This is a closed book exam. You must put your answers on this piece of paper 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. Assume an 8-bit signed integer format. Give the 8-bit binary representation of the value -88.

(5) Question 2. When you add two 8-bit unsigned 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) Write software that is friendly.
B) Write software using structured programming.
C) Write software so there is drop out.
D) Implement a ceiling and floor.
E) Write software to mask the two input data
F) Promote the numbers and perform the addition with this new precision.
G) Demote the numbers and perform the addition with this new precision.
H) Convert the numbers to signed and perform the addition with signed math.

(5) Question 3. Consider the following two instructions

```
ldab #10
subb #200
```

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

What will be the value of the carry (C) bit?

What will be the value of the negative (N) bit?
(5) Question 4. Consider the result of executing the following three 9S12 assembly instructions.

```
lda a #100
lda b #3
mul
```

What is the value in Register B after these three instructions are executed?

(5) Question 5. We are designing an ohmmeter that measures resistance in the range of 0 to 3,999Ω with a resolution of 1 Ω. What is the precision of this system in decimal digits?

(5) Question 6. How many bus cycles does it take to execute `bsr` on a real 9S12?

(10) Question 7. Assume PC is $6000, and Register Y is initially $2233. You may assume location $0812 and $0813 are initially 0. Show the simplified bus cycles occurring when the `sty` 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.`

```
$6000 7D0812    sty $0812
```

<table>
<thead>
<tr>
<th>R/W</th>
<th>Addr</th>
<th>Data</th>
<th>Changes to A,B,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>

(15) Question 8. Draw the circuit diagram interfacing two positive logic switches to PP7 and PP6. Please specify the resistance values for the resistors and the chip numbers (e.g., 7406) for any digital logic you use. You may assume PP7 and PP6 are inputs.

```
PP7
PP6
9S12
```
For questions 9 and 10, you will write three subroutines. Don’t worry about establishing the reset vector, creating a main program, calling the subroutines or initializing the stack pointer.

(25) Question 9. You may use these definitions. Data is a signed 8 bit variable.

PTT   equ  $0240
DDRT  equ  $0242
org  $0800 ; RAM
Data  rmb  1 ; signed 8-bit
org $4000 ; ROM

Part a) Write an assembly subroutine that makes PT7 an output and Data equal to 100. Comments are required. Please make the software friendly.

Part b) Write an assembly subroutine that sets PT7 to 1 if Data>25, and does not change PT7 if Data≤25. Comments are required. Please make the software friendly.

(20) Question 10. You will write a subroutine with two 8-bit unsigned inputs and one 8-bit unsigned output. The inputs are passed in using RegA and RegB. Find the maximum of these two unsigned numbers and return the result in RegA. (Do not try to use the maxa instruction.)

;*****Max subroutine**********************
; Inputs:  RegA is the first number, RegB is the second number
; Outputs: RegA is the maximum of first and second
Max
aba 8-bit add RegA=RegA+RegB
abx unsigned add RegX=RegX+RegB
aby unsigned add RegY=RegY+RegB
adca 8-bit add with carry to RegA
addc 8-bit add with carry to RegB
adda 8-bit add to RegA
addb 8-bit add to RegB
adddd 16-bit add to RegD
anda 8-bit logical and to RegA
andb 8-bit logical and to RegB
andcc 8-bit logical and to RegCC
asl/lsl 8-bit left shift Memory
aslb/lslb 8-bit arith left shift RegB
asla/lsla 8-bit left shift RegA
asl/lsl 8-bit arith left shift RegB
bclr PTT,#$01 ;clears PT0=0
bclr bit clear in memory
bcs branch if carry set
bitb 8-bit and with RegB, sets CCR
bita 8-bit and with RegA, sets CCR
bmi branch if result is negative (N=1)
bmi branch if result is negative (N=1)
bpl branch if result is positive (N=0)
brtab always
brcc branch if carry clear
brclr branch if bits are clear
brs set bit in memory
bset bit set in memory
bset PTT,#$01 ;sets PT2=0
bset PTT,#$04 ;sets PT2=1
bcnx branch never
bvc branch if overflow clear
bvs branch if overflow set
call subroutine in expanded memory
cbr branch to subroutine
cbz branch if zeros
ccf clear carry bit, C=0
clcr clear memory
clra RegA=0 clear
clrb RegB=0 clear
clv clear overflow bit, V=0
cmpeq 8-bit compare RegA with memory
cmpeq 8-bit compare RegB with memory
cmpeq 8-bit logical complement to RegA
cmpeq 8-bit logical complement to RegB
cmpeq 8-bit logical complement to RegX with memory
cmpeq 16-bit compare RegX with memory
cmpx 8-bit logical and to RegA
cmpx 8-bit logical and to RegB
cmpx 8-bit logical and to RegX
clcr 8-bit memory clear
cmpa 8-bit compare RegA with memory
cmpb 8-bit compare RegB with memory
cmpg 8-bit logical complement to RegG
cmpg 8-bit logical complement to RegH
cmpg 8-bit logical complement to RegX
cmpx 16-bit compare RegX with memory
cpy 16-bit compare RegY with memory
ctype 16-bit compare RegY with memory
dac 8-bit decrement RegA
dacm 8-bit decrement RegB
dacmd 8-bit decrement RegD
adc 16-bit decrement RegSP
dax 16-bit decrement RegX
dbne decrement and branch if result≠0
dbne A,loop
dbeq increment and branch if result=0
dbeq Y,loop
dbeq increment and branch if result=0
dbeq A,loop
dbeq increment and branch if result=0
dbeq A,loop
dbeq increment and branch if result=0
dbeq A,loop
dbeq increment and branch if result=0
dbeq A,loop
dbeq increment and branch if result=0
dbeq A,loop
dbeq increment and branch if result=0
dbeq A,loop
dbeq increment and branch if result=0
dbeq A,loop
dbeq increment and branch if result=0
dbeq A,loop
dbeq increment and branch if result=0
dbeq A,loop
dbeq increment and branch if result=0
dbeq A,loop
dbeq increment and branch if result=0
dbeq A,loop
dbeq increment and branch if result=0
dbeq A,loop
dbeq increment and branch if result=0
dbeq A,loop
dbeq increment and branch if result=0
dbeq A,loop
dbeq increment and branch if result=0
dbeq A,loop
dbeq increment and branch if result=0
dbeq A,loop
dbeq increment and branch if result=0
dbeq A,loop
dbeq increment and branch if result=0
dbeq A,loop
dbeq increment and branch if result=0
dbeq A,loop

<table>
<thead>
<tr>
<th>oraa</th>
<th>8-bit logical or to RegA</th>
</tr>
</thead>
<tbody>
<tr>
<td>orab</td>
<td>8-bit logical or to RegB</td>
</tr>
<tr>
<td>orcc</td>
<td>8-bit logical or to RegCC</td>
</tr>
<tr>
<td>psha</td>
<td>push 8-bit RegA onto stack</td>
</tr>
<tr>
<td>pshb</td>
<td>push 8-bit RegB onto stack</td>
</tr>
<tr>
<td>pshc</td>
<td>push 8-bit RegCC onto stack</td>
</tr>
<tr>
<td>pshd</td>
<td>push 16-bit RegD onto stack</td>
</tr>
<tr>
<td>pshx</td>
<td>push 16-bit RegX onto stack</td>
</tr>
<tr>
<td>pshy</td>
<td>push 16-bit RegY onto stack</td>
</tr>
<tr>
<td>pula</td>
<td>pop 8 bits off stack into RegA</td>
</tr>
<tr>
<td>pulb</td>
<td>pop 8 bits off stack into RegB</td>
</tr>
<tr>
<td>pulc</td>
<td>pop 8 bits off stack into RegCC</td>
</tr>
<tr>
<td>puld</td>
<td>pop 16 bits off stack into RegD</td>
</tr>
<tr>
<td>pulx</td>
<td>pop 16 bits off stack into RegX</td>
</tr>
<tr>
<td>puly</td>
<td>pop 16 bits off stack into RegY</td>
</tr>
<tr>
<td>rev</td>
<td>Fuzzy logic rule evaluation</td>
</tr>
<tr>
<td>revw</td>
<td>weighted Fuzzy rule evaluation</td>
</tr>
<tr>
<td>rol</td>
<td>8-bit roll shift left Memory</td>
</tr>
<tr>
<td>rola</td>
<td>8-bit roll shift left RegA</td>
</tr>
<tr>
<td>rolb</td>
<td>8-bit roll shift left RegB</td>
</tr>
<tr>
<td>ror</td>
<td>8-bit roll shift right Memory</td>
</tr>
<tr>
<td>rora</td>
<td>8-bit roll shift right RegA</td>
</tr>
<tr>
<td>rorb</td>
<td>8-bit roll shift right RegB</td>
</tr>
<tr>
<td>rtc</td>
<td>return sub in expanded memory</td>
</tr>
<tr>
<td>rti</td>
<td>return from interrupt</td>
</tr>
<tr>
<td>rts</td>
<td>return from subroutine</td>
</tr>
<tr>
<td>sha</td>
<td>8-bit subtract RegA-RegB</td>
</tr>
<tr>
<td>sbca</td>
<td>8-bit subtract with carry from RegA-RegB</td>
</tr>
<tr>
<td>sbcb</td>
<td>8-bit sub with carry from RegB</td>
</tr>
<tr>
<td>sec</td>
<td>set carry bit, C=1</td>
</tr>
<tr>
<td>sel</td>
<td>set 1=1, disable interrupts</td>
</tr>
<tr>
<td>sex</td>
<td>sign extend 8-bit to 16-bit reg</td>
</tr>
<tr>
<td>sex B,D</td>
<td>extend B into D</td>
</tr>
</tbody>
</table>

**Example Addressing Modes**

<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>= equ</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 db fcb .byte</td>
<td>Allocate byte(s) of storage with initialized values</td>
</tr>
<tr>
<td>fcc</td>
<td>Create an ASCII string (no termination character)</td>
</tr>
<tr>
<td>dc.w dw fdb .word</td>
<td>Allocate word(s) of storage with initialized values</td>
</tr>
<tr>
<td>dc.l dl .long</td>
<td>Allocate 32-bit long word(s) of storage with initialized values</td>
</tr>
<tr>
<td>ds ds.b rmb .blkb</td>
<td>Allocate bytes of storage without initialization</td>
</tr>
<tr>
<td>ds.w .blkw</td>
<td>Allocate bytes 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>
**STY**  
**Store Index Register Y**

**Operation:**  
\[(YH : YL) \Rightarrow M : M + 1\]

- **N:** Set if MSB of result is set; cleared otherwise
- **Z:** Set if result is $0000$; cleared otherwise
- **V:** 0; cleared

**Description:** Stores the content of index register Y in memory. The most significant byte of Y is stored at the specified address, and the least significant byte of Y is stored at the next higher byte address (the specified address plus one).

<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>STY opr16a</td>
<td>EXT</td>
<td>7D hh ll</td>
<td>PWO</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. Decrement 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>