000              fdb  main
```

Part a) Think about how this program executes up to and including the first execution of `dex`

Fill in specific hexadecimal bytes that are pushed on the stack.

Using an arrow, label to which box the SP points.

Your solution may or may not use all the boxes.

Part b) How many times is the subroutine called after reset and before stop?

(5) **Question 6.** Assume PC is $4000$, Register D is initially $1122$, and Register X is $2000$. You may assume all RAM locations are initially 0. Show the simplified bus cycles occurring when the `std` instruction is executed. In the “changes” column, specify which registers get modified during that cycle, and the corresponding new values. Do not worry about changes to the CCR. *Just show the one instruction.*

```
$4000  6C04          std  4,x
```

<table>
<thead>
<tr>
<th>R/W</th>
<th>Addr</th>
<th>Data</th>
<th>Changes to D,X,Y,S,PC,IR,EAR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
For questions 7, 8, and 9, don’t worry about establishing the reset vector, creating a main program, or initializing the stack pointer. You may use the following definitions:

```
PTP   equ  $0258
DDRP  equ  $025A
```

**Question 7.** Assume a positive logic switch is connected to PP1, and the direction register is properly initialized. Write assembly code that waits until the switch is pressed.

**Question 8.** Assume Buffer is an array of 100 16-bit numbers, located in RAM. Write assembly code that adds one to each element. Implement the following C (you can implement the result without making it precisely match the C code):

```
for(i=0; i<100; i++)
    Buffer[i] = Buffer[i]+1;
```
(20) **Question 9.** Assume Register A contains an 8-bit unsigned number, which is the input parameter to the subroutine. Assume Port P bit 5 is an output to an LED. Write an assembly language subroutine that tests Reg A, if it is greater than 100, turn on the LED, otherwise do not change the LED. Full credit will be given to a friendly solution.

```
;***** turn on LED if RegA>100 **************
;Inputs: RegA is an unsigned 8-bit number,
;Outputs: none
LEDout
```
aba 8-bit add RegA+RegB
abx unsigned add RegX+RegX+RegB
aby unsigned add RegY+RegY+RegB
adca 8-bit add with carry to RegA
addb 8-bit add with carry to RegB
adda 8-bit add to RegA
addb 8-bit add to RegB
addd 16-bit add to RegD
anda 8-bit logical and to RegA
andb 8-bit logical and to RegB
anddc 8-bit logical and to RegCC
asl/lsl 8-bit left shift Memory
asl/lsla 8-bit left shift RegA
aslb/lslb 8-bit left shift RegB
asl/lsl 8-bit left shift Memory
asrb 8-bit arith right shift to RegB
asra 8-bit arith right shift to RegA
asr 8-bit arith right shift Memory
asld/lsld 16-bit left shift RegD
aslb/lslb 8-bit left shift RegB
asld/lsls 16-bit left shift RegD
asr 8-bit arith right shift Memory
asra 8-bit arith right shift to RegA
asrb 8-bit arith right shift to RegB
bcc branch if carry clear
bcir bit clear in memory
bcrl PTT, #01
bcs branch if carry set
beq branch if result is zero (Z=1)
beqg enter background debug mode
beqh branch if incremented
behi branch if unsigned >
bhs branch if unsigned ≥
bhs branch if unsigned ≥
bil 8-bit and with RegA, sets CCR
bime branch if result is negative (N=1)
bmne branch if result is nonzero (Z=0)
blp 8-bit positive
bra branch always
bcrlr branch if bits are clear
bcrl PTT, #01, loop
brn branch never
brset branch if bits are set
bset PTT, #01, loop
bls branch if signed ≤
bls branch if signed ≤
blt branch if signed <
blt branch if signed <
blmi branch if result is negative (N=1)
bmsn branch if result is nonzero (Z=0)
bpl branch if result is positive (N=0)
br branch always
brclr branch if bits are clear
brclr PTT, #01, loop
brn branch never
brset branch if bits are set
bset PTT, #01, loop
bls branch if signed ≤
bls branch if signed ≤
blt branch if signed <
blt branch if signed <
blmi branch if result is negative (N=1)
bmsn branch if result is nonzero (Z=0)
bpl branch if result is positive (N=0)
br branch always
bcrlr branch if bits are clear
bcrl PTT, #01, loop
brn branch never
brset branch if bits are set
bset PTT, #01, loop
bls branch if signed ≤
bls branch if signed ≤
blt branch if signed <
blt branch if signed <
blmi branch if result is negative (N=1)
bmsn branch if result is nonzero (Z=0)
bpl branch if result is positive (N=0)
br branch always
bcrlr branch if bits are clear
bcrl PTT, #01, loop
brn branch never
brset branch if bits are set
bset PTT, #01, loop
bls branch if signed ≤
bls branch if signed ≤
blt branch if signed <
blt branch if signed <
blmi branch if result is negative (N=1)
bmsn branch if result is nonzero (Z=0)
bpl branch if result is positive (N=0)
br branch always
bcrlr branch if bits are clear
bcrl PTT, #01, loop
brn branch never
brset branch if bits are set
bset PTT, #01, loop
bls branch if signed ≤
bls branch if signed ≤
blt branch if signed <
blt branch if signed <
blmi branch if result is negative (N=1)
bmsn branch if result is nonzero (Z=0)
bpl branch if result is positive (N=0)
br branch always
bcrlr branch if bits are clear
bcrl PTT, #01, loop
brn branch never
brset branch if bits are set
bset PTT, #01, loop
bls branch if signed ≤
bls branch if signed ≤
blt branch if signed <
blt branch if signed <
blmi branch if result is negative (N=1)
bmsn branch if result is nonzero (Z=0)
bpl branch if result is positive (N=0)
br branch always
bcrlr branch if bits are clear
bcrl PTT, #01, loop
brn branch never
brset branch if bits are set
bset PTT, #01, loop
bls branch if signed ≤
bls branch if signed ≤
blt branch if signed <
blt branch if signed <
blmi branch if result is negative (N=1)
oraa 8-bit logical or to RegA
orab 8-bit logical or to RegB
orcc 8-bit logical or to RegCC
psha push 8-bit RegA onto stack
pslb push 8-bit RegB onto stack
pslh push 16-bit RegD onto stack
pslx push 16-bit RegX onto stack
psly push 16-bit RegY onto stack
rev Fuzzy logic rule evaluation
revw weighted Fuzzy rule evaluation
rol 8-bit roll shift left Memory
rola 8-bit roll shift left RegA
rolb 8-bit roll shift left RegB
ror 8-bit roll shift right Memory
rorb 8-bit roll shift right RegB
rtc return sub in expanded memory
rti return from interrupt
rts return from subroutine
sba 8-bit subtract RegA-RegB
sbca 8-bit sub with carry from RegA
sbcb 8-bit sub with carry from RegB
sec set carry bit, C=1
sei set I=1, disable interrupts
sev set overflow bit, V=1
sex sign extend 8-bit to 16-bit reg
sex B,D
staa 8-bit store memory from RegA
stab 8-bit store memory from RegB
std 16-bit store memory from RegD
sts 16-bit store memory from SP
stx 16-bit store memory from RegX
sty 16-bit store memory from RegY
suba 8-bit sub from RegA
subb 8-bit sub from RegB
subd 16-bit sub from RegD
swi software interrupt, trap
tab transfer A to B
tap transfer A to CC
tba transfer B to A
tbeq test and branch if result=0
tbl 8-bit look up and interpolation
tbeq Y,loop
tn e test and branch if result=0
tna A,loop
tfr transfer register to register
tpa transfer CC to A
tst 8-bit compare memory with zero
tsta 8-bit compare RegA with zero
tstb 8-bit compare RegB with zero
tsx transfer S to X
tsy transfer Y to S
wai wait for interrupt
wav weighted Fuzzy logic average
xgdx exchange RegD with RegX
xgdy exchange RegD with RegY

---

element addressing

<table>
<thead>
<tr>
<th>Pseudo op</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>org</td>
<td>Specific absolute address to put subsequent object code</td>
</tr>
<tr>
<td>=</td>
<td>Define a constant symbol</td>
</tr>
<tr>
<td>set</td>
<td>Define or redefine a constant symbol</td>
</tr>
<tr>
<td>dc.b</td>
<td>Allocate byte(s) of storage with initialized values</td>
</tr>
<tr>
<td>db</td>
<td>Create an ASCII string (no termination character)</td>
</tr>
<tr>
<td>fcb .byte</td>
<td>Allocate word(s) of storage with initialized values</td>
</tr>
<tr>
<td>fcc</td>
<td>Allocate 32-bit long word(s) of storage with initialized values</td>
</tr>
<tr>
<td>dw</td>
<td>Allocate byte(s) of storage without initialization</td>
</tr>
<tr>
<td>fdb .word</td>
<td>Allocate byte(s) of storage without initialization</td>
</tr>
<tr>
<td>dl .long</td>
<td>Allocate 32-bit long word(s) of storage without initialization</td>
</tr>
<tr>
<td>ds .blkb</td>
<td>Allocate byte(s) of storage without initialization</td>
</tr>
<tr>
<td>ds .blkw</td>
<td>Allocate byte(s) of storage without initialization</td>
</tr>
<tr>
<td>ds .l .blkl</td>
<td>Allocate 32-bit words of storage without initialization</td>
</tr>
</tbody>
</table>
STD

Store Double Accumulator

Operation: \((A : B) \Rightarrow M : M + 1\)

Description: Stores the content of double accumulator D in memory location
M : M + 1. The content of D is unchanged.

<table>
<thead>
<tr>
<th>Source Form</th>
<th>Address Mode</th>
<th>Object Code</th>
<th>HCS12 Access Detail</th>
</tr>
</thead>
<tbody>
<tr>
<td>STD opr8a</td>
<td>DIR</td>
<td>5C dd</td>
<td>PW</td>
</tr>
<tr>
<td>STD opr16a</td>
<td>EXT</td>
<td>7C hh 11</td>
<td>PWO</td>
</tr>
<tr>
<td>STD oprx0_xyssp</td>
<td>IDX</td>
<td>6C xb</td>
<td>PW</td>
</tr>
<tr>
<td>STD oprx9,xyssp</td>
<td>IDX1</td>
<td>6C xb ff</td>
<td>PWO</td>
</tr>
<tr>
<td>STD oprx16,xyssp</td>
<td>IDX2</td>
<td>6C xb ee ff</td>
<td>PWP</td>
</tr>
</tbody>
</table>

BSR

Branch to Subroutine

Operation: \((SP) - $0002 \Rightarrow SP\)

RTNH : RTNL \(\Rightarrow M(SP) : M(SP+1)\)

(\(PC\) + Rel \(\Rightarrow PC\)

Description: Sets up conditions to return to normal program flow, then transfers control to a subroutine. Uses the address of the instruction after the BSR as a return address. Decrements the SP by two, to allow the two bytes of the return address to be stacked. Stacks the return address (the SP points to the high-order byte of the return address). Branches to a location determined by the branch offset. Subroutines are normally terminated with an RTS instruction, which restores the return address from the stack.

<table>
<thead>
<tr>
<th>Source Form</th>
<th>Address Mode</th>
<th>Object Code</th>
<th>Access Detail HCS12</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSR rel8</td>
<td>REL</td>
<td>07 rr</td>
<td>SPPP</td>
</tr>
</tbody>
</table>

RTS

Return from Subroutine

Operation: \((M(SP) : M(SP+1)) \Rightarrow PCH : PCL ; (SP) + $0002 \Rightarrow SP\)

Description: Restores context at the end of a subroutine. Loads the program counter with a 16-bit value pulled from the stack and increments the stack pointer by two. Program execution continues at the address restored from the stack.

<table>
<thead>
<tr>
<th>Source Form</th>
<th>Address Mode</th>
<th>Object Code</th>
<th>Access Detail HCS12</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTS</td>
<td>INH</td>
<td>3D</td>
<td>UfPPP</td>
</tr>
</tbody>
</table>
**SUBB**  
Subtract B

**Operation:**  \((B) - (M) \Rightarrow B\)

**Description:** Subtracts the content of memory location \(M\) from the content of accumulator \(B\) and places the result in \(B\). For subtraction instructions, the \(C\) status bit represents a borrow.

- **N:** Set if MSB of result is set; cleared otherwise
- **Z:** Set if result is $00$; cleared otherwise
- **V:** \(B_7 \cdot M_7 \cdot R_7 + B_7 \cdot M_7 \cdot R_7\)
  Set if a two's complement overflow resulted from the operation; cleared otherwise
- **C:** \(B_7 \cdot M_7 + M_7 \cdot R_7 + R_7 \cdot B_7\)
  Set if the value of the content of memory is larger than the value of the accumulator; cleared otherwise

<table>
<thead>
<tr>
<th>Source Form</th>
<th>Address Mode</th>
<th>Object Code</th>
<th>Access Detail</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBB #opr&amp;i</td>
<td>IMM</td>
<td>C0 ii</td>
<td>P</td>
</tr>
<tr>
<td>SUBB opr&amp;a</td>
<td>DIR</td>
<td>D0 dd</td>
<td>rPf</td>
</tr>
<tr>
<td>SUBB opr16a</td>
<td>EXT</td>
<td>F0 hh ll</td>
<td>rPPO</td>
</tr>
<tr>
<td>SUBB oprx0_xysp</td>
<td>IDX</td>
<td>E0 xb</td>
<td>rPf</td>
</tr>
<tr>
<td>SUBB oprx9_xysp</td>
<td>IDX1</td>
<td>E0 xb ff</td>
<td>rPPO</td>
</tr>
<tr>
<td>SUBB oprx16_xysp</td>
<td>IDX2</td>
<td>E0 xb ee ff</td>
<td>rPP</td>
</tr>
<tr>
<td>SUBB [D,xysp]</td>
<td>[D,IDX]</td>
<td>E0 xb</td>
<td>fIFrPf</td>
</tr>
<tr>
<td>SUBB [opr16_xysp]</td>
<td>[IDX2]</td>
<td>E0 xb ee ff</td>
<td>fIPrPf</td>
</tr>
</tbody>
</table>