First:_____ Middle Initial: _____ Last:_____

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.	
(5) Question 10.	
(5) Question 11.	
(5) Question 12.	
(5) Question 13.	
(5) Question 14.	

(10) Question 15.

(10) Question 16.

(10) Question 17.

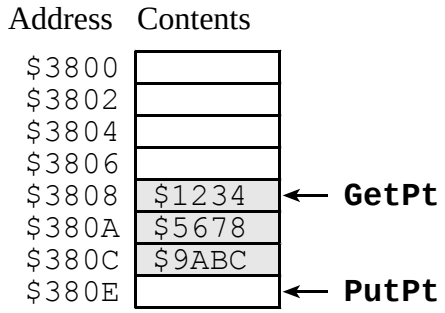
For questions 1 and 2, consider the following C program.

```
xxxx short V=4;
void function(void){ yyyy short W=4;
}
```

(5) Question 1. What qualifier should be used for **xxxx** so that **V** is allocated in ROM? Select from **signed unsigned volatile const static extern**

(5) Question 2. What qualifier should be used for **yyyy** so that **W** is permanently allocated in RAM? Select from **signed unsigned volatile const static extern**

This Fifo queue can hold up to eight 16-bit data values, and the picture shows it currently is holding three values (shaded).



(5) Question 3. What value is returned if we were to call **Fifo_Get** at this point?

(5) Question 4. Next, assume we call **Fifo_Put**. What will be the new **PutPt** after we call **Fifo_Put**?

Questions 5 and 6 involve the following assembly code.

```
main lds #$4000
     ldy #1000
     pshy        ; pass 16-bit in parameter on stack
     jsr sub1
     puly        ; balance stack
     stop
data set xxx     ; binding of 16-bit local variable
in   set yyy     ; binding of 16-bit input parameter
sub1 pshx        ; save register X
     tsx         ; RegX stack frame
     leas -2,s   ; allocate 16-bit local variable called data
;****body of the subroutine
     ldd  in,x   ; get a copy of in parameter
     std  data,x ; store into local variable data
;****end of body
     leas 2,s    ; deallocate data
     pulx        ; restore register X
     rts         ; return
```

(5) Question 5. What value should you use in the **xxx** position to implement the binding of the local variable, **data**?

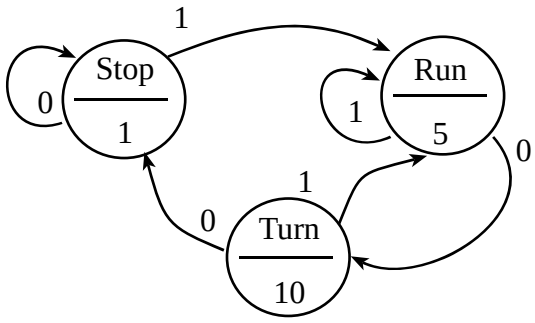
(5) Question 6. What value should you use in the **yyy** position to implement the binding of the parameter, **in**?

(5) Question 7. Consider a 10-bit ADC with a range of 0 to +5V. What is the approximate resolution of this ADC? Give units.

(5) Question 8. What term do we use to describe it when the debugging code itself makes a small but acceptable change in the behavior of a software system?

(5) Question 9. To verify the proper functionality of a subroutine, we write a special main program that provides a known and repeatable sequence of inputs to the subroutine under test. In this way, each time the test subroutine is changed, we can be sure the change in output values is caused by the software modification and not due to a change in input values. What is this debugging procedure called?

The following 6812 assembly program implements a one-input four-output finite state machine. The input is on Port M bit 0 and the output is on Port T bits 3,2,1,0.




```
     org  $4000  Put in ROM

Stop fcb  1          ;Output
     fdb  Stop,Run   ;Next
Run  fcb  5
     fdb  Turn,Run
Turn fcb  10
     fdb  sss,ttt
Main lds  #$4000
     bset DDRT,#$0F  ; PT3-0 outputs
     bclr DDRM,#$01  ; PM0 is input
     ldx  #Stop      ; RegX is the State pointer
FSM  yyy             ; RegA is Output value for this state
     staa PTT        ; Perform the output
     ldab PTM        ; Read input
     andb #$01       ; just interested in bit 0
     lslb            ; 2 bytes per 16 bit address
     abx             ; add 0,2 depending on input
     zzz             ; Next state depending on input
     bra  FSM
```

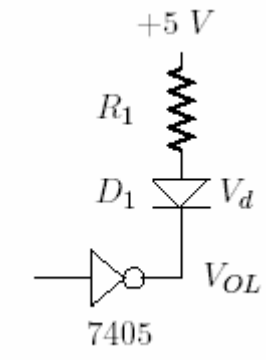
(5) Question 10. What should you put in the **sss,ttt** positions?

(5) Question 11. Which instruction causes an unfriendly operation?

(5) Question 12. What instruction (op code and operand) goes in the **yyy** position?

(5) Question 13. What instruction (op code and operand) goes in the **zzz** position?

(5) Question 14. Specify the resistor value for R_1, assuming LED current I_d is 2 mA, the LED voltage V_d is 1.5 V, and the gate output voltage V_{OL} is 0.5V.



(10) Question 15. Write a subroutine that generates one conversion on channel 4 of the 9S12C32 ADC and returns the 10-bit unsigned right-justified ADC conversion in Reg X (return by value). You may assume the ADC has already been enabled with the following function.

```
ADC_Init  movb #$80,ATDCTL2   ;ADPU=1 enables A/D
          movb #$08,ATDCTL3   ;sequence length=1
          movb #$04,ATDCTL4   ;10-bit A/D
          rts
```

Address	Bit 7	6	5	4	3	2	1	Bit 0	Name
$0082	ADPU	AFFC	AWAI	ETRIGLE	ETRIGP	ETRIG	ASCIE	ASCIF	ATDCTL2
$0083	0	S8C	S4C	S2C	S1C	FIFO	FRZ1	FRZ0	ATDCTL3
$0084	SRES8	SMP1	SMP0	PRS4	PRS3	PRS2	PRS1	PRS0	ATDCTL4
$0085	DJM	DSGN	SCAN	MULT	0	CC	CB	CA	ATDCTL5
$0086	SCF	0	ETORF	FIFOR	0	CC2	CC1	CC0	ATDSTAT0
$008B	CCF7	CCF6	CCF5	CCF4	CCF3	CCF2	CCF1	CCF0	ATDSTAT1
$008D	Bit 7	6	5	4	3	2	1	Bit 0	ATDDIEN
$0270	PTAD7	PTAD6	PTAD5	PTAD4	PTAD3	PTAD2	PTAD1	PTAD0	PTAD
$0272	DDRAD7	DDRAD6	DDRAD5	DDRAD4	DDRAD3	DDRAD2	DDRAD1	DDRAD0	DDRAD

address	msb															lsb	Name
$0090	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	ATDDR0
$0092	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	ATDDR1
$0094	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	ATDDR2
$0096	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	ATDDR3
$0098	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	ATDDR4
$009A	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	ATDDR5
$009C	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	ATDDR6
$009E	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	ATDDR7

(10) Question 16. Write a subroutine that outputs a null-terminated string to the SCI transmitter. An address to the string is pushed on the stack (call by reference). You may assume the SCI has already been enabled with the following function.

```
SCI_Init  movb #$0c,SCICR2  ;enable SCI
          movw #1,SCIBD     ;baud rate=250000
          rts
```

Addr	Bit 7	6	5	4	3	2	1	Bit 0	Name
$00C8	BTST	BSPL	BRLD	SBR12	SBR11	SBR10	SBR9	SBR8	SCIBD
$00C9	SBR7	SBR6	SBR5	SBR4	SBR3	SBR2	SBR1	SBR0	
$00CB	TIE	TCIE	RIE	ILIE	TE	RE	RWU	SBK	SCICR2
$00CC	TDRE	TC	RDRF	IDLE	OR	NF	FE	PF	SCISR1
$00CF	R7T7	R6T6	R5T5	R4T4	R3T3	R2T2	R1T1	R0T0	SCIDRL

An example calling sequence is

```
Message  fcb "Hello world",0
Main     lds  #$4000
         jsr  SCI_Init
         movw #Message,2,-sp  ;push address on stack
         jsr  SCI_Out         ;your subroutine
         leas 2,s             ;balance stack
```

(10) Question 17. Assume 16-bit TCNT is enabled and running. Write an SWI handler that reads the value on TCNT and returns the result in Reg X (return by value). An example calling sequence

```
swi
```

After the software interrupt returns, RegX contains the value from TCNT. The SWI vector is located at $FFF6.

SWI **Software Interrupt** **SWI**

Operation:
$(SP) - \$0002 \Rightarrow SP; RTN_H : RTN_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$
$(SP) - \$0002 \Rightarrow SP; Y_H : Y_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$
$(SP) - \$0002 \Rightarrow SP; X_H : X_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$
$(SP) - \$0002 \Rightarrow SP; B : A \Rightarrow (M_{(SP)} : M_{(SP+1)})$
$(SP) - \$0001 \Rightarrow SP; CCR \Rightarrow (M_{(SP)})$
$1 \Rightarrow I$
$(SWI\ Vector) \Rightarrow PC$

Description: Causes an interrupt without an external interrupt service request. Uses the address of the next instruction after SWI as a return address. Stacks the return address, index registers Y and X, accumulators B and A, and the CCR, decrementing the SP before each item is stacked. The I mask bit is then set, the PC is loaded with the SWI vector, and instruction execution resumes at that location. SWI is not affected by the I mask bit.

RTI **Return from Interrupt** **RTI**

Operation:
$(M_{(SP)}) \Rightarrow CCR; (SP) + \$0001 \Rightarrow SP$
$(M_{(SP)} : M_{(SP+1)}) \Rightarrow B : A; (SP) + \$0002 \Rightarrow SP$
$(M_{(SP)} : M_{(SP+1)}) \Rightarrow X_H : X_L; (SP) + \$0004 \Rightarrow SP$
$(M_{(SP)} : M_{(SP+1)}) \Rightarrow PC_H : PC_L; (SP) - \$0002 \Rightarrow SP$
$(M_{(SP)} : M_{(SP+1)}) \Rightarrow Y_H : Y_L; (SP) + \$0004 \Rightarrow SP$

Description: Restores system context after interrupt service processing is completed. The condition codes, accumulators B and A, index register X, the PC, and index register Y are restored to a state pulled from the stack. The X mask bit may be cleared as a result of an RTI instruction, but cannot be set if it was cleared prior to execution of the RTI instruction.

```
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
adcb   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
andcc  8-bit logical and to RegCC
asl/lsl   8-bit left shift Memory
asla/lsla 8-bit left shift RegA
aslb/lslb 8-bit arith left shift RegB
asld/lsld 16-bit left shift RegD
asr    8-bit arith right shift Memory
asra   8-bit arith right shift
asrb   8-bit arith right shift to RegB
bcc    branch if carry clear
bclr   clear bits in memory
bcs    branch if carry set
beq    branch if result is zero (Z=1)
bge    branch if signed ≥
bgnd   enter background debug mode
bgt    branch if signed >
bhi    branch if unsigned >
bhs    branch if unsigned ≥
bita   8-bit and with RegA, sets CCR
bitb   8-bit and with RegB, sets CCR
ble    branch if signed ≤
blo    branch if unsigned <
bls    branch if unsigned ≤
blt    branch if signed <
bmi    branch if result is negative (N=1)
bne    branch if result is nonzero (Z=0)
bpl    branch if result is positive (N=0)
bra    branch always
brclr  branch if bits are clear,
brn    branch never
brset  branch if bits are set
bset   set bits in memory
bsr    branch to subroutine
bvc    branch if overflow clear
bvs    branch if overflow set
call   subroutine in expanded memory
cba    8-bit compare RegA with RegB
clc    clear carry bit, C=0
cli    clear I=0, enable interrupts
clr    8-bit Memory clear
clra   RegA clear
clrb   RegB clear
clv    clear overflow bit, V=0
cmpa   8-bit compare RegA with memory
cmpb   8-bit compare RegB with memory
com    8-bit logical complement to Memory
coma   8-bit logical complement to RegA
comb   8-bit logical complement to RegB
cpd    16-bit compare RegD with memory
cpx    16-bit compare RegX with memory
cpy    16-bit compare RegY with memory
daa    8-bit decimal adjust accumulator
dbeq   decrement and branch if result=0
dbne   decrement and branch if result≠0
dec    8-bit decrement memory
deca   8-bit decrement RegA
decb   8-bit decrement RegB
des    16-bit decrement RegSP
dex    16-bit decrement RegX
dey    16-bit decrement RegY
ediv   RegY=(Y:D)/RegX, unsigned divide
edivs  RegY=(Y:D)/RegX, signed divide
emacs  16 by 16 signed mult, 32-bit add
emaxd  16-bit unsigned maximum in RegD
emaxm  16-bit unsigned maximum in memory
emind  16-bit unsigned minimum in RegD
eminm  16-bit unsigned minimum in memory
emul   RegY:D=RegY*RegD unsigned mult
emuls  RegY:D=RegY*RegD signed mult
eora   8-bit logical exclusive or to RegA
eorb   8-bit logical exclusive or to RegB
etbl   16-bit look up and interpolation
exg    exchange register contents
fdiv   16-bit unsigned fractional divide
ibeq   increment and branch if result=0
ibne   increment and branch if result≠0
idiv   16-bit unsigned divide, X=D/X
idivs  16-bit signed divide, X=D/X
inc    8-bit increment memory
inca   8-bit increment RegA
incb   8-bit increment RegB
ins    16-bit increment RegSP
inx    16-bit increment RegX
iny    16-bit increment RegY
jmp    jump always
jsr    jump to subroutine
lbcc   long branch if carry clear
lbcs   long branch if carry set
lbeq   long branch if result is zero
lbge   long branch if signed ≥
lbgt   long branch if signed >
lbhi   long branch if unsigned >
lbhs   long branch if unsigned ≥
lble   long branch if signed ≤
lblo   long branch if unsigned <
lbls   long branch if unsigned ≤
lblt   long branch if signed <
lbmi   long branch if result is negative
lbne   long branch if result is nonzero
lbpl   long branch if result is positive
lbra   long branch always
lbrn   long branch never
lbvc   long branch if overflow clear
lbvs   long branch if overflow set
ldaa   8-bit load memory into RegA
ldab   8-bit load memory into RegB
ldd    16-bit load memory into RegD
lds    16-bit load memory into RegSP
ldx    16-bit load memory into RegX
ldy    16-bit load memory into RegY
leas   16-bit load effective addr to SP
leax   16-bit load effective addr to X
leay   16-bit load effective addr to Y
lsr    8-bit logical right shift memory
lsra   8-bit logical right shift RegA
lsrb   8-bit logical right shift RegB
lsrd   16-bit logical right shift RegD
maxa   8-bit unsigned maximum in RegA
maxm   8-bit unsigned maximum in memory
mem    determine the membership grade
mina   8-bit unsigned minimum in RegA
minm   8-bit unsigned minimum in memory
movb   8-bit move memory to memory
movw   16-bit move memory to memory
mul    RegD=RegA*RegB
neg    8-bit 2's complement negate memory
nega   8-bit 2's complement negate RegA
negb   8-bit 2's complement negate RegB
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
pshb   push 8-bit RegB onto stack
pshc   push 8-bit RegCC onto stack
pshd   push 16-bit RegD onto stack
pshx   push 16-bit RegX onto stack
pshy   push 16-bit RegY 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 RegCC
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
rola   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=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
tap    transfer A to CC
tba    transfer B to A
tbeq   test and branch if result=0
tbl    8-bit look up and interpolation
tbne   test and branch if result≠0
tfr    transfer register to register
tpa    transfer CC to A
trap   illegal op code, or software trap
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 S to Y
txs    transfer X to S
tys    transfer Y to S
wai    wait for interrupt
wav    weighted Fuzzy logic average
xgdx   exchange RegD with RegX
xgdy   exchange RegD with RegY
```

example	addressing mode	Effective Address
ldaa #u	immediate	none
ldaa u	direct	EA is 8-bit address (0 to 255)
ldaa U	extended	EA is a 16-bit address
ldaa m,r	5-bit index	EA=r+m (-16 to 15)
ldaa v,+r	pre-increment	r=r+v, EA=r (1 to 8)
ldaa v,-r	pre-decrement	r=r-v, EA=r (1 to 8)
ldaa v,r+	post-increment	EA=r, r=r+v (1 to 8)
ldaa v,r-	post-decrement	EA=r, r=r-v (1 to 8)
ldaa A,r	Reg A offset	EA=r+A, zero padded
ldaa B,r	Reg B offset	EA=r+B, zero padded
ldaa D,r	Reg D offset	EA=r+D
ldaa q,r	9-bit index	EA=r+q (-256 to 255)
ldaa W,r	16-bit index	EA=r+W (-32768 to 65535)
ldaa [D,r]	D indirect	EA={r+D}
ldaa [W,r]	indirect	EA={r+W} (-32768 to 65535)

Freescale 6812 addressing modes

Pseudo op				meaning
org				Specific absolute address to put subsequent object code
=	**equ**			Define a constant symbol
set				Define or redefine a constant symbol
dc.b	**db**	**fcb**	**.byte**	Allocate byte(s) of storage with initialized values
fcc				Create an ASCII string (no termination character)
dc.w	**dw**	**fdb**	**.word**	Allocate word(s) of storage with initialized values
dc.l	**dl**		**.long**	Allocate 32-bit long word(s) of storage with initialized values
ds	**ds.b**	**rmb**	**.blkb**	Allocate bytes of storage without initialization
ds.w			**.blkw**	Allocate bytes of storage without initialization
ds.l			**.blkl**	Allocate 32-bit words of storage without initialization