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.

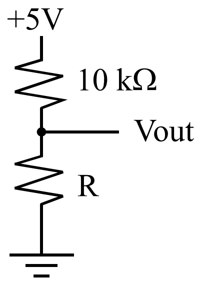
____

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?



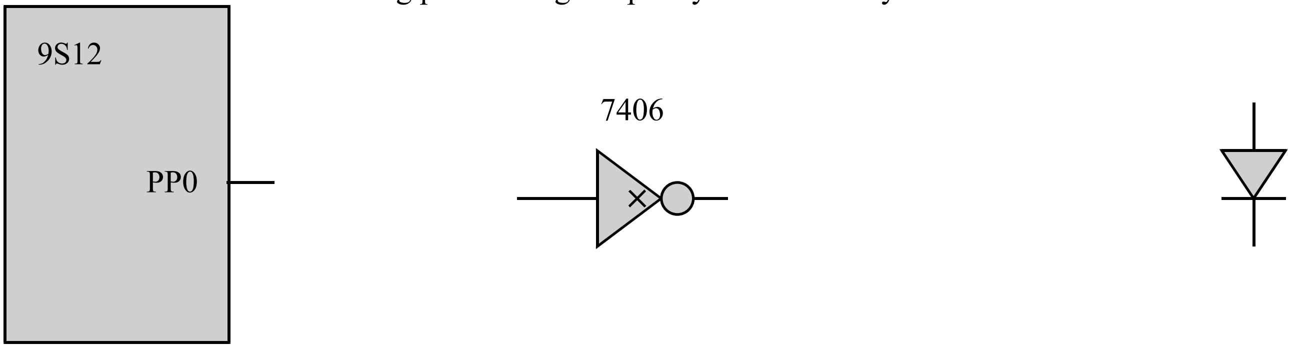
(6) Question 4. A measurement system has a range of 0 to 19.9 cm and a resolution of 0.1 cm. Assume a variable called **`position`** is allocated in RAM. Figure out the smallest number of bytes needed to allocate **`position`** and write your code accordingly.

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?

```
pshb
stx  2,-sp
sty  2,sp-
leas 3,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.

```
calc TFR   D,X
     LDY   0,X
     LDD   #314
     EMUL
     LDX   #1000
     IDIVS
     TFR   X,D
     RTS
```

```
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
        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 **TCNT** equals **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

```
   movb #$40,TFLG1
   ldd  TC6
   addd #5000
   std  TC6
```

6) The software executes `rti`

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

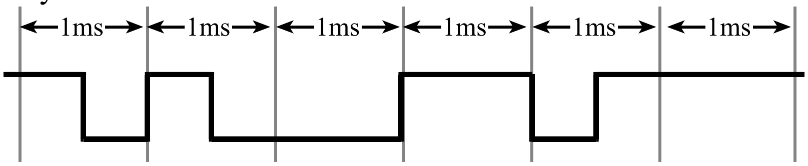
A) 1,2,3,4,5,6
B) 4,1,3,5,2,6 ___________
C) 1,3,4,2,5,6
D) 1,4,3,2,5,6
E) 5,3,2,1,4,6
F) None of the above sequences are possible

(4) Question 11. Give the simplified memory cycles produced when the following one instruction is executed. Assume the PC contains $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.

```
$4007 164200    jsr $4200
```

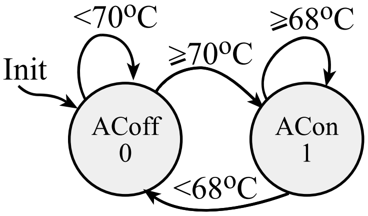
R/W	Address	Data

(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.



____

(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 ºF, then the AC comes on. If the AC is on and the temperature falls below 68 ºF, the AC is shut off. If the temperature is between 68 and 70 ºF, the AC remains in its present state. The temperature sensor is attached to PAD0, such that in 10-bit mode a temperature of 68 ºF returns a 10-bit ADC value of 680, and a temperature of 70 ºF 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 ºF resolution), and two next states depending on whether the input is above or below the threshold. Fill in necessary code into the two boxes.

```
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(;;){};`

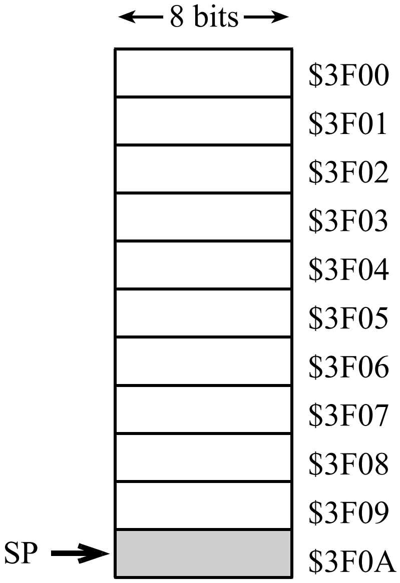
Part c) Write a C function that samples ADC channel 0 using busy-wait synchronization. ADC format should be right justified.

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.

```
          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

```
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	
`addd`	16-bit add to RegD	
`anda`	8-bit logical and to RegA	
`andb`	8-bit logical and to RegB	
`andcc`	8-bit logical and to RegCC	
`asl/lsl`	8-bit left shift Memory	
`asla/lsla`	8-bit left shift RegA	
`aslb/lslb`	8-bit arithmetic left shift RegB	
`asld/lsld`	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)	
`bge`	branch if signed ≥	
`bgnd`	enter background debug mode	
`bgt`	branch if signed >	
`bhi`	branch if unsigned >	
`bhs`	branch if unsigned ≥	
`bita`	8-bit and with RegA, sets CCR	
`bitb`	8-bit and with RegB, sets CCR	
`ble`	branch if signed ≤	
`blo`	branch if unsigned <	
`bls`	branch if unsigned ≤	
`blt`	branch if signed <	
`bmi`	branch if result is negative (N=1)	
`bne`	branch if result is nonzero (Z=0)	
`bpl`	branch if result is positive (N=0)	
`bra`	branch always	
`brclr`	branch if bits are clear	`brclr PTT,#$01,loop`
`brn`	branch never	
`brset`	branch if bits are set	`brset PTT,#$01,loop`
`bset`	bit set in memory	`bset PTT,#$04`
`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`	8-bit memory clear	
`clra`	RegA clear	
`clrb`	RegB clear	
`clv`	clear overflow bit, V=0	
`cmpa`	8-bit compare RegA with memory	
`cmpb`	8-bit compare RegB with memory	
`com`	8-bit logical complement to memory	
`coma`	8-bit logical complement to RegA	
`comb`	8-bit logical complement to RegB	
`cpd`	16-bit compare RegD with memory	
`cpx`	16-bit compare RegX with memory	
`cpy`	16-bit compare RegY with memory	
`daa`	8-bit decimal adjust accumulator	
`dbeq`	decrement and branch if result=0	`dbeq Y,loop`
`dbne`	decrement 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	
`ediv`	RegY=(Y:D)/RegX, 32-bit by 16-bit unsigned divide	
`edivs`	RegY=(Y:D)/RegX, 32-bit by 16-bit signed divide	
`emacs`	16 by 16 signed multiply, 32-bit add	
`emaxd`	16-bit unsigned maximum in RegD	
`emaxm`	16-bit unsigned maximum in memory	
`emind`	16-bit unsigned minimum in RegD	
`eminm`	16-bit unsigned minimum in memory	
`emul`	RegY:D=RegY*RegD, 16 by 16 to 32-bit unsigned multiply	
`emuls`	RegY:D=RegY*RegD, 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 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 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	
`inca`	8-bit increment RegA	
`incb`	8-bit increment RegB	
`ins`	16-bit increment RegSP	
`inx`	16-bit increment RegX	
`iny`	16-bit increment RegY	
`jmp`	jump always	
`jsr`	jump to subroutine	
`lbcc`	long branch if carry clear	
`lbcs`	long branch if carry set	
`lbeq`	long branch if result is zero	
`lbge`	long branch if signed ≥	
`lbgt`	long branch if signed >	
`lbhi`	long branch if unsigned >	
`lbhs`	long branch if unsigned ≥	
`lble`	long branch if signed ≤	
`lblo`	long branch if unsigned <	
`lbls`	long branch if unsigned ≤	
`lblt`	long branch if signed <	
`lbmi`	long branch if result is negative	
`lbne`	long branch if result is nonzero	
`lbpl`	long branch if result is positive	
`lbra`	long branch always	
`lbrn`	long branch never	
`lbvc`	long branch if overflow clear	
`lbvs`	long branch if overflow set	
`ldaa`	8-bit load memory into RegA	
`ldab`	8-bit load memory into RegB	
`ldd`	16-bit load memory into RegD	
`lds`	16-bit load memory into RegSP	
`ldx`	16-bit load memory into RegX	
`ldy`	16-bit load memory into RegY	
`leas`	16-bit load effective addr to SP	`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	`movw #13,SCIBD`
`mul`	8 by 8 to 16-bit unsigned RegD=RegA*RegB	
`neg`	8-bit 2's complement negate memory	
`nega`	8-bit 2's complement negate RegA	
`negb`	8-bit 2's complement negate RegB	
`oraa`	8-bit logical or to RegA	
`orab`	8-bit logical or to RegB	
`orcc`	8-bit logical or to RegCC	
`psha`	push 8-bit RegA onto stack	
`pshb`	push 8-bit RegB onto stack	
`pshc`	push 8-bit RegCC onto stack	
`pshd`	push 16-bit RegD onto stack	
`pshx`	push 16-bit RegX onto stack	
`pshy`	push 16-bit RegY onto stack	
`pula`	pop 8 bits off stack into RegA	
`pulb`	pop 8 bits off stack into RegB	
`pulc`	pop 8 bits off stack into RegCC	
`puld`	pop 16 bits off stack into RegD	
`pulx`	pop 16 bits off stack into RegX	
`puly`	pop 16 bits off stack into RegY	
`rev`	Fuzzy logic rule evaluation	
`revw`	weighted Fuzzy rule evaluation	
`rol`	8-bit roll shift left Memory	
`rola`	8-bit roll shift left RegA	
`rolb`	8-bit roll shift left RegB	
`ror`	8-bit roll shift right Memory	
`rora`	8-bit roll shift right RegA	
`rorb`	8-bit roll shift right RegB	
`rtc`	return sub in expanded memory	
`rti`	return from interrupt	
`rts`	return from subroutine	
`sba`	8-bit subtract RegA=RegA-RegB	
`sbca`	8-bit sub with carry from RegA	
`sbcb`	8-bit sub with carry from RegB	
`sec`	set carry bit, C=1	
`sei`	set I=1, disable interrupts	
`sev`	set overflow bit, V=1	
`sex`	sign extend 8-bit to 16-bit reg	`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	
`xgdx`	exchange RegD with RegX	
`xgdy`	exchange RegD with RegY	

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-incr	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 **`PC`**

Pseudo op	Meaning
`org`	Where to put subsequent code
`= equ set`	Define a constant symbol
`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 word(s) without init

n is Metrowerks number

Vector	**n**	Interrupt Source	Arm
$FFFE		**Reset**	None
$FFF8	3	**Trap**	None
$FFF6	4	**SWI**	None
$FFF0	7	**Real time interrupt**	CRGINT.RTIE
$FFEE	8	**Timer channel 0**	TIE.C0I
$FFEC	9	**Timer channel 1**	TIE.C1I
$FFEA	10	**Timer channel 2**	TIE.C2I
$FFE8	11	**Timer channel 3**	TIE.C3I
$FFE6	12	**Timer channel 4**	TIE.C4I
$FFE4	13	**Timer channel 5**	TIE.C5I
$FFE2	14	**Timer channel 6**	TIE.C6I
$FFE0	15	**Timer channel 7**	TIE.C7I
$FFDE	16	**Timer overflow**	TSCR2.TOI
$FFD6	20	**SCI0 TDRE, RDRF**	SCI0CR2.TIE,RIE
$FFD4	21	**SCI1 TDRE, RDRF**	SCI1CR2.TIE,RIE
$FFCE	24	**Key Wakeup J**	PIEJ.[7,6,1,0]
$FFCC	25	**Key Wakeup H**	PIEH.[7:0]
$FF8E	56	**Key Wakeup P**	PIEP.[7:0]

Interrupt Vectors and interrupt number.

Address	Bit 7	6	5	4	3	2	1	Bit 0	Name
$0040	IOS7	IOS6	IOS5	IOS4	IOS3	IOS2	IOS1	IOS0	TIOS
$0044-5	Bit 15	14	13	12	11	10		Bit 0	TCNT
$0046	TEN	TSWAI	TSFRZ	TFFCA	0	0	0	0	TSCR1
$004C	C7I	C6I	C5I	C4I	C3I	C2I	C1I	C0I	TIE
$004D	TOI	0	PUPT	RDPT	TCRE	PR2	PR1	PR0	TSCR2
$004E	C7F	C6F	C5F	C4F	C3F	C2F	C1F	C0F	TFLG1
$004F	TOF	0	0	0	0	0	0	0	TFLG2
$0050-1	Bit 15	14	13	12	11	10		Bit 0	TC0
$0052-3	Bit 15	14	13	12	11	10		Bit 0	TC1
$0054-5	Bit 15	14	13	12	11	10		Bit 0	TC2
$0056-7	Bit 15	14	13	12	11	10		Bit 0	TC3
$0058-9	Bit 15	14	13	12	11	10		Bit 0	TC4
$005A-B	Bit 15	14	13	12	11	10		Bit 0	TC5
$005C-D	Bit 15	14	13	12	11	10		Bit 0	TC6
$005E-F	Bit 15	14	13	12	11	10		Bit 0	TC7
$0082	ADPU	AFFC	ASWAI	ETRIGLE	ETRIGP	ETRIG	ASCIE	ASCIF	ATD0CTL2
$0083	0	S8C	S4C	S2C	S1C	FIFO	FRZ1	FRZ0	ATD0CTL3
$0084	SRES8	SMP1	SMP0	PRS4	PRS3	PRS2	PRS1	PRS0	ATD0CTL4
$0085	DJM	DSGN	SCAN	MULT	0	CC	CB	CA	ATD0CTL5
$0086	SCF	0	ETORF	FIFOR	0	CC2	CC1	CC0	ATD0STAT0
$008B	CCF7	CCF6	CCF5	CCF4	CCF3	CCF2	CCF1	CCF0	ATD0STAT1
$008D	Bit 7	6	5	4	3	2	1	Bit 0	ATD0DIEN
$008F	PAD07	PAD06	PAD05	PAD04	PAD03	PAD02	PAD01	PAD00	PORTAD0
$0090-1	Bit 15	14	13	12	11	10		Bit 0	ATD0DR0
$0092-3	Bit 15	14	13	12	11	10		Bit 0	ATD0DR1
$0094-5	Bit 15	14	13	12	11	10		Bit 0	ATD0DR2
$0096-7	Bit 15	14	13	12	11	10		Bit 0	ATD0DR3
$0098-9	Bit 15	14	13	12	11	10		Bit 0	ATD0DR4
$009A-B	Bit 15	14	13	12	11	10		Bit 0	ATD0DR5
$009C-D	Bit 15	14	13	12	11	10		Bit 0	ATD0DR6
$009E-F	Bit 15	14	13	12	11	10		Bit 0	ATD0DR7
$00C9	0	0	0	SBR12	SBR11	SBR10		SBR0	SCI0BD
$00CA	LOOPS	SCISWAI	RSRC	M	WAKE	ILT	PE	PT	SCI0CR1
$00CB	TIE	TCIE	RIE	ILIE	TE	RE	RWU	SBK	SCI0CR2
$00CC	TDRE	TC	RDRF	IDLE	OR	NF	FE	PF	SCI0SR1
$00CD	0	0	0	0	0	BRK13	TXDIR	RAF	SCI0SR2
$00CF	R7/T7	R6/T6	R5/T5	R4/T4	R3/T3	R2/T2	R1/T1	R0/T0	SCI0DRL
$00D0-1	0	0	0	SBR12	SBR11	SBR10		SBR0	SCI1BD
$00D2	LOOPS	SCISWAI	RSRC	M	WAKE	ILT	PE	PT	SCI1CR1
$00D3	TIE	TCIE	RIE	ILIE	TE	RE	RWU	SBK	SCI1CR2
$00D4	TDRE	TC	RDRF	IDLE	OR	NF	FE	PF	SCI1SR1
$00D5	0	0	0	0	0	BRK13	TXDIR	RAF	SCI1SR2
$00D7	R7/T7	R6/T6	R5/T5	R4/T4	R3/T3	R2/T2	R1/T1	R0/T0	SCI1DRL
$0240	PT7	PT6	PT5	PT4	PT3	PT2	PT1	PT0	PTT
$0242	DDRT7	DDRT6	DDRT5	DDRT4	DDRT3	DDRT2	DDRT1	DDRT0	DDRT
$0248	PS7	PS6	PS5	PS4	PS3	PS2	PS1	PS0	PTS
$024A	DDRS7	DDRS6	DDRS5	DDRS4	DDRS3	DDRS2	DDRS1	DDRS0	DDRS
$0250	PM7	PM6	PM5	PM4	PM3	PM2	PM1	PM0	PTM
$0252	DDRM7	DDRM6	DDRM5	DDRM4	DDRM3	DDRM2	DDRM1	DDRM0	DDRM
$0258	PP7	PP6	PP5	PP4	PP3	PP2	PP1	PP0	PTP
$025A	DDRP7	DDRP6	DDRP5	DDRP4	DDRP3	DDRP2	DDRP1	DDRP0	DDRP
$0260	PH7	PH6	PH5	PH4	PH3	PH2	PH1	PH0	PTH
$0262	DDRH7	DDRH6	DDRH5	DDRH4	DDRH3	DDRH2	DDRH1	DDRH0	DDRH
$0268	PJ7	PJ6	0	0	0	0	PJ1	PJ0	PTJ
$026A	DDRJ7	DDRJ6	0	0	0	0	DDRJ1	DDRJ0	DDRJ

`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/$2^{\mathbf{n}}$, with PLL **TCNT** is 24MHz/$2^{\mathbf{n}}$, **n** ranges from 0 to 7

				E = 8 MHz		E = 24 MHz	
			Divide	TCNT	TCNT	TCNT	TCNT
PR2	PR1	PR0	by	period	frequency	period	frequency
0	0	0	1	125 ns	8 MHz	41.7 ns	24 MHz
0	0	1	2	250 ns	4 MHz	83.3 ns	12 MHz
0	1	0	4	500 ns	2 MHz	167 ns	6 MHz
0	1	1	8	1 µs	1 MHz	333 ns	3 MHz
1	0	0	16	2 µs	500 kHz	667 ns	1.5 MHz
1	0	1	32	4 µs	250 kHz	1.33 µs	667 kHz
1	1	0	64	8 µs	125 kHz	2.67 µs	333 kHz
1	1	1	128	16 µs	62.5 kHz	5.33 µs	167 kHz

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

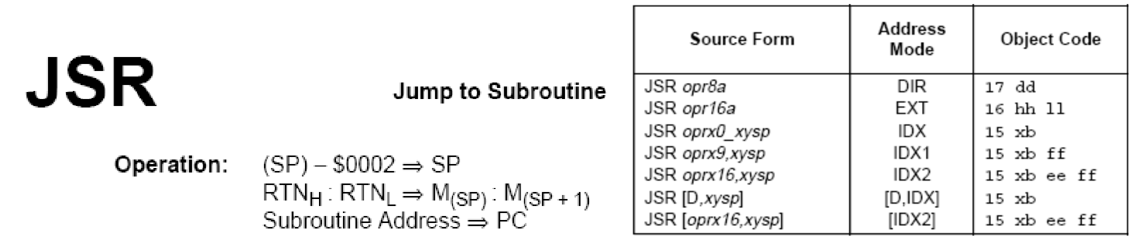
bit 7 TDRE, Transmit Data Register Empty Flag
Set if transmit data can be written to **SCI0DRL**
Cleared by **SCI0SR1** read with TDRE set followed by **SCI0DRL** write
bit 5 RDRF, Receive Data Register Full
set if a received character is ready to be read from **SCI0DRL**
Clear the RDRF flag by reading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

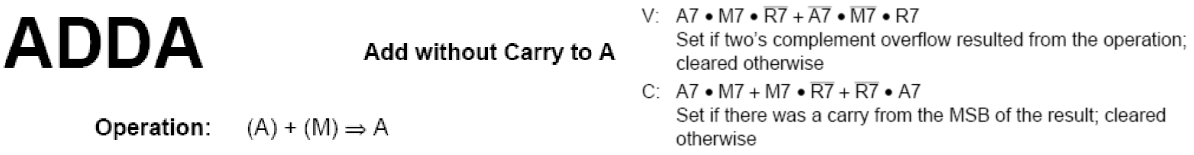
bit 7 SCF cleared on a write to **ATD0CTL5** and is set when the conversion sequence is done



JSR Jump to Subroutine

Operation: (SP) – \$0002 ⇒ SP
$RTN_H : RTN_L \Rightarrow M_{(SP)} : M_{(SP+1)}$
Subroutine Address ⇒ PC

Source Form	Address Mode	Object Code
JSR *opr8a*	DIR	17 dd
JSR *opr16a*	EXT	16 hh ll
JSR *oprx0_xysp*	IDX	15 xb
JSR *oprx9,xysp*	IDX1	15 xb ff
JSR *oprx16,xysp*	IDX2	15 xb ee ff
JSR [D,*xysp*]	[D,IDX]	15 xb
JSR [*oprx16,xysp*]	[IDX2]	15 xb ee ff



ADDA Add without Carry to A

Operation: (A) + (M) ⇒ A

V: $A7 \bullet M7 \bullet \overline{R7} + \overline{A7} \bullet \overline{M7} \bullet R7$
Set if two's complement overflow resulted from the operation; cleared otherwise

C: $A7 \bullet M7 + M7 \bullet \overline{R7} + \overline{R7} \bullet A7$
Set if there was a carry from the MSB of the result; cleared otherwise