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. *Please read the entire quiz before starting.*

(5) Question 1.

(5) Question 2.

(5) Question 3.

(5) Question 4.

(5) Question 5.

(5) Question 6.

(5) Question 7.

(5) Question 8.

(5) Question 9.

(10) Question 10.

Pulse bset PTT,#1 ; send pulse on PT0

    bclr PTT,#1

    rts
(15) Question 11.

Convert

(5) Question 12.

(calc)

(15) Question 13.

;short calc(void){

;short sum,n;

;n = 100;

;sum = 0;

;do{ sum = sum+n;

;} while(--n);

;return(sum);}
This Fifo queue can hold up to eight 8-bit data values, and the picture shows it currently is holding three values (shaded).

(5) Question 1. What value is returned if we were to call \texttt{Fifo\_Get} at this point?

(5) Question 2. Next, assume we call \texttt{Fifo\_Put}. What will be the new \texttt{PutPt} after we call \texttt{Fifo\_Put}?

Questions 3 and 4 involve the following assembly program involving a stack frame.

```assembly
main
lds #$4000
ldaa #100
psha    ; pass 8-bit in parameter on stack
jsr sub2
leas 1,s  ; balance stack
here bra here

data set xxx  ; binding of 8-bit local variable
in set yyy   ; binding of 8-bit input parameter
sub2 pshx    ; save register X
des         ; allocate 8-bit local variable called data
tsx         ; RegX stack frame

;****body of the subroutine
ldab in,x   ; get a copy of in parameter
stab data,x ; store into local variable data

;****end of body
ins         ; deallocate data
pulx        ; restore register X
rts         ; return
```

(5) Question 3. What value should you use in the \texttt{xxx} position to implement the binding of the local variable, \texttt{data}?

(5) Question 4. What value should you use in the \texttt{yyy} position to implement the binding of the parameter, \texttt{in}?

(5) Question 5. Specify the resistor value for $R_1$, assuming LED current $I_d$ is 1 mA, the LED voltage $V_d$ is 2.5 V, and the gate output voltage $V_{OL}$ is 0.5V.

(5) Question 6. Which three events cause an interrupt to occur? Specify three letters in any order.
A) The software disarms the interrupt (e.g., RTIE=0)
B) The I bit in the CCR is set
C) The I bit in the CCR is clear
D) The software arms the interrupt (e.g., RTIE=1)
E) The software acknowledges the interrupt, clearing the flag (e.g., RTIF=0)
F) The software sets the flag bit (e.g., RTIF=1)
G) The hardware sets the flag bit (e.g., RTIF=1)
H) The hardware acknowledges the interrupt, clearing the flag (e.g., RTIF=0)
(5) **Question 7.** Consider a 10-bit ADC with a range of -10 to +10V. What is the approximate resolution of this ADC? Give units.

(5) **Question 8.** The 0 to 5V 10-bit ADC on the 9S12C32 uses the successive approximation conversion technique. This technique involves a series of guesses. Which will be the first guess?
   A) 0 V
   B) 5V/1024 = 5mV
   C) 5V/256 = 20mV
   D) 1.25 V
   E) 2.5 V
   F) 3.75 V
   G) 5 V

(5) **Question 9.** The following ISR has a bug (I know it doesn’t do anything):

```
;******called when RTIF is set  ****************
RTIhandler
   ldaa PTT
   sei
   rti
   org $FFF0
   fdb RTIhandler
```

A) Register A is altered by the ISR, so the main program will be confused.
B) The `sei` instruction disables interrupts, so no more interrupts will occur
C) This ISR did not acknowledge the interrupt (clear RTIF), so it will interrupt over and over continuously.
D) This ISR did not acknowledge the interrupt (clear RTIF) so no more interrupts will occur.
E) The stack is unbalanced, so it will crash.
F) The ISR didn’t need to set the I bit with the `sei`, because the `rti` instruction will automatically set the I bit when the handler returns.

(15) **Question 10.** A complicated software system includes this subroutine

```
Pulse bset PTT,#1 ; send pulse on PT0
   bclr PTT,#1
   rts
```

Design a minimally intrusive debugging instrument that will allow you to measure how many times this subroutine has been called. You may assume that this subroutine is called less than 1000 times. Include global variable definitions, an initialization subroutine and the instrument added to `Pulse` that counts. No comments required for this question.

(15) **Question 11.** Write a subroutine that converts a 10-bit ADC sample into a position. The input parameter is passed call by reference in Reg Y, meaning Reg Y contains a pointer to the input 16-bit input data. The output parameter should be returned by value in RegD, meaning Reg D itself contains the result. The conversion is a linear function (Output = 2.5*Input+100). The range of input values is 0 to 1023. The output range is 100 to 2556. The following main program illustrates how data is passed into and out of your subroutine. *Comments are required.*
The following table shows some example data. (don’t worry about rounding the LSB)

<table>
<thead>
<tr>
<th>Input</th>
<th>2.5*Input</th>
<th>Output</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>100</td>
<td>1.00 cm</td>
</tr>
<tr>
<td>100</td>
<td>250</td>
<td>350</td>
<td>3.50 cm</td>
</tr>
<tr>
<td>1000</td>
<td>2500</td>
<td>2600</td>
<td>26.00 cm</td>
</tr>
<tr>
<td>1022</td>
<td>2555</td>
<td>2655</td>
<td>26.55 cm</td>
</tr>
</tbody>
</table>

(hint: you can solve this problem one of two ways. First, you could use the fdiv instruction. Second, you could rewrite the formula as Output = 2*Input+Input/2+100.

(5) Question 12. A serial port will be used to transfer 2000 bytes of information per second. The protocol is 1 start bit, 8 data bits, and 2 stop bits. What is the slowest baud rate that can handle this serial transfer?

(20) Question 13. Translate explicitly (line by line) the following C program to assembly. Both variables (n, sum) must be stored on the stack, including symbolic binding. For each line of C, fetch necessary values off the stack, operate, and store back to the stack as appropriate. For example, sum = sum+n; should be implemented as read sum from stack, read n from stack, add, write result back to stack. The output parameter should be returned by value in Reg D. No additional comments required for this question.

```c
short calc(void) {
    short sum, n; // two 16-bit signed variables
    n = 100;
    sum = 0;
    do { sum = sum + n; } while (n > 0);  // means decrement and branch back if not zero
    return (sum);  // return a 16-bit result in Reg D
}
```
orcc 8-bit logical or to RegCC
psha push 8-bit RegA onto stack
pslb push 8-bit RegB onto stack
pslc push 8-bit RegC onto stack
psld push 8-bit RegD onto stack
pula pop 8 bits off stack into RegA
pulb pop 8 bits off stack into RegB
pulc pop 8 bits off stack into RegC
puld pop 16 bits off stack into RegD
pulx pop 16 bits off stack into RegX
puly pop 16 bits off stack into RegY
rev Fuzzy logic rule evaluation
revw weighted Fuzzy rule evaluation
rol 8-bit roll shift left Memory
rolla 8-bit roll shift left RegA
rolb 8-bit roll shift left RegB
ror 8-bit roll shift right Memory
rorA 8-bit roll shift right RegA
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
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
tba transfer B to A
tbeq test and branch if result=0
tbeq Y,loop
tbl 8-bit look up and interpolation
tbne test and branch if result≠0
tfr transfer register to register
tpa transfer CC to A
trp illegal instruction interrupt
trc return from interrupt
tst 8-bit compare memory with zero
tsa 8-bit compare RegA with zero
tsb 8-bit compare RegB with zero
tsx transfer S to X
tsy transfer S to Y
txs transfer X to S
tfr transfer register to register
wave weighted Fuzzy logic average
xgdx exchange RegD with RegX
xgdy exchange RegD with RegY

Example addressing mode

<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>Allocate byte(s) of storage with initialized values</td>
</tr>
<tr>
<td>fcb</td>
<td>Create an ASCII string (no termination character)</td>
</tr>
<tr>
<td>fcc</td>
<td>Allocate word(s) of storage with initialized values</td>
</tr>
<tr>
<td>dc.w</td>
<td>Allocate word(s) of storage with initialized values</td>
</tr>
<tr>
<td>dw</td>
<td>Allocate word(s) of storage with initialized values</td>
</tr>
<tr>
<td>fdb</td>
<td>Allocate 32-bit long word(s) of storage with initialized values</td>
</tr>
<tr>
<td>.byte</td>
<td>Allocate bytes of storage without initialization</td>
</tr>
<tr>
<td>.word</td>
<td>Allocate bytes of storage without initialization</td>
</tr>
<tr>
<td>.long</td>
<td>Allocate 32-bit words of storage without initialization</td>
</tr>
<tr>
<td>rmb</td>
<td>Allocate 32-bit words of storage without initialization</td>
</tr>
</tbody>
</table>

Freescale 6812 addressing modes

<table>
<thead>
<tr>
<th>Pseudo op</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldwa #u</td>
<td>Immediate</td>
</tr>
<tr>
<td>ldwa u</td>
<td>Direct</td>
</tr>
<tr>
<td>ldwa U</td>
<td>Extended</td>
</tr>
<tr>
<td>ldwa m,r</td>
<td>5-bit index</td>
</tr>
<tr>
<td>ldwa v,+r</td>
<td>pre-increment</td>
</tr>
<tr>
<td>ldwa v,-r</td>
<td>de-increment</td>
</tr>
<tr>
<td>ldwa v,r</td>
<td>post-increment</td>
</tr>
<tr>
<td>ldwa a,r</td>
<td>Reg A offset</td>
</tr>
<tr>
<td>ldwa b,r</td>
<td>Reg B offset</td>
</tr>
<tr>
<td>ldwa d,r</td>
<td>Reg D offset</td>
</tr>
<tr>
<td>ldwa q,r</td>
<td>9-bit index</td>
</tr>
<tr>
<td>ldwa [0,]</td>
<td>Indirect</td>
</tr>
<tr>
<td>ldwa [D,r]</td>
<td>Indirect</td>
</tr>
<tr>
<td>ldwa [W,r]</td>
<td>Indirect</td>
</tr>
<tr>
<td>ldwa [W+r]</td>
<td>Indirect</td>
</tr>
<tr>
<td>ldwa [W+W]</td>
<td>Indirect</td>
</tr>
</tbody>
</table>

<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>Allocate byte(s) of storage with initialized values</td>
</tr>
<tr>
<td>fcb</td>
<td>Create an ASCII string (no termination character)</td>
</tr>
<tr>
<td>fcc</td>
<td>Allocate word(s) of storage with initialized values</td>
</tr>
<tr>
<td>dc.w</td>
<td>Allocate word(s) of storage with initialized values</td>
</tr>
<tr>
<td>dw</td>
<td>Allocate word(s) of storage with initialized values</td>
</tr>
<tr>
<td>fdb</td>
<td>Allocate 32-bit long word(s) of storage with initialized values</td>
</tr>
<tr>
<td>.byte</td>
<td>Allocate bytes of storage without initialization</td>
</tr>
<tr>
<td>.word</td>
<td>Allocate bytes of storage without initialization</td>
</tr>
<tr>
<td>.long</td>
<td>Allocate 32-bit words of storage without initialization</td>
</tr>
<tr>
<td>rmb</td>
<td>Allocate 32-bit words of storage without initialization</td>
</tr>
</tbody>
</table>
FDIV

Fractional Divide

Operation: \((D) \div (X) = X; \text{Remainder} = D\)

Description: Divides an unsigned 16-bit numerator in double accumulator \(D\) by an unsigned 16-bit denominator in index register \(X\), producing an unsigned 16-bit quotient in \(X\) and an unsigned 16-bit remainder in \(D\). If both the numerator and the denominator are assumed to have radix points in the same positions, the radix point of the quotient is to the left of bit 15. The numerator must be less than the denominator. In the case of overflow (denominator is less than or equal to the numerator) or division by zero, the quotient is set to $FFFF$, and the remainder is indeterminate.

FDIV is equivalent to multiplying the numerator by \(2^{16}\) and then performing 32 by 16-bit integer division. The result is interpreted as a binary-weighted fraction, which resulted from the division of a 16-bit integer by a larger 16-bit integer. A result of $00001$ corresponds to 0.000015, and $FFFF$ corresponds to 0.9998. The remainder of an IDIV instruction can be resolved into a binary-weighted fraction by an FDIV instruction. The remainder of an FDIV instruction can be resolved into the next 16 bits of binary-weighted fraction by another FDIV instruction.

<table>
<thead>
<tr>
<th>S</th>
<th>X</th>
<th>H</th>
<th>I</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
</tr>
</tbody>
</table>

Z: Set if quotient is $0000$; cleared otherwise

V: \(1\) if \(X \leq D\)

Set if the denominator was less than or equal to the numerator; cleared otherwise

C: \(X_{15} \bullet X_{14} \bullet X_{13} \bullet X_{12} \bullet \ldots \bullet X_{3} \bullet X_{2} \bullet X_{1} \bullet X_{0}\)

Set if denominator was $0000$; cleared otherwise