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 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.

__________

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 to see if all the changes in output are positive.

__________

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 averaging all the changes in output.

__________

Part d) 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.

__________

(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 800 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 is resistance is needed for R in the circuit so the output voltage Vout is 3V?

![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 `location` is allocated in RAM. Figure out the smallest number of bytes needed to allocate `location` and write your code accordingly.

**Part a)** Write assembly code that multiplies the variable by 0.25 storing the result back into `location`. For example, if the initial value is 2.4 cm, the final value will be 0.6 cm.

**Part b)** Write assembly code that adds 1.0 cm to the variable storing the result back into `location`. For example, if the initial value is 2.4 cm, the final value will be 3.4 cm. You do not need to consider overflow.

(4) **Question 5.** Assume the SCI0 is already initialized; write a C function that transmits one character using busy-wait synchronization.
(4) **Question 6.** The desired LED operating point is 1V, 10mA. Assume the $V_{OL}$ of the 7406 is 0.5V. Interface this LED to PP0 using positive logic. Specify values for any resistors needed.

```
9S12
PP0
```

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

```
pshb
stx 2,-sp
sty 1,sp-
leas 2,sp
pulx
```

(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</th>
<th>D,X</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LDY</td>
<td>0,X</td>
</tr>
<tr>
<td></td>
<td>LDD</td>
<td>#314</td>
</tr>
<tr>
<td></td>
<td>EMUL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LDX</td>
<td>#1000</td>
</tr>
<tr>
<td></td>
<td>IDIVS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TFR</td>
<td>X,D</td>
</tr>
<tr>
<td></td>
<td>RTS</td>
<td></td>
</tr>
</tbody>
</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.

```
ldaa #156
suba #-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 7 interrupt.
1) The TCNT equals TC7 and the hardware sets the flag bit (e.g., C7F=1)
2) The PC is set to the contents of the output compare 7 vector
3) The CCR, A, B, X, Y, PC are pushed on the stack
4) The I bit in the CCR is set by hardware
5) The software executes something like
```
movb #$80, TFLG1
1dd  TC7
addd #5000
std  TC7
```
6) The software executes 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 $4005, and the SP equals $3FF8. 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.

```
$4005 164200   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](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 PAD1, 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 ten times a second in the background using output compare 1 interrupts. For each execution, the ISR software should sample PAD1, 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{
    int output;
    float threshold;
    StateType next_state_above;
    StateType next_state_below;
};

typedef const struct State StateType;
typedef StateType * StatePtr;
#define On &Machine[0]
#define Off &Machine[1]
StateType Machine[2]={
    Iinit,  // Initialize init state
    ACon, // Temperature above 70°F
    AcOff, // Temperature below 68°F
    ACon, // Temperature between 68 and 70°F
    AcOff, // Temperature below 68°F
    ACon // Temperature above 70°F
};
```

Jonathan Valvano  December 8, 2010
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 1 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 1 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 100ms.
(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.

```
org $2000
Buf rmb 4 ; place for two 16-bit numbers
Size 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.

```
ldx #MyData ;pointer to empty place
jsr YourGetFifo ; your program puts 16-bit data into MyData
```

**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

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

```
MyProgram
   ld #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
adddd 16-bit add to RegD
anda 8-bit logical and to RegA
andb 8-bit logical and to RegB
anddce 8-bit logical and to RegCC
asl/lsl 8-bit left shift RegB
asl/lsla 8-bit left shift RegA
aslb/lslb 8-bit arithmetic left shift RegB
asld/lslld 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
bclrt PTT,#$01
bcs branch if carry set
beq branch if result is zero (Z=1)
bege branch if signed ≥
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 ≤
blt branch if signed <
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)
brt branch always
brclr branch if bits are clear
brclrt PTT,#$01,loop
brn branch never
brset branch if bits are set
bset bit set in memory
bset PTT,#$04
bshr 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
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
cpda 16-bit compare RegD with memory
cpdx 16-bit compare RegX with memory
cpy 16-bit compare RegY with memory
daaa 8-bit decimal adjust accumulator
dbeq decrement and branch if result=0
dbeq Y,loop
dbn decremend and branch if result=0
dbne A,loop
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
div RegY=(Y:D)/RegX, 32-bit by 16-bit unsigned divide
divs RegY=(Y:D)/RegX, 32-bit by 16-bit signed divide
divu 16-bit unsigned divide
emacs 16 by 16 signed multiply, 32-bit add
emaxd 16-bit unsigned maximum in RegD
emaxm 16-bit unsigned maximum in memory
eminm 16-bit unsigned minimum in memory
emul RegY=D RegY=D, 16 by 16 to 32-bit unsigned multiply
emuls RegY=D* RegY=D, 16 by 16 to 32-bit 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 frac div, X=(65536*D)/X
ibeq increment and branch if result=0
ibne increment and branch if result≠0
idiv 16-bit by 16-bit unsigned div, X=D/X, D=remainder
idivs 16-bit by 16-bit signed divide, X=D/X, D=remainder
inc 8-bit increment memory
inc 8-bit increment RegA
incb 8-bit increment RegB
ins 16-bit increment RegSP
inx 16-bit increment RegX
inyc 16-bit increment RegY
jmp jump always
jsr jump to subroutine
lbcc long branch if carry clear
lbcwc long branch if carry set
lbeg long branch if result is zero
lbge long branch if signed ≥
lbglt branch if signed >
lbhi long branch if unsigned >
lbhs long branch if unsigned ≥
lble long branch if signed ≤
lblc long branch if unsigned <
lbls long branch if unsigned ≤
lt bit 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
lbpl long branch always
lbrc long branch never
lbvc long branch if overflow clear
lvs long branch if overflow set
ldaa 8-bit load memory into RegA
ldab 8-bit load memory into RegB
lddd 16-bit load memory into RegD
lds 16-bit load memory into RegSP
ldx 16-bit load memory into RegX
ldyx 16-bit load memory into RegY
ldyd 16-bit load effective addr to SP
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
maxa 8-bit unsigned maximum in RegA
maxm 8-bit unsigned maximum in memory
mem determine the Fuzzy logic membership grade
mina 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 2's complement negate memory
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
puly pop 16 bits off stack into RegX
pulx pop 16 bits off stack into RegY
rev Fuzzy logic rule evaluation
revv 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
rotc 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
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
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 tbeq Y,loop
tbl 8-bit look up and interpolation
tbne test and branch if result≠0 tbne A,loop
tfr transfer register to register tfr X,Y
tpa transfer CC to A
trap illegal instruction interrupt
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

Example | Mode | Effective Address
---|---|---
ldaa #u | immediate | No EA
ldaa u | direct | EA is 8-bit address
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-inc | r=r+v, EA=r (1 to 8)
ldaa v,-r | pre-dec | r=r-v, EA=r (1 to 8)
ldaa v,r+ | post-inc | EA=r, r=r+v (1 to 8)
ldaa v,r- | post-dec | 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
ldaa W,r | 16-bit index | EA=r+W
ldaa [D,r] | D indirect | EA=[r+D]
ldaa [W,r] | indirect | EA=[r+W]

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

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

n is Metrowerks number
Vector | n | Interrupt | Source | Arm
---|---|---|---|---
SFFE | Reset | None
SFFF | 3 | Trap | None
SFFF6 | 4 | SWI | None
SFFF0 | 7 | Real time interrupt | CRGINT.RTIE
SFFE8 | 8 | Timer channel 0 | TIE.C0I
SFFEC | 9 | Timer channel 1 | TIE.C1I
SFFE8A | 10 | Timer channel 2 | TIE.C2I
SFFE8 | 11 | Timer channel 3 | TIE.C3I
SFFE6 | 12 | Timer channel 4 | TIE.C4I
SFFE4 | 13 | Timer channel 5 | TIE.C5I
SFFE2 | 14 | Timer channel 6 | TIE.C6I
SFFE0 | 15 | Timer channel 7 | TIE.C7I
SFFDE | 16 | Timer overflow | TSCR2.TOIs
SFFD6 | 20 | SCI0 TDRE, RDRF | SCI0CR2.TIE.RIE
SFFD4 | 21 | SCI1 TDRE, RDRF | SCI1CR2.TIE.RIE
SFFCE | 24 | Key Wakeup J | PIEJ.[7,6,1,0]
SFFCC | 25 | Key Wakeup H | PIEH.[7:0]
SFF8E | 56 | Key Wakeup P | PIEP.[7:0]

Interrupt Vectors and interrupt number.
### Address

<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 Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0040</td>
<td>I05</td>
<td>I06</td>
<td>I05</td>
<td>I04</td>
<td>I03</td>
<td>I02</td>
<td>I01</td>
<td>I00 TIOS</td>
</tr>
<tr>
<td>S0044-5</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  Bit 0 TCNT</td>
</tr>
<tr>
<td>S0046</td>
<td>TEN</td>
<td>TSMAI</td>
<td>TSFRZ</td>
<td>FFCA</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0  TSCR1</td>
</tr>
<tr>
<td>S004C</td>
<td>C07</td>
<td>C06</td>
<td>C05</td>
<td>C04</td>
<td>C03</td>
<td>C02</td>
<td>C01</td>
<td>C00 TIE</td>
</tr>
<tr>
<td>S004D</td>
<td>TOI</td>
<td>0</td>
<td>PUP1</td>
<td>RDPF</td>
<td>TCRR</td>
<td>PR2</td>
<td>PR1</td>
<td>PR0 TSCR2</td>
</tr>
<tr>
<td>S004E</td>
<td>C0F</td>
<td>C0F</td>
<td>C0F</td>
<td>C0F</td>
<td>C0F</td>
<td>C0F</td>
<td>C0F</td>
<td>Bit 0 TFLG1</td>
</tr>
<tr>
<td>S004F</td>
<td>TOF</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>S0050-1</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  Bit 0 TC0</td>
</tr>
<tr>
<td>S0052-3</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  Bit 0 TC1</td>
</tr>
<tr>
<td>S0054-5</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  Bit 0 TC2</td>
</tr>
<tr>
<td>S0056-7</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  Bit 0 TC3</td>
</tr>
<tr>
<td>S0058-9</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  Bit 0 TC4</td>
</tr>
<tr>
<td>S005A-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  Bit 0 TC5</td>
</tr>
<tr>
<td>S005C-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  Bit 0 TC6</td>
</tr>
<tr>
<td>S005E-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  Bit 0 TC7</td>
</tr>
<tr>
<td>S0082</td>
<td>ADPU</td>
<td>AFFC</td>
<td>ASWAI</td>
<td>ETRIGE</td>
<td>ETRIGP</td>
<td>ETRIG</td>
<td>ASCIE</td>
<td>ASCIF</td>
</tr>
<tr>
<td>S0083</td>
<td>0</td>
<td>S8C</td>
<td>S4C</td>
<td>S2C</td>
<td>S1C</td>
<td>FFO</td>
<td>FRZ1</td>
<td>FRZ0</td>
</tr>
<tr>
<td>S0084</td>
<td>SRES5</td>
<td>SMP1</td>
<td>SMP0</td>
<td>PRS4</td>
<td>PRS3</td>
<td>PRS2</td>
<td>PRS1</td>
<td>PRS0</td>
</tr>
<tr>
<td>S0085</td>
<td>DJM</td>
<td>DSGN</td>
<td>SCAN</td>
<td>MULT</td>
<td>0</td>
<td>CC</td>
<td>CA</td>
<td>0</td>
</tr>
<tr>
<td>S0086</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>
</tr>
<tr>
<td>S008B</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>
</tr>
<tr>
<td>S008D</td>
<td>Bit 7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>Bit 0 ATD0DIEN</td>
</tr>
<tr>
<td>S008F</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>
</tr>
<tr>
<td>S0090-1</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  Bit 0 ATD0DR0</td>
</tr>
<tr>
<td>S0092-3</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  Bit 0 ATD0DR1</td>
</tr>
<tr>
<td>S0094-5</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  Bit 0 ATD0DR2</td>
</tr>
<tr>
<td>S0096-7</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  Bit 0 ATD0DR3</td>
</tr>
<tr>
<td>S0098-9</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  Bit 0 ATD0DR4</td>
</tr>
<tr>
<td>S009A-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  Bit 0 ATD0DR5</td>
</tr>
<tr>
<td>S009C-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  Bit 0 ATD0DR6</td>
</tr>
<tr>
<td>S009E-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  Bit 0 ATD0DR7</td>
</tr>
<tr>
<td>S00C9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SBR12</td>
<td>SBR11</td>
<td>SBR10</td>
<td>SBR0</td>
<td>SC08BD</td>
</tr>
<tr>
<td>S00CA</td>
<td>LOOP</td>
<td>SCISWAI</td>
<td>RSRC</td>
<td>M</td>
<td>WAKE</td>
<td>ILT</td>
<td>PE</td>
<td>PT</td>
</tr>
<tr>
<td>S00CB</td>
<td>TIE</td>
<td>TCIE</td>
<td>RIE</td>
<td>ILIE</td>
<td>TE</td>
<td>RE</td>
<td>RUW</td>
<td>SBK</td>
</tr>
<tr>
<td>S00CC</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>
</tr>
<tr>
<td>S00CD</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>BRK13</td>
<td>TXDIR</td>
</tr>
<tr>
<td>S00CF</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>
</tr>
<tr>
<td>S00D0-1</td>
<td>Bit 7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>Bit 0 SBR0</td>
</tr>
<tr>
<td>S00D2</td>
<td>LOOP</td>
<td>SCISWAI</td>
<td>RSRC</td>
<td>M</td>
<td>WAKE</td>
<td>ILT</td>
<td>PE</td>
<td>PT</td>
</tr>
<tr>
<td>S00D3</td>
<td>TIE</td>
<td>TCIE</td>
<td>RIE</td>
<td>ILIE</td>
<td>TE</td>
<td>RE</td>
<td>RUW</td>
<td>SBK</td>
</tr>
<tr>
<td>S00D4</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>
</tr>
<tr>
<td>S00D5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>BRK13</td>
<td>TXDIR</td>
</tr>
<tr>
<td>S00D7</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>
</tr>
<tr>
<td>S00D8</td>
<td>PJ7</td>
<td>PJ6</td>
<td>PJ5</td>
<td>PJ4</td>
<td>PJ3</td>
<td>PJ2</td>
<td>PJ1</td>
<td>PT0</td>
</tr>
<tr>
<td>S0242</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>
</tr>
<tr>
<td>S0248</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>
</tr>
<tr>
<td>S024A</td>
<td>DDRS7</td>
<td>DDRS6</td>
<td>DDRS5</td>
<td>DDRS4</td>
<td>DDRS3</td>
<td>DDRS2</td>
<td>DDRS1</td>
<td>DDRS0</td>
</tr>
<tr>
<td>S0250</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>
</tr>
<tr>
<td>S0252</td>
<td>DDM7</td>
<td>DDM6</td>
<td>DDM5</td>
<td>DDM4</td>
<td>DDM3</td>
<td>DDM2</td>
<td>DDM1</td>
<td>DDM0</td>
</tr>
<tr>
<td>S0258</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>
</tr>
<tr>
<td>S025A</td>
<td>DDRP7</td>
<td>DDRP6</td>
<td>DDRP5</td>
<td>DDRP4</td>
<td>DDRP3</td>
<td>DDRP2</td>
<td>DDRP1</td>
<td>DDRP0</td>
</tr>
<tr>
<td>S0260</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>
</tr>
<tr>
<td>S0262</td>
<td>DDDR7</td>
<td>DDDR6</td>
<td>DDDR5</td>
<td>DDDR4</td>
<td>DDDR3</td>
<td>DDDR2</td>
<td>DDDR1</td>
<td>DDDR0</td>
</tr>
<tr>
<td>S0268</td>
<td>PJ7</td>
<td>PJ6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>PTJ</td>
</tr>
<tr>
<td>S026A</td>
<td>DDRJ7</td>
<td>DDRJ6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>DDRJ1</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 \( 8 \text{MHz}/2^n \), with PLL, **TCNT** is \( 24 \text{MHz}/2^n \), \( n \) ranges from 0 to 7

<table>
<thead>
<tr>
<th></th>
<th>PR2</th>
<th>PR1</th>
<th>PR0</th>
<th>Divide by</th>
<th>E = 8 MHz</th>
<th></th>
<th>E = 24 MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>125 ns</td>
<td>8 MHz</td>
<td>41.7 ns</td>
<td>24 MHz</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>250 ns</td>
<td>4 MHz</td>
<td>83.3 ns</td>
<td>12 MHz</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>500 ns</td>
<td>2 MHz</td>
<td>167 ns</td>
<td>6 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>
<td>333 ns</td>
<td>3 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>
<td>667 ns</td>
<td>1.5 MHz</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>32</td>
<td>4 ( \mu )s</td>
<td>250 kHz</td>
<td>1.33 ( \mu )s</td>
<td>667 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>
<td>2.67 ( \mu )s</td>
<td>333 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>
<td>5.33 ( \mu )s</td>
<td>167 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:** \((SP) - \$0002 \Rightarrow SP\)

\[ RTN_h \cdot RTN_l = M_{(SP)} \cdot M_{(SP+1)} \]

Subroutine Address = PC

<table>
<thead>
<tr>
<th>Source Form</th>
<th>Address Mode</th>
<th>Object Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>JSR ap(r6)</td>
<td>DIR</td>
<td>17 dd</td>
</tr>
<tr>
<td>JSR ap(r16)</td>
<td>EXT</td>
<td>16 hh, ll</td>
</tr>
<tr>
<td>JSR ap(r0),(y),(sp)</td>
<td>IDX</td>
<td>15 db</td>
</tr>
<tr>
<td>JSR ap(r0),(y),(sp)</td>
<td>IDX+1</td>
<td>15 db, ef</td>
</tr>
<tr>
<td>JSR ap(r0),(y),(sp)</td>
<td>IDX+2</td>
<td>15 db, ++ ef</td>
</tr>
<tr>
<td>JSR [D],(y),(sp)</td>
<td>[IDX]</td>
<td>15 db</td>
</tr>
<tr>
<td>JSR [ap(r16),(y),(sp)]</td>
<td>[IDX+2]</td>
<td>15 db, ++ ef</td>
</tr>
</tbody>
</table>

**SUBA**

*Subtract A*

**Operation:** \((A) - (M) \Rightarrow A\)

**V:** \(A7 \cdot M7 \cdot R7 + A7 \cdot M7 \cdot R7\)
- Set if a twos complement overflow resulted from the operation; cleared otherwise

**C:** \(A7 \cdot M7 + M7 + R7 + R7 + A7\)
- Set if the value of the content of memory is larger than the value of the accumulator; cleared otherwise