(4) Question 1. An embedded system will use an 11-bit ADC to measure a distance. The measurement system range is -5 to +5 m. The frequency components of the distance signal can vary from DC (0 Hz) up to 200 Hz. You will use a periodic output compare interrupt to sample the ADC. What rate (in Hz) should you activate an output compare interrupt? Give a brief explanation.

(4) Question 2. A 1-ms periodic output compare interrupt is used to spin a stepper motor. During each ISR the four-bit motor output is set to 5, 6, 10, then 9. The stepper is interfaced to PT3-0, and the following four instructions occur during each ISR, without any delay between these instructions.

```
movb #$05,PTT
movb #$06,PTT
movb #$0A,PTT
movb #$09,PTT
```

There are 200 steps per rotation of the motor. What will happen?
A) The motor will spin at 5 rps = (1 step/ms)*(1000ms/s)*(1 rot/200steps)
B) The motor will spin at 20 rps = (4 step/ms)*(1000ms/s)*(1 rot/200steps)
C) The motor will spin at 1000 rps = (1 rot/ms)*(1000ms/s)
D) The motor will spin at 4000 rps = (4 rot/ms)*(1000ms/s)
E) The motor will not spin at all

(4) Question 3. Write a subroutine to sample ADC channel 5 of the 9S12DP512. Assume the ADC initialized for a 10-bit sample, sequence length is 1, and the ADC clock is 1 MHz. Implement right-justified conversions, and return the result in RegY.
(8) Question 4. Assume you have a 12-bit ADC with a range of 0 to +4 V (not the 9S12). Write a subroutine that converts the ADC sample into a fixed-point number with a resolution of 0.001 V. The input parameter is call by value in RegD containing the right-justified ADC sample (0 to 4095). Minimize errors due to dropout and overflow. Return by value the integer part of the fixed-point number in RegY. E.g., if the input voltage is 1.25 V then RegY is returned as 1250.

(4) Question 5. Assume RegA = $55, RegY=$1234 and RegX = $5678. What is the value in RegX after executing these instructions?
```
psha
  stx 2,-sp
  sty 2,sp-
  leas 2,sp
```
```
pula
pulx
```

(4) Question 6. These seven events all occur during each RDRF interrupt.
1) There is data in the receive data register and the hardware sets the flag bit (e.g., RDRF=1)
2) The SCI vector address is loaded into the PC
3) The I bit in the CCR is set by hardware
4) The software reads SCI1DRL
5) The software reads SCI1SR1
6) The CCR, A, B, X, Y, PC are pushed on the stack
7) The software executes rti
Which of the following sequences could be possible? Pick one answer A-F (only one is correct)
A) 1,3,6,2,4,5,7
B) 5,1,3,4,2,6,7
C) 1,2,5,3,4,6,7
D) 1,6,3,2,5,4,7
E) 1,6,3,2,4,5,7
F) None of the above sequences are possible
(10) Question 7. Write software that increments a 16-bit global variable every 2 msec using output compare 3. Show the complete main program, the OC3 ISR, the interrupt vector, and the reset vector. After initialization the main program executes a do-nothing loop. Write code as friendly as possible. Assume the E clock is 8 MHz. To make it easier for me to grade, leave TSCR2 equal to 0.

```assembly
org $0800
Count rmb 2 ;incremented every 2 msec
org $4000
```
(5) Question 8. Consider a serial port operating with a baud rate of 10,000 bits per second. Draw the waveform occurring at the PS3 output (voltage levels are +5 and 0) when the ASCII ‘S’ ($53) is transmitted on SCI1. The protocol is 1 start, 8 data and 1 stop bit. SCI1 is initially idle, and the software writes the $53 to SCI1DRL at time=0. Show the PS3 line before and after the frame, assuming the channel is idle before and after the frame.

<table>
<thead>
<tr>
<th>PS3</th>
<th>t</th>
<th>-1</th>
<th>0</th>
<th>100</th>
<th>200</th>
<th>300</th>
<th>400</th>
<th>500</th>
<th>600</th>
<th>700</th>
<th>800</th>
<th>900</th>
<th>1000</th>
<th>1100</th>
<th>1200 us</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(5) Question 9. Consider a computer network where two 9S12s are connected via their SCI1 ports, using the 3-wire cable like in Lab 7. The transmitter of computer 1 is connected to the receiver of computer 2, and the transmitter of computer 2 is connected to the receiver of computer 1. Initially, both SCI1 ports are idle. The baud rate on both computers is initialized to 10000 bits/sec, with 1 start, 8 data and 1 stop bit. Both computers have their RDRF flags armed and enabled. The transmitters are active, but not armed for interrupts. The I bit is clear in both computers. At time 0, computer 1 reads SCI1SR1 then writes to SCI1DRL. The RDRF ISR in computer 2 will read its SCI1DRL then write to its SCI1DRL (echo the data back). Approximately how long after computer 1 writes to SCI1DRL will an RDRF interrupt occur back in computer 1? Assume the software execution time is fast compared to the I/O transmission time.

(4) Question 10. Consider the following three-bit DAC connected to Port T. Fill in the expected response table assuming $V_{OH}$ is 5 V and $V_{OL}$ is 0 V.

<table>
<thead>
<tr>
<th>PT2</th>
<th>PT1</th>
<th>PT0</th>
<th>Vout (V)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
(8) Question 11. Assume the PC equals $4003, and the SP equals $3FFE. Initially, memory contains data as shown in the figure. You will be executing one instruction and answering questions about executing that one instruction.

Part a) Given the initial conditions in this figure, what instruction will be executed next?

Part b) As you execute that one instruction, two bytes are stored into memory? Give the addresses and the 8-bit data values that are stored.

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>$01</td>
<td>$3FFC</td>
</tr>
<tr>
<td>$02</td>
<td>$3FFD</td>
</tr>
<tr>
<td>$3D</td>
<td>$3FFE</td>
</tr>
<tr>
<td>$04</td>
<td>$3FFF</td>
</tr>
<tr>
<td>$CF</td>
<td>$4000</td>
</tr>
<tr>
<td>$3F</td>
<td>$4001</td>
</tr>
<tr>
<td>$FE</td>
<td>$4002</td>
</tr>
<tr>
<td>$07</td>
<td>$4003</td>
</tr>
<tr>
<td>$02</td>
<td>$4004</td>
</tr>
<tr>
<td>$20</td>
<td>$4005</td>
</tr>
<tr>
<td>$FC</td>
<td>$4006</td>
</tr>
<tr>
<td>$3B</td>
<td>$4007</td>
</tr>
</tbody>
</table>

Part c) What is the SP after the one instruction is executed?

Part d) What is the PC after the one instruction is executed?

(5) Question 12. You are given an LED with a (1.5 V, 30 mA) operating point. Interface this LED to the 9S12 using a 7406, such that the LED is on when PP0 is high and the LED is off when PP0 is low. The $V_{OL}$ of the 7406 is 0.5 V. Label all resistor values. No software is required.
10) **Question 13.** A positive logic switch is connected to PT0 and a positive logic LED is connected to PP0. Design a Moore finite state machine that counts the number of times the switch is pressed and released, so that the LED is turned on if the switch is pressed 3 or more times. The LED should come on after the switch is released the third time. Switch bounce causes the input to toggle low/high/low/high every time the switch is touched, and to toggle high/low/high/low every time the switch is released. This bounce is typically less than 1ms. You may assume the switch input is high for at least 100 ms when touched and low for at least 100 ms when released. In other words, the maximum rate at which the operator will push the switch is 5 times/sec. To eliminate switch bounce, you will read the input at a rate slower than every 10 ms, but faster than every 100 ms. The FSM controller will repeat this sequence in the foreground over and over:

1) Output to the LED, as defined by the state
2) Wait a prescribed amount of time, as defined by the state
3) Input from the switch
4) Go to the next state, as defined by the state and by the input

*Draw the FSM graph.* Specify the initial state. NO SOFTWARE IS REQUIRED.
Question 14. In this question, the subroutine implements a call by reference parameter passed on the stack. There are no return parameters. Call by reference means an address to the data is pushed on the stack. A typical calling sequence is

```
org $4000
Data fcb 200          ;8-bit information
Main lds #$4000
    movw #Data,2,-sp ;pointer to the Data is pushed
    jsr Subroutine
    leas 2,sp         ;discard parameter
```

The subroutine allocates one 8-bit local variable, L1, and uses RegX frame pointer addressing to access the local variable and parameter. The binding for these three are

```
Pt set ??? ;16-bit pointer to 8-bit data
L1 set ??? ;8-bit local variable
```

```
Subroutine
    pshx ;save old stack frame pointer
    tsx ;establish new stack frame pointer
    leas -1,sp ;allocate L1

;---------start of body-------------------
    ldaa ????? ;Reg A = value of the parameter
    staa L1,x ;save parameter into local L1

;---------end of body---------------------
    leas 1,sp ;deallocate
    pulx
    rts
```

Part a) Show the binding for the ??? parameters in the above program.

```
Pt set _____

L1 set _____
```

Part b) Show the operand for the ???? in the above program. In particular, you must use Register X stack frame addressing, Pt binding, and bring the value of the parameter into Register A. It can be done in one instruction, but for partial credit you can use two instructions.
(15) Question 15. This FIFO queue has 8 allocated locations and can hold up to eight 8-bit data values. The picture shows it currently holding three values (shaded). The FIFO and its three variables are defined in RAM. When the counter is zero the FIFO is empty.

```
org $3900
Fifo rmb 8 ;allocates 8 bytes
GetI rmb 1 ;index where to find oldest data
PutI rmb 1 ;index where to put next data
Cnt rmb 1 ;number of elements stored in fifo
```

This function initializes the FIFO

```
Fifo_Init clr Cnt ;no data in Fifo
    clr GetI ;Get next from Fifo[GetI]
    clr PutI ;Put next into Fifo[PutI]
rts
```

Write an assembly subroutine, `Fifo_Put`, that implements the put operation. The input parameter contains the data to put as call by value in RegA, and a result code is returned in RegB. If RegB=1, then the input data was successfully stored. If RegB=0, the data could not be saved in the FIFO because it was previously full at the time of the call.

```
;input: RegA,   Output: RegB=success
Fifo_Put
```
aba 8-bit add RegA=RegA+RegB
abx unsigned add RegX=RegX+RegB (zero pad)
aby unsigned add RegY=RegY+RegB (zero pad)
adc a 8-bit add with carry to RegA
addc 8-bit add with carry to RegB
add 8-bit add to RegA
addb 8-bit add to RegB
addd 16-bit add to RegD
andra 8-bit logical and to RegA
anddb 8-bit logical and to RegB
andccc 8-bit logical and to RegCC
asl/lsl 8-bit left shift Memory
asl/lslb 8-bit left shift RegA, A = A*2
asl/lslb 8-bit left shift RegB, B = B*2
aslld/lsld 16-bit left shift RegD, D = D*2
asr 8-bit arith right shift Memory
asra 8-bit arith right shift to RegA, signed A = A/2
asrb 8-bit arith right shift to RegB, signed B = B/2
bcc branch if carry clear
bcclr bit clear in memory bcclr PTT,#$01
bcs branch if carry set
beq branch if result is zero (Z=1)
bege branch if signed ≥
bgnd enter background debug mode
bgt branch if signed >
bhi branch if unsigned >
bs branch if unsigned ≥
bits 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 ≤
bit branch if signed ≤
bmi branch if result is negative (N=1)
bne branch if result is nonzero (Z=0)
bl 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 brset PTT,#$01,loop
bset bit set clear in memory bset PTT,#$04
bshr branch to subroutine
bvc branch if overflow clear
bvs branch if overflow set
call subroutine in expanded memory (push PC, PPAGE)
cba 8-bit compare RegA with RegB
cic clear carry bit, C=0
cil clear I=0, enable interrupts
cir 8-bit memory clear
cira RegA clear, A=0
cirb RegB clear, B=0
ciiv 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
com.a 8-bit logical complement to RegA, A = ~A
comb 8-bit logical complement to RegB, B = ~B
cpd 16-bit compare RegD with memory
cpx 16-bit compare RegX with memory
cpy 16-bit compare RegY with memory
da 8-bit decimal adjust accumulator (BCD addition)
dbeq branch and decrement if result=0 dbeq Y,loop
dbne branch and decrement if result≠0 dbne A,loop
dec 8-bit decrement memory
deca 8-bit decrement RegA, A = A-1
decb 8-bit decrement RegB, B = B-1
des 16-bit decrement RegSP, SP = SP-1
dex 16-bit decrement RegX, X = X-1
dey 16-bit decrement RegY, Y = Y-1
div RegY=(Y:D)/RegX, unsigned divide
divs RegY=(Y:D)/RegX, signed divide
emac 16 by 16 signed multiply, 32-bit add
emaxd 16-bit unsigned maximum in RegD
emaxm 16-bit unsigned maximum in memory
emin 16-bit unsigned minimum in RegD
eminn 16-bit unsigned minimum in memory
emul RegY=RegY*RegD unsigned multiply
emuls RegY=RegY*RegD signed multiply
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 exg X,Y
fdiv unsigned fract div, X=(65536*D)/X
ibeq increment and branch if result=0 ibeq Y,loop
ibne increment and branch if result≠0 ibne A,loop
idiv 16-bit unsigned div, X=D/X, D=remainder
idivs 16-bit signed divide, X=D/X, D=remainder
inc 8-bit increment memory
inc A 8-bit increment RegA, A = A+1
incb 8-bit increment RegB, B = B+1
ins 16-bit increment RegSP, SP = SP+1
inx 16-bit increment RegX, X = X+1
iny 16-bit increment RegY, Y = Y+1
jmp jump always
jsr jump to subroutine
lbcc long branch if carry clear
lbcs long branch if carry set
lbene 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 ≥
lbl long branch if signed ≤
lblo long branch if unsigned <
lbns long branch if result is negative
lbne long branch if result is nonzero
lbpl long branch if result is positive
lbra long branch always
lbrc long branch never
lbvcc 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
ldl 16-bit load memory into RegD
ldls 16-bit load memory into RegSP
ldlx 16-bit load memory into RegX
ldly 16-bit load memory into RegY
leas 16-bit load effective addr to SP leas 2,sp
leax 16-bit load effective addr to X leax 2,x
leay 16-bit load effective addr to Y leay 2,y
ler 8-bit logical right shift memory
lera 8-bit logical right shift RegA, A = A/2
lers 8-bit logical right shift RegB, B = B/2
lrsd 16-bit logical right shift RegD, D = D/2
maxa 8-bit unsigned maximum in RegA maxa 0,x
maxm 8-bit unsigned maximum in memory maxm 0,x
mem determine the Fuzzy Logic membership grade
mina 8-bit unsigned minimum in RegA mina 0,x
minm 8-bit unsigned minimum in memory minm 0,x
movb 8-bit move memory to memory movb #100,PTT
movw 16-bit move memory to memory movw #133,SCIBD
mul RegD=RegA*RegB
### Instruction Set

- **neg** 8-bit 2's complement negate memory
- **nega** 8-bit 2's complement negate RegA, A = -A
- **negb** 8-bit 2's complement negate RegB, B = -B
- **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
- **psy** 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 8 bits off stack into RegD
- **pulx** pop 8 bits off stack into RegX
- **puly** pop 16 bits off stack into RegY
- **rev** Fuzzy Logic rule evaluation
- **revw** weighted Fuzzy Logic 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 RegA
- **rorc** 8-bit roll shift right RegB
- **rtd** return sub in expanded memory (pull PPAGE, PC)
- **rte** return from interrupt (pull CCR,B,A,X,Y,PC)
- **rts** return from subroutine (pull PC)
- **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 F=1, disable interrupts
- **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 (push PC,Y,X,A,B,CCR)
- **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** test and branch if result=0
- **tfr** transfer register to register
- **tpa** transfer CC to A
- **trap** illegal instruction interrupt, software interrupt
- **tst** 8-bit compare memory with zero
- **tsta** 8-bit compare RegA with zero
- **tstb** 8-bit compare RegB with zero
- **txs** transfer S to X
- **tys** transfer Y to S
- **wai** wait for interrupt

### Example Operations

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Mode</th>
<th>Effective Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldab #u</td>
<td>immediate</td>
<td>No EA</td>
</tr>
<tr>
<td>ldab u</td>
<td>direct</td>
<td>EA is 8-bit address</td>
</tr>
<tr>
<td>ldab U</td>
<td>extended</td>
<td>EA is a 16-bit address</td>
</tr>
<tr>
<td>ldab m,r</td>
<td>5-bit index</td>
<td>EA=r+m (-16 to 15)</td>
</tr>
<tr>
<td>ldab v,+r</td>
<td>pre-incr</td>
<td>r=r+v, EA=r (1 to 8)</td>
</tr>
<tr>
<td>ldab v,-r</td>
<td>pre-dec</td>
<td>r=r-v, EA=r (1 to 8)</td>
</tr>
<tr>
<td>ldab v,+,r</td>
<td>post-inc</td>
<td>EA=r, r=r+v (1 to 8)</td>
</tr>
<tr>
<td>ldab v,-,r</td>
<td>post-dec</td>
<td>EA=r, r=r-v (1 to 8)</td>
</tr>
<tr>
<td>ldab A,r</td>
<td>Reg A offset</td>
<td>EA=r+A, zero padded</td>
</tr>
<tr>
<td>ldab B,r</td>
<td>Reg B offset</td>
<td>EA=r+B, zero padded</td>
</tr>
<tr>
<td>ldab D,r</td>
<td>Reg D offset</td>
<td>EA=r+D</td>
</tr>
<tr>
<td>ldab q,r</td>
<td>9-bit index</td>
<td>EA=r+q</td>
</tr>
<tr>
<td>ldab W,r</td>
<td>16-bit index</td>
<td>EA=r+W</td>
</tr>
<tr>
<td>ldab [D,r]</td>
<td>D indirect</td>
<td>EA=r+D</td>
</tr>
<tr>
<td>ldab [W,r]</td>
<td>indirect</td>
<td>EA=r+W</td>
</tr>
</tbody>
</table>

### Freescale 6812 Addressing Mases

- **r** is X, Y, SP, or PC

### Pseudo Op

<table>
<thead>
<tr>
<th>Meaning</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>= equ</td>
<td>set</td>
</tr>
<tr>
<td>= equ</td>
<td>Where to put subsequent code</td>
</tr>
<tr>
<td>dc.b db</td>
<td>Allocate byte(s) with values</td>
</tr>
<tr>
<td>db fcb</td>
<td>Create an ASCII string</td>
</tr>
<tr>
<td>.byte</td>
<td>Allocate word(s) with values</td>
</tr>
<tr>
<td>dc.w dw</td>
<td>Allocate word(s) with values</td>
</tr>
<tr>
<td>.word</td>
<td>Allocate 32-bit with values</td>
</tr>
<tr>
<td>ds db</td>
<td>Allocate bytes without init</td>
</tr>
<tr>
<td>.rmb .blkw</td>
<td>Allocate word(s) without init</td>
</tr>
<tr>
<td>ds .w .blkw</td>
<td>Allocate word(s) without init</td>
</tr>
</tbody>
</table>

### Vector Support

- **PIEP.**[7:0]**
- **PIEH.**[7:0]**
- **PIEJ.**[7,6,1,0]**
- **SCI1CR2.TIE,RIE**
- **SCI0CR2.TIE,RIE**
- **TSCR2.TOI**
- **CRGINT.RTIE**
- **STC0.TIE,RIE**
- **STC1.TIE,RIE**
- **Key Wakeup J**
- **Key Wakeup H**
- **Key Wakeup P**

### Some Interrupt Vectors
<table>
<thead>
<tr>
<th>Address</th>
<th>Bit 7</th>
<th>Bit 6</th>
<th>Bit 5</th>
<th>Bit 4</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0040$</td>
<td>IOS7</td>
<td>IOS6</td>
<td>IOS5</td>
<td>IOS4</td>
<td>IOS3</td>
<td>IOS2</td>
<td>IOS1</td>
<td>IOS0</td>
<td>TIOS</td>
</tr>
<tr>
<td>$0044$</td>
<td>TEN</td>
<td>TSWAI</td>
<td>TSFRZ</td>
<td>TFFCA</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>TSCR1</td>
</tr>
<tr>
<td>$004C$</td>
<td>C7I</td>
<td>C6I</td>
<td>C5I</td>
<td>C4I</td>
<td>C3I</td>
<td>C2I</td>
<td>C1I</td>
<td>C0I</td>
<td>TIE</td>
</tr>
<tr>
<td>$004D$</td>
<td>TOI</td>
<td>0</td>
<td>PUP T</td>
<td>RDPT</td>
<td>TCRE</td>
<td>PR2</td>
<td>PR1</td>
<td>PR0</td>
<td>TSCR2</td>
</tr>
<tr>
<td>$004E$</td>
<td>C7F</td>
<td>C6F</td>
<td>C5F</td>
<td>C4F</td>
<td>C3F</td>
<td>C2F</td>
<td>C1F</td>
<td>C0F</td>
<td>TFLG1</td>
</tr>
<tr>
<td>$004F$</td>
<td>TOF</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>TFLG2</td>
</tr>
<tr>
<td>$0050$</td>
<td>Bit 15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>TC0</td>
</tr>
<tr>
<td>$0052$</td>
<td>Bit 15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>TC1</td>
</tr>
<tr>
<td>$0054$</td>
<td>Bit 15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>TC2</td>
</tr>
<tr>
<td>$0056$</td>
<td>Bit 15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>TC3</td>
</tr>
<tr>
<td>$0058$</td>
<td>Bit 15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>TC4</td>
</tr>
<tr>
<td>$005A$</td>
<td>Bit 15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>TC5</td>
</tr>
<tr>
<td>$005C$</td>
<td>Bit 15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>TC6</td>
</tr>
<tr>
<td>$005E$</td>
<td>Bit 15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>TC7</td>
</tr>
<tr>
<td>$0082$</td>
<td>ADPU</td>
<td>AFFC</td>
<td>ASWAI</td>
<td>ETRIGL</td>
<td>ETRIGP</td>
<td>ETRIG</td>
<td>ASCIE</td>
<td>ASCIF</td>
<td>ATD0CTL2</td>
</tr>
<tr>
<td>$0084$</td>
<td>SRE5</td>
<td>SMP1</td>
<td>SMP0</td>
<td>PRS4</td>
<td>PRS3</td>
<td>PRS2</td>
<td>PRS1</td>
<td>PRS0</td>
<td>ATD0CTL3</td>
</tr>
<tr>
<td>$0085$</td>
<td>DJM</td>
<td>DSGN</td>
<td>SCAN</td>
<td>MULT</td>
<td>0</td>
<td>CC</td>
<td>CB</td>
<td>CA</td>
<td>ATD0CTL4</td>
</tr>
<tr>
<td>$0086$</td>
<td>SCF</td>
<td>0</td>
<td>ETORF</td>
<td>FIFOR</td>
<td>0</td>
<td>CC2</td>
<td>CC1</td>
<td>CC0</td>
<td>ATD0CTL5</td>
</tr>
<tr>
<td>$0088$</td>
<td>CCF7</td>
<td>CCF6</td>
<td>CCF5</td>
<td>CCF4</td>
<td>CCF3</td>
<td>CCF2</td>
<td>CCF1</td>
<td>CCF0</td>
<td>ATD0STAT0</td>
</tr>
<tr>
<td>$008A$</td>
<td>Bit 7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>ATD0DIEN</td>
</tr>
<tr>
<td>$008C$</td>
<td>PAD07</td>
<td>PAD06</td>
<td>PAD05</td>
<td>PAD04</td>
<td>PAD03</td>
<td>PAD02</td>
<td>PAD01</td>
<td>PAD00</td>
<td>PORTAD0</td>
</tr>
<tr>
<td>$0090$</td>
<td>Bit 15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>ATD0DR0</td>
</tr>
<tr>
<td>$0092$</td>
<td>Bit 15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>ATD0DR1</td>
</tr>
<tr>
<td>$0094$</td>
<td>Bit 15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>ATD0DR2</td>
</tr>
<tr>
<td>$0096$</td>
<td>Bit 15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>ATD0DR3</td>
</tr>
<tr>
<td>$0098$</td>
<td>Bit 15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>ATD0DR4</td>
</tr>
<tr>
<td>$009A$</td>
<td>Bit 15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>ATD0DR5</td>
</tr>
<tr>
<td>$009C$</td>
<td>Bit 15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>ATD0DR6</td>
</tr>
<tr>
<td>$009E$</td>
<td>Bit 15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>ATD0DR7</td>
</tr>
<tr>
<td>$00C9$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SBR12</td>
<td>SBR11</td>
<td>SBR10</td>
<td>SBR0</td>
<td>SC10B</td>
<td></td>
</tr>
<tr>
<td>$00CA$</td>
<td>LOOPS</td>
<td>SCIWSAI</td>
<td>RSRC</td>
<td>M</td>
<td>WAKE</td>
<td>ILT</td>
<td>PE</td>
<td>PT</td>
<td>SCI0C1R</td>
</tr>
<tr>
<td>$00CB$</td>
<td>TIE</td>
<td>TCIE</td>
<td>RIE</td>
<td>ILIE</td>
<td>TE</td>
<td>RE</td>
<td>RWU</td>
<td>SBK</td>
<td>SCI0C2R</td>
</tr>
<tr>
<td>$00CC$</td>
<td>TDRE</td>
<td>TC</td>
<td>RDRF</td>
<td>IDLE</td>
<td>OR</td>
<td>NF</td>
<td>FE</td>
<td>PF</td>
<td>SCI0SR1</td>
</tr>
<tr>
<td>$00CD$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>BRK13</td>
<td>TXDIR</td>
<td>RAF</td>
<td>SCI0SR2</td>
</tr>
<tr>
<td>$00CF$</td>
<td>R7/T7</td>
<td>R6/T6</td>
<td>R5/T5</td>
<td>R4/T4</td>
<td>R3/T3</td>
<td>R2/T2</td>
<td>R1/T1</td>
<td>R0/T0</td>
<td>SCI0DRL</td>
</tr>
<tr>
<td>$00D0$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SBR12</td>
<td>SBR11</td>
<td>SBR10</td>
<td>SBR0</td>
<td>SCI1B</td>
<td></td>
</tr>
<tr>
<td>$00D2$</td>
<td>LOOPS</td>
<td>SCIWSAI</td>
<td>RSRC</td>
<td>M</td>
<td>WAKE</td>
<td>ILT</td>
<td>PE</td>
<td>PT</td>
<td>SCI1C1R</td>
</tr>
<tr>
<td>$00D3$</td>
<td>TIE</td>
<td>TCIE</td>
<td>RIE</td>
<td>ILIE</td>
<td>TE</td>
<td>RE</td>
<td>RWU</td>
<td>SBK</td>
<td>SCI1C2R</td>
</tr>
<tr>
<td>$00D4$</td>
<td>TDRE</td>
<td>TC</td>
<td>RDRF</td>
<td>IDLE</td>
<td>OR</td>
<td>NF</td>
<td>FE</td>
<td>PF</td>
<td>SCI1SR1</td>
</tr>
<tr>
<td>$00D5$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>BRK13</td>
<td>TXDIR</td>
<td>RAF</td>
<td>SCI1SR2</td>
</tr>
<tr>
<td>$00D7$</td>
<td>R7/T7</td>
<td>R6/T6</td>
<td>R5/T5</td>
<td>R4/T4</td>
<td>R3/T3</td>
<td>R2/T2</td>
<td>R1/T1</td>
<td>R0/T0</td>
<td>SCI1DRL</td>
</tr>
<tr>
<td>$0240$</td>
<td>PT7</td>
<td>PT6</td>
<td>PT5</td>
<td>PT4</td>
<td>PT3</td>
<td>PT2</td>
<td>PT1</td>
<td>PT0</td>
<td>PTT</td>
</tr>
<tr>
<td>$0242$</td>
<td>DDR7</td>
<td>DDR6</td>
<td>DDR5</td>
<td>DDR4</td>
<td>DDR3</td>
<td>DDR2</td>
<td>DDR1</td>
<td>DDR0</td>
<td>DDR</td>
</tr>
<tr>
<td>$0244$</td>
<td>PS7</td>
<td>PS6</td>
<td>PS5</td>
<td>PS4</td>
<td>PS3</td>
<td>PS2</td>
<td>PS1</td>
<td>PS0</td>
<td>PTS</td>
</tr>
<tr>
<td>$024A$</td>
<td>DDR57</td>
<td>DDR56</td>
<td>DDR55</td>
<td>DDR54</td>
<td>DDR53</td>
<td>DDR52</td>
<td>DDR51</td>
<td>DDR50</td>
<td>DDRS</td>
</tr>
<tr>
<td>$0250$</td>
<td>PM7</td>
<td>PM6</td>
<td>PM5</td>
<td>PM4</td>
<td>PM3</td>
<td>PM2</td>
<td>PM1</td>
<td>PM0</td>
<td>PTM</td>
</tr>
<tr>
<td>$0252$</td>
<td>DDRM7</td>
<td>DDRM6</td>
<td>DDRM5</td>
<td>DDRM4</td>
<td>DDRM3</td>
<td>DDRM2</td>
<td>DDRM1</td>
<td>DDRM0</td>
<td>DDRM</td>
</tr>
<tr>
<td>$0258$</td>
<td>PP7</td>
<td>PP6</td>
<td>PP5</td>
<td>PP4</td>
<td>PP3</td>
<td>PP2</td>
<td>PP1</td>
<td>PP0</td>
<td>PTP</td>
</tr>
<tr>
<td>$0260$</td>
<td>PH7</td>
<td>PH6</td>
<td>PH5</td>
<td>PH4</td>
<td>PH3</td>
<td>PH2</td>
<td>PH1</td>
<td>PH0</td>
<td>PTH</td>
</tr>
<tr>
<td>$0262$</td>
<td>DDRH7</td>
<td>DDRH6</td>
<td>DDRH5</td>
<td>DDRH4</td>
<td>DDRH3</td>
<td>DDRH2</td>
<td>DDRH1</td>
<td>DDRH0</td>
<td>DDRH</td>
</tr>
<tr>
<td>$0268$</td>
<td>PJ7</td>
<td>PJ6</td>
<td>PJ5</td>
<td>PJ4</td>
<td>PJ3</td>
<td>PJ2</td>
<td>PJ1</td>
<td>PJ0</td>
<td>PTJ</td>
</tr>
<tr>
<td>$026A$</td>
<td>DDRJ</td>
<td>DDRJ</td>
<td>DDRJ</td>
<td>DDRJ</td>
<td>DDRJ</td>
<td>DDRJ</td>
<td>DDRJ</td>
<td>DDRJ</td>
<td>DDRJ</td>
</tr>
</tbody>
</table>

**TSCR1** is the first 8-bit timer control register.

- **TEN**, 1 allows the timer to function normally, 0 means disable timer including **TCNT**

**TSCR2** is the second 8-bit timer control register.

- Bits 2, 1, 0 are **PR2**, **PR1**, **PR0**, which select the rate, let **n** be the 3-bit number formed by **PR2**, **PR1**, **PR0**

- Without PLL **TCNT** is 8MHz/2^n, with PLL **TCNT** is 24MHz/2^n, **n** ranges from 0 to 7

**TIOS** is the 8-bit output compare select register, one bit for each channel (1 = output compare, 0 = input capture)

**TIE** is the 8-bit output compare arm register, one bit for each channel (1 = armed, 0 = disarmed)
SCIxBD is 16-bit SCI baud rate register, let \( n \) be the 16-bit number  
Baud rate is 250 kHz/\( n \)

SCIxCB2 is 8-bit SCI control register

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>TIE, Transmit Interrupt Enable</td>
<td>0 = TDRE interrupts disabled, 1 = interrupt whenever TDRE set</td>
</tr>
<tr>
<td>5</td>
<td>RIE, Receiver Interrupt Enable</td>
<td>0 = RDRF interrupts disabled, 1 = interrupt whenever RDRF set</td>
</tr>
<tr>
<td>3</td>
<td>TE, Transmitter Enable</td>
<td>0 = Transmitter disabled, 1 = SCI transmit logic is enabled</td>
</tr>
<tr>
<td>2</td>
<td>RE, Receiver Enable</td>
<td>0 = Receiver disabled, 1 = Enables the SCI receive circuitry</td>
</tr>
</tbody>
</table>

SCIxCB1 is 8-bit SCI status register

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>TDRE, Transmit Data Register Empty Flag</td>
<td>Set if transmit data can be written to SCDR</td>
</tr>
<tr>
<td>5</td>
<td>RDRF, Receive Data Register Full</td>
<td>Set if a received character is ready to be read from SCIxDRL</td>
</tr>
</tbody>
</table>

SCIxCB2

<table>
<thead>
<tr>
<th>Source Form</th>
<th>Address Mode</th>
<th>Object Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDAA #opr8</td>
<td>IMM</td>
<td>06, i8</td>
</tr>
<tr>
<td>LDAA opr8</td>
<td>DIR</td>
<td>06, d8</td>
</tr>
<tr>
<td>LDAA opr16h</td>
<td>EXT</td>
<td>06, hh, 11</td>
</tr>
<tr>
<td>LDAA oprx0,xyp</td>
<td>IDX</td>
<td>a6, xh</td>
</tr>
<tr>
<td>LDAA oprx3,xyp</td>
<td>IDX1</td>
<td>a6, xh, FF</td>
</tr>
<tr>
<td>LDAA oprx16,xyp</td>
<td>IDX2</td>
<td>a6, xh, ee, FF</td>
</tr>
<tr>
<td>LDAA [D,xyp]</td>
<td>[D,IDX]</td>
<td>a6, xh</td>
</tr>
<tr>
<td>LDAA [oprx16,xyp]</td>
<td>[IDX2]</td>
<td>a6, xh, ee, FF</td>
</tr>
</tbody>
</table>

LDAA

EMUL

Extended Multiply 16-Bit by 16-Bit (Unsigned)

<table>
<thead>
<tr>
<th>Source Form</th>
<th>Address Mode</th>
<th>Object Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMUL</td>
<td>INH</td>
<td>13</td>
</tr>
</tbody>
</table>

EDIV

Extended Divide 32-Bit by 16-Bit (Unsigned)

<table>
<thead>
<tr>
<th>Source Form</th>
<th>Address Mode</th>
<th>Object Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDIV</td>
<td>INH</td>
<td>11</td>
</tr>
</tbody>
</table>

BSR

Branch to Subroutine

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

RTS

Return from Subroutine

<table>
<thead>
<tr>
<th>Source Form</th>
<th>Address Mode</th>
<th>Object Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTS</td>
<td>INH</td>
<td>3D</td>
</tr>
</tbody>
</table>