First:________________ Last:____________________

This is a closed book exam. You must put your answers in the boxes 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."

Signature ______________________________________

(10) Question 1. State the term that is described by each definition.

Part a) You are given an ADC to test. You measure a change in voltage, ΔV, such that whenever the input changes by at least this much the ADC result becomes different. Conversely, if you change the input by less than this amount, sometimes the ADC result does not change.

Part b) A drawing with circles and rectangles. The circles are software modules and the rectangles are hardware. If a software module modifies hardware there is an arrow from the circle to the rectangle. If the software in a first module invokes a function in a second module there is an arrow from the first to second circle.

Part c) A non-divisible entity when transmitting serial data. In the UART lab this semester this entity was 10 bits wide.

Part d) A property of a system such that the time delay between a request and the service of the request is always less than a constant. This time response is acceptable to our customers.

Part e) A data structure that implements last in first out behavior. In other words, the information retrieved is always the information that was most recently entered.

Part f) A rule that states if we sample the ADC at a rate of f_s, the digital samples can represent the frequencies from 0 to ½ f_s.

Part g) A type of software comprised of these building blocks: sequence, if-then, and while-loop.

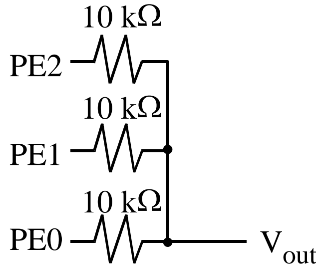
Part h) A debugging process run by determining in advance, either by analytical algorithm or explicit calculations, the expected outputs of strategic intermediate stages and final results for typical inputs. We then run our program and compare the actual outputs with this template of expected results.

Part i) The name given to describe 1,048,576 bits.

Part j) The specification defining where software will start when power is first applied to the system.

(5) Question 2. Consider what happens when a program calls a function. This invocation creates a stack frame. List three objects that may be present in the stack frame.

(5) Question 3. What is the output voltage V_{out} when PE2 is high, PE1 is high, and PE0 is low? Assume V_{OH} is 3.3V and V_{OL} = 0V.



(5) Question 4. Assume **`Height`** is the integer part of an 8-bit signed fixed-point variable with a resolution of 0.1 cm. The goal is to subtract 0.5 cm from the value of the variable. Will the following software always operate properly?

```
LDR R0,=Height
LDRSB R1,[R0]   ;read and promote to 32 bits
SUB R1,R1,#5    ;perform the subtraction in 32-bit mode
STRB [R0]       ;demote back to 8 bits, store into variable
```

A) Yes, the program has no errors.
B) No, an error occurs if the V bit is set by the **`SUB`** instruction.
C) No, error can occur during the demotion.
D) No, an error occurs if the C bit is set by the **`SUB`** instruction.
E) No, one needs to divide by 10 to get the correct result.
F) No, the **`SUB`** instruction should have been **`SUB R1,R1,#0.5`**
G) No, dropout can occur.

(5) Question 5. Design the circuit that interfaces an LED to Port E bit 0. Assume the LED voltage drop is 2 V and the desired LED current is 2 mA. When the software outputs a high, the voltage on PE0 becomes 3.2 V (LED on). When the software outputs a low, the voltage on PE0 becomes 0.2 V (LED off). Show the interface (a 7406 driver will not be needed because the current is low)

(15) Question 6. There are arrays of 16-bit signed numbers. The arrays are variable length with -32768 termination. The entry -32768 is not a data point in the array. For example, here are three such possible arrays. If -32768 is the first entry, the array is empty.

```
short buf1[5]={4,1000,-1000,0,-32768};
short buf2[7]={6,-4,100,200,2,0,-32768};
short buf3[1]={-32768};
```

Part a) Write a C function that takes a pointer to an array and returns the sum of all the data points. For example

```
Result1 = Sum(buf1);  // should return 4 = 4+1000-1000
Result2 = Sum(buf2);  // should return 304 = 6-4+100+200+2+0
Result3 = Sum(buf3);  // should return 0 because array is empty
```

You are not allowed to add any global variables. For the C implementation, do not worry about overflow when calculating the sum.

Part b) Write an assembly subroutine that performs the same operation. The pointer to the array is passed in Register R0, and the 16-bit result is returned in Register R0. You are not allowed to add any global variables. Different from the C implementation, return Register R0 equal to -32768 if overflow occurs during the calculations.

(10) Question 7. 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

```
     AREA    |.text|, CODE, READONLY, ALIGN=2
Data DCD  100          ;32-bit information
Main LDR  R0,=Data     ;pointer to Data
     PUSH {R0}         ;pointer to the Data is pushed
     MOV  R0,#0        ;no cheating, parameter not in R0, on stack
     BL   Subroutine
     ADD  SP,SP,#4     ;discard parameter
```

The subroutine allocates one 32-bit local variable, **L1**, and uses SP stack pointer addressing to access the local variable and parameter. The binding for these two are

```
Pt  EQU  ???  ;32-bit pointer to 32-bit data
L1  EQU  ???  ;32-bit local variable
Subroutine
    SUB  SP,SP,#4  ;allocate L1
    PUSH {R10,R11}
;---------start of body-------------------
    ????????       ;Reg R10 points to data
    LDR  R11,[R10] ;R11= value of the data
    STR  [SP,#L1]  ;save parameter into local L1
;---------end of body---------------------
    POP  {R10,R11}
    ADD  SP,SP,#4  ;deallocate L1
    BX   LR
```

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

Pt EQU [ ]

L1 EQU [ ]

Part b) Show the instruction(s) for the **????????** in the above program. In particular, you must use SP stack frame addressing, **Pt** binding, and bring the value of the parameter into Register R10. It should be done in one instruction. It has to work in general, but for this example calling sequence, the instruction **LDR R11,[R10]** will load the value of 100 into Register R11.

(10) Question 8. Write C function that samples the ADC. The channel number 0 to 7 is passed into the function using call by value. The function returns the 10-bit sample value in right-justified format. Your function should start the ADC, wait for the ADC to finish using busy-wait synchronization, then read one 10-bit conversion from the ADC. You may assume the ADC interface is already initialized to sample one channel in 10-bit mode. The result should vary from 0 to 1023.

Part a) Show the C code you would place in the header file (ADC.h). Comments will be graded.

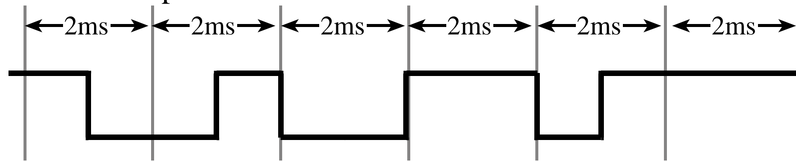
Part b) Show the C code you would place in the code file (ADC.c). Comments are not required for this part.

(5) Question 9. Assume the bus clock is operating at 50 MHz. The SysTick initialization executes these instructions.

```
SysTick_Init
    LDR R1, =NVIC_ST_RELOAD_R        ; R1 = &NVIC_ST_RELOAD_R
    ?????????
    STR R0, [R1]
    LDR R1, =NVIC_ST_CTRL_R          ; R1 = &NVIC_ST_CTRL_R
    ?????????
    STR R2, [R1]
    BX  LR                           ; return
```

What value goes in the two `?????` placea to make the interrupt frequency 100 Hz?

(5) Question 10. You observe the following waveform at the output of a UART port. You know the format is 1 start, 8 data and 1 stop bit.



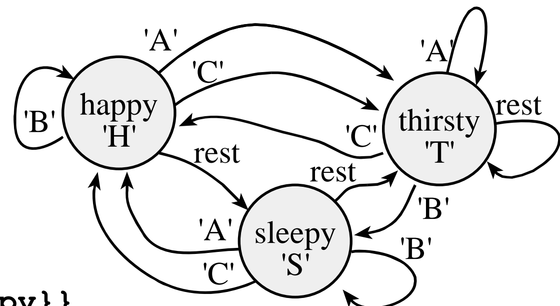
Part a) What is the baud rate?

Part b) What 8-bit number is being transmitted? Give your answer in hexadecimal.

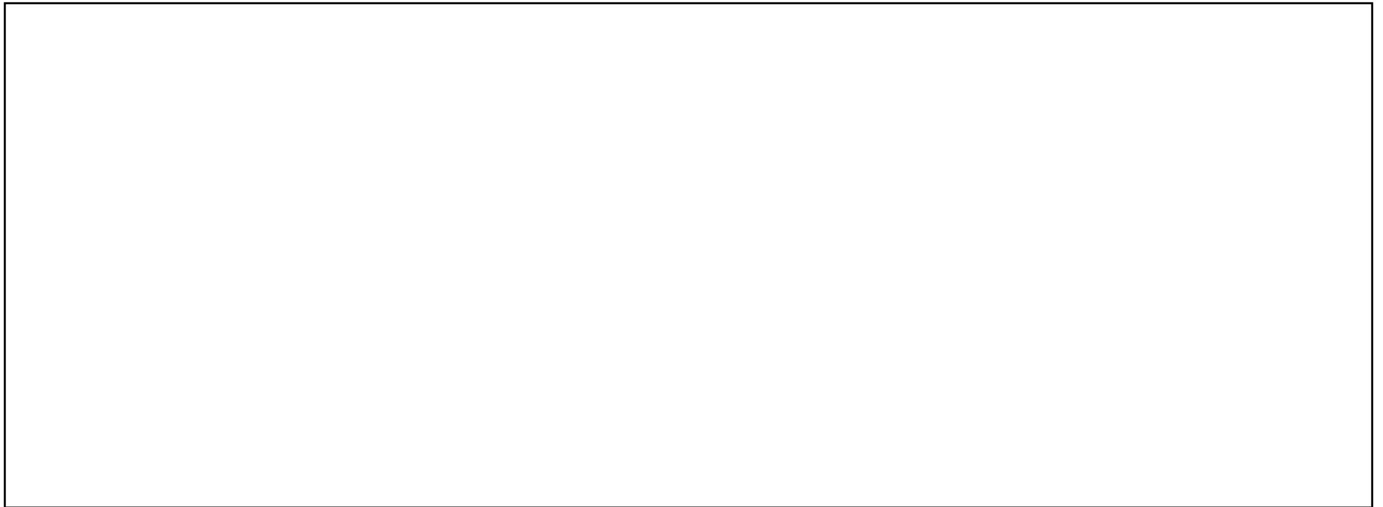
(25) Question 11. In this problem, your software will implement a Moore FSM using the UART0 serial port. You must use UART0 RTRIS input interrupts, but not output interrupts. The baud rate is 10000 bits/sec. You may assume the bus clock is 50 MHz. The input arrives as ASCII characters from the UART receiver. The output leaves as ASCII characters out of the UART transmitter. The arrow labeled "rest" should be taken for any input not 'A' 'B' or 'C'. The code for the FSM structure is given

```
const struct State{
  unsigned char Out;             // Output via UART
  const struct State *Next[4];  // if input ='A', 'B', 'C' or other
};
typedef const struct State StateType;
typedef StateType * StatePtr;

#define happy   &fsm[0]
#define thirsty &fsm[1]
#define sleepy  &fsm[2]
StateType fsm[3]={
     {'H', {thirsty,happy,thirsty,sleepy}},
     {'T', {thirsty,sleepy,happy,thirsty}},
     {'S', {happy,sleepy,happy,thirsty}}
};
StatePtr Pt;  // Current State
```



Part a) Write the initialization function in C that sets up the UART0. The main program will call this initialization once at the beginning, and then perform unrelated tasks. This function should arm and enable interrupts. Initialize the current state pointer **Pt** to the happy state. No loops are allowed.



Part c) Write the ISR in C that runs the FSM using UART0. There will be exactly one input and one output for each invocation of the ISR. No loops are allowed. For each RTRIS interrupt:

Read the input frame (clear RTRIS)
Go to the next state (depending on the current state and the input)
Perform the output of that new state
Return from interrupt

Memory access instructions
```
   LDR    Rd, [Rn]         ; load 32-bit number at [Rn] to Rd
   LDR    Rd, [Rn,#off]    ; load 32-bit number at [Rn+off] to Rd
   LDR    Rd, =value       ; set Rd equal to any 32-bit value (PC rel)
   LDRH   Rd, [Rn]         ; load unsigned 16-bit at [Rn] to Rd
   LDRH   Rd, [Rn,#off]    ; load unsigned 16-bit at [Rn+off] to Rd
   LDRSH  Rd, [Rn]         ; load signed 16-bit at [Rn] to Rd
   LDRSH  Rd, [Rn,#off]    ; load signed 16-bit at [Rn+off] to Rd
   LDRB   Rd, [Rn]         ; load unsigned 8-bit at [Rn] to Rd
   LDRB   Rd, [Rn,#off]    ; load unsigned 8-bit at [Rn+off] to Rd
   LDRSB  Rd, [Rn]         ; load signed 8-bit at [Rn] to Rd
   LDRSB  Rd, [Rn,#off]    ; load signed 8-bit at [Rn+off] to Rd
   STR    Rt, [Rn]         ; store 32-bit Rt to [Rn]
   STR    Rt, [Rn,#off]    ; store 32-bit Rt to [Rn+off]
   STRH   Rt, [Rn]         ; store least sig. 16-bit Rt to [Rn]
   STRH   Rt, [Rn,#off]    ; store least sig. 16-bit Rt to [Rn+off]
   STRB   Rt, [Rn]         ; store least sig. 8-bit Rt to [Rn]
   STRB   Rt, [Rn,#off]    ; store least sig. 8-bit Rt to [Rn+off]
   PUSH   {Rt}             ; push 32-bit Rt onto stack
   POP    {Rd}             ; pop 32-bit number from stack into Rd
   ADR    Rd, label        ; set Rd equal to the address at label
   MOV{S} Rd, <op2>        ; set Rd equal to op2
   MOV    Rd, #im16        ; set Rd equal to im16, im16 is 0 to 65535
   MVN{S} Rd, <op2>        ; set Rd equal to -op2
```
Branch instructions
```
   B    label   ; branch to label    Always
   BEQ  label   ; branch if Z == 1   Equal
   BNE  label   ; branch if Z == 0   Not equal
   BCS  label   ; branch if C == 1   Higher or same, unsigned ≥
   BHS  label   ; branch if C == 1   Higher or same, unsigned ≥
   BCC  label   ; branch if C == 0   Lower, unsigned <
   BLO  label   ; branch if C == 0   Lower, unsigned <
   BMI  label   ; branch if N == 1   Negative
   BPL  label   ; branch if N == 0   Positive or zero
   BVS  label   ; branch if V == 1   Overflow
   BVC  label   ; branch if V == 0   No overflow
   BHI  label   ; branch if C==1 and Z==0  Higher, unsigned >
   BLS  label   ; branch if C==0 or  Z==1  Lower or same, unsigned ≤
   BGE  label   ; branch if N == V   Greater than or equal, signed ≥
   BLT  label   ; branch if N != V   Less than, signed <
   BGT  label   ; branch if Z==0 and N==V  Greater than, signed >
   BLE  label   ; branch if Z==1 and N!=V  Less than or equal, signed ≤
   BX   Rm      ; branch indirect to location specified by Rm
   BL   label   ; branch to subroutine at label
   BLX  Rm      ; branch to subroutine indirect specified by Rm
```
Interrupt instructions
```
   CPSIE  I                 ; enable interrupts  (I=0)
   CPSID  I                 ; disable interrupts (I=1)
```

Logical instructions
```
   AND{S} {Rd,} Rn, <op2> ; Rd=Rn&op2    (op2 is 32 bits)
   ORR{S} {Rd,} Rn, <op2> ; Rd=Rn|op2    (op2 is 32 bits)
   EOR{S} {Rd,} Rn, <op2> ; Rd=Rn^op2    (op2 is 32 bits)
   BIC{S} {Rd,} Rn, <op2> ; Rd=Rn&(~op2) (op2 is 32 bits)
   ORN{S} {Rd,} Rn, <op2> ; Rd=Rn|(~op2) (op2 is 32 bits)
```

```
LSR{S} Rd, Rm, Rs      ; logical shift right Rd=Rm>>Rs  (unsigned)
LSR{S} Rd, Rm, #n      ; logical shift right Rd=Rm>>n   (unsigned)
ASR{S} Rd, Rm, Rs      ; arithmetic shift right Rd=Rm>>Rs (signed)
ASR{S} Rd, Rm, #n      ; arithmetic shift right Rd=Rm>>n  (signed)
LSL{S} Rd, Rm, Rs      ; shift left Rd=Rm<<Rs (signed, unsigned)
LSL{S} Rd, Rm, #n      ; shift left Rd=Rm<<n  (signed, unsigned)
```

Arithmetic instructions

```
ADD{S} {Rd,} Rn, <op2> ; Rd = Rn + op2
ADD{S} {Rd,} Rn, #im12 ; Rd = Rn + im12, im12 is 0 to 4095
SUB{S} {Rd,} Rn, <op2> ; Rd = Rn - op2
SUB{S} {Rd,} Rn, #im12 ; Rd = Rn - im12, im12 is 0 to 4095
RSB{S} {Rd,} Rn, <op2> ; Rd = op2 - Rn
RSB{S} {Rd,} Rn, #im12 ; Rd = im12 – Rn
CMP    Rn, <op2>       ; Rn – op2      sets the NZVC bits
CMN    Rn, <op2>       ; Rn - (-op2)   sets the NZVC bits
MUL{S} {Rd,} Rn, Rm    ; Rd = Rn * Rm       signed or unsigned
MLA    Rd, Rn, Rm, Ra  ; Rd = Ra + Rn*Rm    signed or unsigned
MLS    Rd, Rn, Rm, Ra  ; Rd = Ra - Rn*Rm    signed or unsigned
UDIV   {Rd,} Rn, Rm    ; Rd = Rn/Rm         unsigned
SDIV   {Rd,} Rn, Rm    ; Rd = Rn/Rm         signed
```

Notes

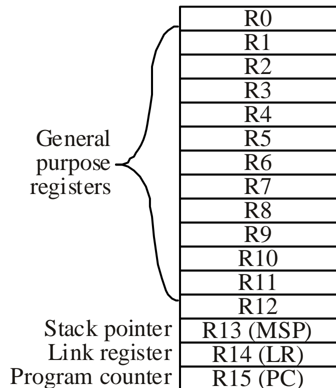
```
Ra Rd Rm Rn Rt represent 32-bit registers
value   any 32-bit value: signed, unsigned, or address
{S}     if S is present, instruction will set condition codes
#im12   any value from 0 to 4095
#im16   any value from 0 to 65535
{Rd,}   if Rd is present Rd is destination, otherwise Rn
#n      any value from 0 to 31
#off    any value from -255 to 4095
label   any address within the ROM of the microcontroller
op2     the value generated by <op2>
```

Examples of flexible operand **<op2>** creating the 32-bit number. E.g., **Rd = Rn+op2**

```
ADD Rd, Rn, Rm         ; op2 = Rm
ADD Rd, Rn, Rm, LSL #n ; op2 = Rm<<n  Rm is signed, unsigned
ADD Rd, Rn, Rm, LSR #n ; op2 = Rm>>n  Rm is unsigned
ADD Rd, Rn, Rm, ASR #n ; op2 = Rm>>n  Rm is signed
ADD Rd, Rn, #constant  ; op2 = constant
```

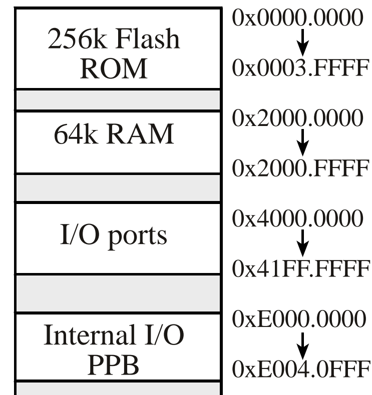
, where **X** and **Y** are hexadecimal digits:

- produced by shifting an 8-bit unsigned value left by any number of bits
- in the form **0x00XY00XY**
- in the form **0xXY00XY00**
- in the form **0xXYXYXYXY**



Condition code bits

N negative
Z zero
V signed overflow
C carry or unsigned overflow



Address	7	6	5	4	3	2	1	0	Name
$400F.E108	GPIOH	GPIOG	GPIOF	GPIOE	GPIOD	GPIOC	GPIOB	GPIOA	SYSCTL_RCGC2_R
$4000.43FC	DATA	DATA	DATA	DATA	DATA	DATA	DATA	DATA	GPIO_PORTA_DATA_R
$4000.4400	DIR	DIR	DIR	DIR	DIR	DIR	DIR	DIR	GPIO_PORTA_DIR_R
$4000.4420	SEL	SEL	SEL	SEL	SEL	SEL	SEL	SEL	GPIO_PORTA_AFSEL_R
$4000.451C	DEN	DEN	DEN	DEN	DEN	DEN	DEN	DEN	GPIO_PORTA_DEN_R

Table 4.5. Some LM3S1968 parallel ports. Each register is 32 bits wide. Bits 31 – 8 are zero.

We set the direction register (e.g., **`GPIO_PORTA_DIR_R`**) to specify which pins are input (0) and which are output (1). We will set bits in the alternative function register when we wish to activate the alternate functions (not GPIO). We use the data register (e.g., **`GPIO_PORTA_DATA_R`**) to perform input/output on the port. For each I/O pin we wish to use whether GPIO or alternate function we must enable the digital circuits by setting the bit in the enable register (e.g., **`GPIO_PORTA_DEN_R`**).

Address	31	30	29-7	6	5	4	3	2	1	0	Name
0xE000E100	G	F	…	UART1	UART0	E	D	C	B	A	NVIC_EN0_R
0xE000E104			…						UART2	H	NVIC_EN1_R

Address	31-24	23-17	16	15-3	2	1	0	Name
$E000E010	0	0	COUNT	0	CLK_SRC	INTEN	ENABLE	NVIC_ST_CTRL_R
$E000E014	0	24-bit RELOAD value						NVIC_ST_RELOAD_R
$E000E018	0	24-bit CURRENT value of SysTick counter						NVIC_ST_CURRENT_R

Address	31-29	28-24	23-21	20-8	7-5	4-0	Name
$E000ED20	TICK	0	PENDSV	0	DEBUG	0	NVIC_SYS_PRI3_R

Table 9.6. SysTick registers.

Table 9.6 shows the SysTick registers used to create a periodic interrupt. SysTick has a 24-bit counter that decrements at the bus clock frequency. Let f_{BUS} be the frequency of the bus clock, and let n be the value of the **RELOAD** register. The frequency of the periodic interrupt will be $f_{BUS}/(n+1)$. First, we clear the **ENABLE** bit to turn off SysTick during initialization. Second, we set the **RELOAD** register. Third, we write to the **`NVIC_ST_CURRENT_R`** value to clear the counter. Lastly, we write the desired mode to the control register, **`NVIC_ST_CTRL_R`**. To turn on the SysTick, we set the **ENABLE** bit. We must set **CLK_SRC**=1, because **CLK_SRC**=0 external clock mode is not implemented on the LM3S/LM4F family. We set **INTEN** to enable interrupts. The standard name for the SysTick ISR is **SysTick_Handler**.

Address	31-17	16	15-10	9	8	7-0	Name
$400F.E000		ADC		MAXADCSPD			SYSCTL_RCGC0_R

	31-14	13-12	11-10	9-8	7-6	5-4	3-2	1-0	
$4003.8020		SS3		SS2		SS1		SS0	ADC_SSPRI_R

	31-16	15-12	11-8	7-4	3-0	
$4003.8014		EM3	EM2	EM1	EM0	ADC_EMUX_R

	31-4	3	2	1	0	
$4003.8000		ASEN3	ASEN2	ASEN1	ASEN0	ADC_ACTSS_R
$4003.80A0				MUX0		ADC_SSMUX3_R
$4003.80A4		TS0	IE0	END0	D0	ADC_SSCTL3_R
$4003.8028		SS3	SS2	SS1	SS0	ADC_PSSI_R
$4003.8004		INR3	INR2	INR1	INR0	ADC_RIS_R
$4003.8008		MASK3	MASK2	MASK1	MASK0	ADC_IM_R
$4003.800C		IN3	IN2	IN1	IN0	ADC_ISC_R

	31-10	9-0	
$4003.80A8		DATA	ADC_SSFIFO3

Table 10.3. The LM3S ADC registers. Each register is 32 bits wide.

Set MAXADCSPD to 00 for slow speed operation. The ADC has four sequencers, but we will use only sequencer 3. We set the **ADC_SSPRI_R** register to 0x3210 to make sequencer 3 the lowest priority. Because we are using just one sequencer, we just need to make sure each sequencer has a unique priority. We set bits 15–12 (**EM3**) in the

ADC_EMUX_R register to specify how the ADC will be triggered. If we specify software start (**EM3**=0x0), then the software writes an 8 (**SS3**) to the **ADC_PSSI_R** to initiate a conversion on sequencer 3. Bit 3 (**INR3**) in the **ADC_RIS_R** register will be set when the conversion is complete. We can enable and disable the sequencers using the **ADC_ACTSS_R** register. There are eight on the LM3S1968. Which channel we sample is configured by writing to the **ADC_SSMUX3_R** register. The **ADC_SSCTL3_R** register specifies the mode of the ADC sample. Clear **TS0**. We set **IE0** so that the **INR3** bit is set on ADC conversion, and clear it when no flags are needed. We will set **IE0** for both interrupt and busy-wait synchronization. When using sequencer 3, there is only one sample, so **END0** will always be set, signifying this sample is the end of the sequence. Clear the **D0** bit. The **ADC_RIS_R** register has flags that are set when the conversion is complete, assuming the **IE0** bit is set. Do not set bits in the **ADC_IM_R** register because we do not want interrupts.

UART0 pins are on PA1 (transmit) and PA0 (receive).

The **UART0_IBRD_R** and **UART0_FBRD_R** registers specify the baud rate. The baud rate **divider** is a 22-bit binary fixed-point value with a resolution of 2^{-6}. The **Baud16** clock is created from the system bus clock, with a frequency of (Bus clock frequency)/**divider**. The baud rate is 16 times slower than **Baud16**

Baud rate = **Baud16**/**16** = (Bus clock frequency)/(16***divider**)

We set bit 4 of the **UART0_LCRH_R** to enable the hardware FIFOs. We set both bits 5 and 6 of the **UART0_LCRH_R** to establish an 8-bit data frame. The **RTRIS** is set on a receiver timeout, which is when the receiver FIFO is not empty and no incoming frames have occurred in a 32-bit time period. The arm bits are in the **UART0_IM_R** register. To acknowledge an interrupt (make the trigger flag become zero), software writes a 1 to the corresponding bit in the **UART0_IC_R** register. We set bit 0 of the **UART0_CTL_R** to enable the UART.

Writing to **UART0_DR_R** register will output on the UART. This data is placed in a 16-deep transmit hardware FIFO. Data are transmitted first come first serve. Received data are place in a 16-deep receive hardware FIFO. Reading from **UART0_DR_R** register will get one data from the receive hardware FIFO. The status of the two FIFOs can be seen in the **UART0_FR_R** register (FF is FIFO full, FE is FIFO empty). The standard name for the UART0 ISR is **UART0_Handler**.

Address	Bits									Name
	31–12	11	10	9	8	7–0				Name
$4000.C000		OE	BE	PE	FE	DATA				UART0_DR_R
	31–3				3	2	1	0		
$4000.C004					OE	BE	PE	FE		UART0_RSR_R
	31–8	7	6	5	4	3	2–0			
$4000.C018		TXFE	RXFF	TXFF	RXFE	BUSY				UART0_FR_R
	31–16	15–0								
$4000.C024		DIVINT								UART0_IBRD_R
	31–6				5–0					
$4000.C028					DIVFRAC					UART0_FBRD_R
	31–8	7	6 – 5	4	3	2	1	0		
$4000.C02C		SPS	WPEN	FEN	STP2	EPS	PEN	BRK		UART0_LCRH_R
	31–10	9	8	7	6–3	2	1	0		
$4000.C030		RXE	TXE	LBE		SIRLP	SIREN	UARTEN		UART0_CTL_R
	31–6				5-3	2-0				
$4000.C034					RXIFLSEL	TXIFLSEL				UART0_IFLS_R
	31-11	10	9	8	7	6	5	4		
$4000.C038		OEIM	BEIM	PEIM	FEIM	RTIM	TXIM	RXIM		UART0_IM_R
$4000.C03C		OERIS	BERIS	PERIS	FERIS	RTRIS	TXRIS	RXRIS		UART0_RIS_R
$4000.C040		OEMIS	BEMIS	PEMIS	FEMIS	RTMIS	TXMIS	RXMIS		UART0_MIS_R
$4000.C044		OEIC	BEIC	PEIC	FEIC	RTIC	TXIC	RXIC		UART0_IC_R

Table 11.2. UART0 registers. Each register is 32 bits wide. Shaded bits are zero.