First: __________________   Last: ______________________

This is a closed book exam. You must put your answers in the space provided. You have 3 hours, so allocate your time accordingly. Please read the entire exam before starting.

Please read and affirm our honor code:
“The core values of The University of Texas at Austin are learning, discovery, freedom, leadership, individual opportunity, and responsibility. Each member of the university is expected to uphold these values through integrity, honesty, trust, fairness, and respect toward peers and community.”

(4) Question 1. First, think of as many DAC parameters as you can. Listed here are experimental procedures one might use to measure a DAC parameter. State the DAC parameter determined by each.

Part a) The input is stepped from minimum to maximum. For each input change, the change in DAC output is measured. The results are processed to see if all the changes in output are positive.

_________

Part b) The input is stepped from minimum to maximum. For each input change, the change in DAC output is measured. The results are processed by averaging all the changes in output.

_________

Part c) The input is stepped from minimum to maximum. For each input change, the change in DAC output is measured. The results are processed by counting the number of changes in output.

_________

Part d) The input is stepped from minimum to maximum. For each input change, the DAC output value is measured. The results are processed by averaging the absolute values of the differences between the measured output and the expected output.

_________

(4) Question 2. Assume the SCI0 is already running, the E clock is 8 MHz, and value in TSCR2 is 3. Write C code that changes the baud rate to 1000 bits/sec. Do not include more code than needed. Your solution must compile as regular C code. It is not a function, just C code.
(5) **Question 3.** What resistance is needed for R in the circuit so the output voltage Vout is 1V?

![Circuit diagram]

(6) **Question 4.** A measurement system has a range of 0 to 19.9 cm and a resolution of 0.1 cm. Assume a variable called `position` is allocated in RAM. Figure out the smallest number of bytes needed to allocate `position` and write your code accordingly.

**Part a)** Write assembly code that multiplies the variable by 0.5 storing the result back into `position`. For example, if the initial value is 2.4 cm, the final value will be 1.2 cm.

**Part b)** Write assembly code that adds 2.0 cm to the variable storing the result back into `position`. For example, if the initial value is 2.4 cm, the final value will be 4.4 cm. You do not need to consider overflow.

(4) **Question 5.** Assume the SCI0 is already initialized; write a C function that receives one character using busy-wait synchronization.
(4) Question 6. The desired LED operating point is 2V, 20mA. Assume the \( V_{OL} \) of the 7406 is 0.5V. Interface this LED to PP0 using positive logic. Specify values for any resistors needed.

(4) Question 7. Assume RegB = $55, RegY=$1234 and RegX = $5678. What is the value in RegX after executing these instructions?

\[
\begin{align*}
&\text{pshb} \\
&\text{stx 2,-sp} \\
&\text{sty 2,sp-} \\
&\text{leas 3,sp} \\
&\text{pulx}
\end{align*}
\]

(6) Question 8. The goal is to write a function that multiplies a signed 16-bit number by 0.314. The assembly on the left was generated by the Metrowerks compiler. Recall the input parameter is passed in Reg D and the output result is returned in Reg D. This C code has an overflow bug. Rewrite the assembly subroutine removing the bug, but maintaining the manner with which parameters are passed.

<table>
<thead>
<tr>
<th>calc</th>
<th>TFR D,X</th>
<th>LDY 0,X</th>
<th>LDD #314</th>
<th>EMUL</th>
<th>LDX #1000</th>
<th>IDIVS</th>
<th>TFR X,D</th>
<th>RTS</th>
</tr>
</thead>
</table>
| short calc(short *in){ | short data; | data = (*in); | data = (314*data)/1000; | return data; |}
(2) **Question 9.** Consider the result of executing the following two 9S12 assembly instructions.

```assembly
ldaa #156
adda #-50
```

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

__________

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

__________

(4) **Question 10.** These six events all occur during each output compare 6 interrupt.

1) The \textbf{TCNT} equals \textbf{TC6} and the hardware sets the flag bit (e.g., C6F=1)
2) The PC is set to the contents of the output compare 6 vector
3) The I bit in the CCR is set by hardware
4) The CCR, A, B, X, Y, PC are pushed on the stack
5) The software executes something like
   
   ```assembly
   movb #$40,TFLG1
   ldd TC6
   addd #5000
   std TC6
   ```
6) The software executes \texttt{rti}

Which of the following sequences could be possible? Pick one answer A-F (only one is correct)

A) 1,2,3,4,5,6  
B) 4,1,3,5,2,6  
C) 1,3,4,2,5,6  
D) 1,4,3,2,5,6  
E) 5,3,2,1,4,6  
F) None of the above sequences are possible

(4) **Question 11.** Give the simplified memory cycles produced when the following one instruction is executed. Assume the PC contains $4007$, and the SP equals $3FF4$. Just show R/W=Read or Write, Address, and Data for each cycle. You may or may not need all 5 entries in the solution box.

\hspace{20pt} $4007$ 164200 \hspace{5pt} jsr $4200$

<table>
<thead>
<tr>
<th>R/W</th>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
(4) Question 12. Consider a serial port operating with a baud rate of 2000 bits per second. The following waveform was measured on the PS1 output (voltage levels are +5 and 0) when one SCI occurs. The protocol is 1 start, 8 data and 1 stop bit. What data in hexadecimal was transmitted? You may assume the channel is idle before and after the frame. Time flows from left to right.

![Waveform Diagram](image)

(24) Question 13. In this problem you must use a C data structure that stores this Moore FSM. The system is an AC thermostat, if the AC is off and temperature rises above 70 °F, then the AC comes on. If the AC is on and the temperature falls below 68 °F, the AC is shut off. If the temperature is between 68 and 70 °F, the AC remains in its present state. The temperature sensor is attached to PAD0, such that in 10-bit mode a temperature of 68 °F returns a 10-bit ADC value of 680, and a temperature of 70 °F returns a 10-bit ADC value of 700. The AC unit is controlled by PT0, such that if the software makes PT0=1, the AC is on. If the software makes PT0=0, the AC is off. This hysteresis avoids the rapid on-off-on-off instability that would occur if the temperature is near the set point.

The controller should be run once a second in the background using output compare 0 interrupts. For each execution, the ISR software should sample PAD0, compare it to the value in the current state, select the next state depending on whether or not the current temperature is above or below threshold, and finally the ISR should output the on/off command to PT0 as specified by the new state.

**Part a)** Show the C code that defines a linked structure for this FSM. Each state contains one output value, one temperature threshold (0.1 °F resolution), and two next states depending on whether the input is above or below the threshold. Fill in necessary code into the two boxes.

```c
const struct State{
};
typedef const struct State StateType;
typedef StateType * StatePtr;
#define ACon &fsm[0]
#define ACoff &fsm[1]
StateType fsm[2] = {
};
```

**Part b)** You are given a function `ADC_Init` that initializes the ADC in 10-bit mode with an ADC clock of 1 MHz. Write the main program that calls `ADC_Init`, initializes the FSM, sets up the output...
compare 0 interrupt, and enables interrupts. Assume the E clock is 8 MHz. The body of the main will be a do nothing loop, such as `while(1);` or `for(;;){};`

**Part c)** Write a C function that samples ADC channel 0 using busy-wait synchronization. ADC format should be right justified.

**Part d)** Write the output compare ISR in C that implements the FSM, interrupting every 1 sec.
(10) **Question 14.** All four parts constitute one assembly subroutine. In this problem you will implement three unsigned 16-bit local variables on the stack using Reg X stack frame addressing and symbolic binding. The variables are called **left**, **center** and **right**. The code in this question is part of a subroutine, which ends in `rts`.

Part a) Show the assembly code that (in this order) saves Register X, establishes the Register X stack frame, and allocates the three 16-bit local variables.

Part b) Assume the stack pointer is equal to $3F0A$ just before `jsr` instruction is executed that calls this subroutine. Execute `jsr` and all of part a) then draw the stack picture showing the return address, the three variables, Register X, and the stack pointer SP. Cross-out the SP arrow and move it to its new location.

Part c) Show the symbolic binding for **left**, **center** and **right**.

Part d) Show code that implements `center=100;` using Reg X stack frame addressing.

Part e) Show the assembly code that deallocates the local variables, and restores Reg X.

```
rts
```
(15) **Question 15.** Implement in assembly language a FIFO queue with the following specifications

1) Two 16-bit words are allocated in RAM to store data in the FIFO
2) One 8-bit counter is allocated in RAM to store the number of elements: 0, 1, or 2
3) If there is one element in the FIFO, it is stored in the first location of the FIFO
4) If there are two elements, the oldest is in the first location and the newest in the second

The following assembly code defines the FIFO in RAM. You can not make changes or additions to the way in which these variables are defined. You can NOT add more variables.

```assembly
org $2000
Fifo rmb 4  ; place for two 16-bit numbers
Count rmb 1  ; 0 means empty, 1 means half, 2 means full
```

**Part a)** Write an assembly subroutine to initialize the FIFO.

**Part b)** Write an assembly subroutine that puts one 16-bit element into the FIFO. The input parameter is call by value in Reg D, and the return parameter is call by value in Reg D: 0 meaning success, and 1 means the data was not stored because there were already two elements in the FIFO.

**Part c)** Write an assembly subroutine that gets one 16-bit element from the FIFO. At the time of the call, Reg X points to an empty place into which the data can be stored. There are two output parameters. The data is returned by reference using the pointer passed in Reg X. The other return parameter is return by value in Reg D: 0 meaning success, and 1 means the data was not removed because there were no elements in the FIFO at the time of the call. Here is an example call

```assembly
MyData rmb 2  ;My variable is different from your fifo

MyProgram
  ldx #MyData  ;pointer to empty place
  jsr YourGetFifo  ; your program puts 16-bit data into MyData
```
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
addc 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 RegA
aslb/lslb 8-bit arithmetic left shift RegB
asld/asld 16-bit left shift RegD
asr 8-bit arithmetic right shift/Memory
asra 8-bit arithmetic right shift to RegA
asrb 8-bit arithmetic right shift to RegB
bcc branch if carry clear
bclr bit clear in memory bclr PTT,$01
bcs branch if carry set
beq branch if result is zero (Z=1)
beq branch if signed ≥
beq branch if signed >
beq branch if unsigned ≥
beq branch if unsigned >
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 ≤
blb 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)
 bpl branch if result is positive (N=0)
bra branch always
brccl branch if bits are clear brccl PTT,$01,loop
brn branch never
brset branch if bits are set brset PTT,$01,loop
bsb set bit in memory bsb PTT,$01
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, RegA-RegB
clc clear carry bit, C=0
cli clear I=0, enable interrupts
clr 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
cmpc 8-bit compare RegB with memory
cmpd 8-bit logical complement to memory
cmpda 8-bit logical complement to RegA
cmpdb 8-bit logical complement to RegB
cmpdc 8-bit logical complement to RegD
cp 16-bit compare RegD with memory
cpx 16-bit compare RegX with memory
cpy 16-bit compare RegY with memory
da 8-bit decrement RegA
dba 8-bit decrement RegB
dbc 8-bit decrement RegC
dbeq decrement and branch if result=0 dbeq Y,loop
dbeq decrement and branch if result=0 dbeq A,loop
dec 8-bit decrement memory
deb 8-bit decrement RegA
debc 8-bit decrement RegB
deq 16-bit decrement RegX
dey 16-bit decrement RegY
div RegY=(Y:D)/RegX, 32-bit by 16-bit unsigned divide
divs RegY=(Y:D)/RegX, 32-bit by 16-bit signed divide
dmacs 16 by 16 signed multiply, 32-bit add
dmax 16-bit unsigned maximum in RegD
maxm 16-bit unsigned maximum in memory
dmax 16-bit unsigned minimum in RegD
mind 16-bit unsigned minimum in memory
dmul RegY=RegY*RegD, 512 by 16-bit unsigned multiply
dmuls RegY=RegY*RegD, 16 by 16 to 32-bit signed multiply
dora 8-bit logical exclusive or to RegA
dorb 8-bit logical exclusive or to RegB
dtcp 16-bit look up and interpolation
dxg exchange register contents dxg X,Y
div unsign f/frac div, X=(65536*D)/X
dbeq increment and branch if result=0 dbeq Y,loop
dbne increment and branch if result≠0 dbne A,loop
didiv 16-bit by 16-bit unsigned div, X=D/X, D=remainder
didivs 16-bit by 16-bit signed divide, X=D/X, D=remainder
dinc 8-bit increment memory
dicina 8-bit increment int RegA
dincb 8-bit increment RegB
dins 16-bit increment RegSP
dinx 16-bit increment RegX
diy 16-bit increment RegY
djmp jump always
jsr jump to subroutine
lbcc long branch if carry clear
lbc set branch if carry set
lbne long branch if result is zero
lbeq long branch if result is equal
lbge long branch if signed ≥
lbgt long branch if signed >
lbhi branch if unsigned >
lbhs branch if unsigned ≥
lbl long branch if signed ≤
lbls branch if unsigned <
lbmi branch if result is negative
lbne branch if result is nonzero
lbpl long branch if result is positive
lbra branch always
lbcc set branch if carry clear
lbcc set branch if carry set
lbne long branch if result is equal
lbeq long branch if result is zero
lbge long branch if signed ≥
lbgt long branch if signed >
lbhi branch if unsigned >
lbhs branch if unsigned ≥
lbl long branch if signed ≤
lbls branch if unsigned <
lbmi branch if result is negative
lbne branch if result is nonzero
lbpl long branch if result is positive
lbra branch always
lbcc set branch if carry clear
lbcc set branch if carry set
lbne long branch if result is equal
lbeq long branch if result is zero
lbge long branch if signed ≥
ldaa 8-bit load memory into RegA
ldab 8-bit load memory into RegB
lddd 16-bit load memory into RegD
ldds 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 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
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
max 8-bit unsigned maximum in RegA
maxm 8-bit unsigned maximum in memory
mem determine the Fuzzy logic membership grade
min 8-bit unsigned minimum in RegA
minm 8-bit unsigned minimum in memory
movb 8-bit move memory to memory movb #100,PTT
movw 16-bit move memory to memory  
mul 8 by 8 to 16-bit unsigned RegD=RegA*RegB  
neg 8-bit’s complement negate memory  
neqa 8-bit’s complement negate RegA  
neqb 8-bit’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  
pult 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 sex B,D  
sta 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  
tbne test and branch if result≠0  
tfr transfer register to register  
tpa transfer CC to A  
trap illegal instruction interrupt  
trap illegal op code, or software trap  
st 8-bit compare memory with zero  
staa 8-bit compare RegA with zero  
strstr 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  
xgy exchange RegD with RegY

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

Freescale 6812 addressing modes r is X, Y, SF, or FC

Pseudo op Meaning
= equ set Where to put subsequent code
dc.b db fcb .byte Allocate byte(s) with values
fcc Create an ASCII string
dc.w dw fdb .word Allocate word(s) with values
dc.l dl .long Allocate 32-bit with values
ds ds b rmb .blkb Allocate bytes without init
ds w .blkw Allocate words without init

n is Metrowerks number

<table>
<thead>
<tr>
<th>Vector</th>
<th>n</th>
<th>Interrupt Source</th>
<th>Arm</th>
</tr>
</thead>
<tbody>
<tr>
<td>$FFFF</td>
<td>Reset</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>$FFFF8</td>
<td>3</td>
<td>Trap</td>
<td>None</td>
</tr>
<tr>
<td>$FFFF6</td>
<td>4</td>
<td>SWI</td>
<td>None</td>
</tr>
<tr>
<td>$FFFF0</td>
<td>7</td>
<td>Real time interrupt</td>
<td>CRGINT.RTIE</td>
</tr>
<tr>
<td>$FFEE</td>
<td>8</td>
<td>Timer channel 0</td>
<td>TIE.C0I</td>
</tr>
<tr>
<td>$FFEC</td>
<td>9</td>
<td>Timer channel 1</td>
<td>TIE.C1I</td>
</tr>
<tr>
<td>$FFEA</td>
<td>10</td>
<td>Timer channel 2</td>
<td>TIE.C2I</td>
</tr>
<tr>
<td>$FFE8</td>
<td>11</td>
<td>Timer channel 3</td>
<td>TIE.C3I</td>
</tr>
<tr>
<td>$FFE6</td>
<td>12</td>
<td>Timer channel 4</td>
<td>TIE.C4I</td>
</tr>
<tr>
<td>$FFE4</td>
<td>13</td>
<td>Timer channel 5</td>
<td>TIE.C5I</td>
</tr>
<tr>
<td>$FFE2</td>
<td>14</td>
<td>Timer channel 6</td>
<td>TIE.C6I</td>
</tr>
<tr>
<td>$FFE0</td>
<td>15</td>
<td>Timer channel 7</td>
<td>TIE.C7I</td>
</tr>
<tr>
<td>$FFDE</td>
<td>16</td>
<td>Timer overflow</td>
<td>TSCR2.TOI</td>
</tr>
<tr>
<td>$FFD6</td>
<td>20</td>
<td>SCI0 TDRE, RDRF</td>
<td>SCI0CR2.TIE,RIE</td>
</tr>
<tr>
<td>$FFD4</td>
<td>21</td>
<td>SCI1 TDRE, RDRF</td>
<td>SCI1CR2.TIE,RIE</td>
</tr>
<tr>
<td>$FFCE</td>
<td>24</td>
<td>Key Wakeup J</td>
<td>PIEJ,[7,6,1,0]</td>
</tr>
<tr>
<td>$FFCC</td>
<td>25</td>
<td>Key Wakeup H</td>
<td>PIEH,[7:0]</td>
</tr>
<tr>
<td>$FF8E</td>
<td>56</td>
<td>Key Wakeup P</td>
<td>PIEP,[7:0]</td>
</tr>
</tbody>
</table>

Interrupt Vectors and interrupt number.
<table>
<thead>
<tr>
<th>Address</th>
<th>Bit 7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>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>$0044-5</td>
<td>Bit 15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>TCNT</td>
</tr>
<tr>
<td>$0046</td>
<td>TEN</td>
<td>TSWSI</td>
<td>TSFRZ</td>
<td>TFFCA</td>
<td>C7I</td>
<td>C5I</td>
<td>C4I</td>
<td>C2I</td>
<td>C1I</td>
</tr>
<tr>
<td>$004D</td>
<td>TOI</td>
<td>0</td>
<td>PUP</td>
<td>RDPT</td>
<td>TCNT</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>$0050-1</td>
<td>Bit 15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>TC0</td>
</tr>
<tr>
<td>$0052-3</td>
<td>Bit 15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>TC1</td>
</tr>
<tr>
<td>$0054-5</td>
<td>Bit 15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>TC2</td>
</tr>
<tr>
<td>$0056-7</td>
<td>Bit 15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>TC3</td>
</tr>
<tr>
<td>$0058-9</td>
<td>Bit 15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>TC4</td>
</tr>
<tr>
<td>$005A-B</td>
<td>Bit 15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>TC5</td>
</tr>
<tr>
<td>$005C-D</td>
<td>Bit 15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>TC6</td>
</tr>
<tr>
<td>$005E-F</td>
<td>Bit 15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>TC7</td>
</tr>
<tr>
<td>$0082</td>
<td>ADPU</td>
<td>AFFC</td>
<td>ASWA</td>
<td>ETTRIGL</td>
<td>ETTRIGP</td>
<td>ETTRIG</td>
<td>ASCIE</td>
<td>ASCIF</td>
<td>ATD0CTL2</td>
</tr>
<tr>
<td>$0083</td>
<td>0</td>
<td>S8C</td>
<td>S4C</td>
<td>S2C</td>
<td>S1C</td>
<td>FI0</td>
<td>FR2</td>
<td>FR0</td>
<td>ATD0CTL3</td>
</tr>
<tr>
<td>$0084</td>
<td>SRES8</td>
<td>SMPI</td>
<td>SMP0</td>
<td>PRS4</td>
<td>PRS3</td>
<td>PRS2</td>
<td>PRS1</td>
<td>PRS0</td>
<td>ATD0CTL4</td>
</tr>
<tr>
<td>$0085</td>
<td>DJSN</td>
<td>SCN</td>
<td>MULT</td>
<td>0</td>
<td>CC</td>
<td>CB</td>
<td>CA</td>
<td>ATD0CTL5</td>
<td></td>
</tr>
<tr>
<td>$0086</td>
<td>SCF</td>
<td>0</td>
<td>ETORF</td>
<td>FFOR</td>
<td>0</td>
<td>CC2</td>
<td>CC1</td>
<td>CC0</td>
<td>ATD0STAT0</td>
</tr>
<tr>
<td>$008B</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>ATD0STAT1</td>
</tr>
<tr>
<td>$008D</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>$008F</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>$0090-1</td>
<td>Bit 15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>ATD0DR0</td>
</tr>
<tr>
<td>$0092-3</td>
<td>Bit 15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>ATD0DR1</td>
</tr>
<tr>
<td>$0094-5</td>
<td>Bit 15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>ATD0DR2</td>
</tr>
<tr>
<td>$0096-7</td>
<td>Bit 15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>ATD0DR3</td>
</tr>
<tr>
<td>$0098-9</td>
<td>Bit 15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>ATD0DR4</td>
</tr>
<tr>
<td>$009A-B</td>
<td>Bit 15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>ATD0DR5</td>
</tr>
<tr>
<td>$009C-D</td>
<td>Bit 15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>ATD0DR6</td>
</tr>
<tr>
<td>$009E-F</td>
<td>Bit 15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>ATD0DR7</td>
</tr>
<tr>
<td>$00CA</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>BBR12</td>
<td>BBR11</td>
<td>BBR10</td>
<td>SBR0</td>
<td>SCIO8BD</td>
<td></td>
</tr>
<tr>
<td>$00CB</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SCIOCR1</td>
</tr>
<tr>
<td>$00CC</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SCIOCR2</td>
</tr>
<tr>
<td>$00CD</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SCI0SR1</td>
</tr>
<tr>
<td>$00CF</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SCI0SR2</td>
</tr>
<tr>
<td>$00D0-1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SCI0DRL</td>
</tr>
<tr>
<td>$00D2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SCI1BD</td>
</tr>
<tr>
<td>$00D3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SCI1CR1</td>
</tr>
<tr>
<td>$00D4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SCI1CR2</td>
</tr>
<tr>
<td>$00D5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SCI1SR1</td>
</tr>
<tr>
<td>$00D7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SCI1SR2</td>
</tr>
<tr>
<td>$00D9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SCI1DRL</td>
</tr>
<tr>
<td>$00D2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SCI1DR0</td>
</tr>
<tr>
<td>$00D3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SCI1DR1</td>
</tr>
<tr>
<td>$00D4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SCI1DR2</td>
</tr>
<tr>
<td>$00D5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SCI1DR3</td>
</tr>
<tr>
<td>$00D7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SCI1DR4</td>
</tr>
<tr>
<td>$00D9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SCI1DR5</td>
</tr>
<tr>
<td>$000A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SCI1DR6</td>
</tr>
<tr>
<td>$000B</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SCI1DR7</td>
</tr>
<tr>
<td>$000C</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SCI1DR8</td>
</tr>
</tbody>
</table>

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

- bit 7 **TEN**, 1 allows the timer to function normally, 0 means disable timer including **TCNT**

**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)
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

<table>
<thead>
<tr>
<th>PR2</th>
<th>PR1</th>
<th>PR0</th>
<th>Divide by</th>
<th>E = 8 MHz</th>
<th>E = 24 MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>TCNT</td>
<td>TCNT frequency</td>
<td>TCNT</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>125 ns</td>
<td>8 MHz</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>250 ns</td>
<td>4 MHz</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>500 ns</td>
<td>2 MHz</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>1 ( \mu )s</td>
<td>1 MHz</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>16</td>
<td>2 ( \mu )s</td>
<td>500 kHz</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>32</td>
<td>4 ( \mu )s</td>
<td>250 kHz</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>64</td>
<td>8 ( \mu )s</td>
<td>125 kHz</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>128</td>
<td>16 ( \mu )s</td>
<td>62.5 kHz</td>
</tr>
</tbody>
</table>

SCI0DRL 8-bit SCI0 data register
SCI0BD is 16-bit SCI0 baud rate register, let \( n \) be the 13-bit number Baud rate is EClk/\( n/16 \)

SCI0CR1 is 8-bit SCI0 control register
bit 4 M, Mode, 0 = One start, eight data, one stop bit, 1 = One start, eight data, ninth data, one stop bit

SCI0CR2 is 8-bit SCI0 control register
bit 7 TIE, Transmit Interrupt Enable, 0 = TDRE interrupts disabled, 1 = interrupt whenever TDRE set
bit 5 RIE, Receiver Interrupt Enable, 0 = RDRF interrupts disabled, 1 = interrupt whenever RDRF set
bit 3 TE, Transmitter Enable, 0 = Transmitter disabled, 1 = SCI transmit logic is enabled
bit 2 RE, Receiver Enable, 0 = Receiver disabled, 1 = Enables the SCI receive circuitry.

SCI0SR1 is 8-bit SCI0 status register
bit 7 TDRE, Transmit Data Register Empty Flag
Set if transmit data can be written to SCI0DRL
Cleared by SCI0SR1 read with TDRE set followed by SCI0DRL write
bit 5 RDRF, Receive Data Register Full
set if a received character is ready to be read from SCI0DRL
Clear the RDRF flag by reading SCI0SR1 with RDRF set and then reading SCI0DRL

ATD0CTL5 is used to start an ADC conversion
bit 7 DJM is set to 1 for right justified and to 0 for left justified
bits 2-0 specify the ADC channel to sample
ATD0STAT0 is used to tell when the ADC conversion is done
bit 7 SCF cleared on a write to ATD0CTL5 and is set when the conversion sequence is done

### JSR
Jump to Subroutine

**Operation:**

\[(S P) - S 0 0 0 2 \Rightarrow S P \]

RTNH, RTNL, \( M_{SP} \), \( M_{SP+1} \)

Subroutine Address = PC

### ADDA
Add without Carry to A

**Operation:**

\((A) + (M) \Rightarrow A\)

V: A7 • M7 • R7 + A7 • M7 • R7
Set if two's complement overflow resulted from the operation; cleared otherwise

C: A7 • M7 + M7 • R7 + R7 • A7
Set if there was a carry from the MSB of the result; cleared otherwise