Final Exam

Date: December 14, 2019

Printed Name: ______________________________ ______________________________
Last, First

Your signature is your promise that you have not cheated and will not cheat on this exam, nor will you help others to cheat on this exam. *<u>You will not reveal the contents of this exam to others who are taking the makeup thereby giving them an undue advantage</u>*:

Signature: ____

Instructions:

- ***Write your UT EID on all pages (at the top) and circle your instructor's name at the bottom.***
- Closed book and closed notes. No books, no papers, no data sheets (other than the last four pages of this Exam)
- No devices other than pencil, pen, eraser (no calculators, no electronic devices), please turn cell phones off.
- Please be sure that your answers to all questions (and all supporting work that is required) are contained in the space (boxes) provided. *Anything outside the boxes will be ignored in grading*.
- You have 180 minutes, so allocate your time accordingly.
- For all questions, unless otherwise stated, find the most efficient (time, resources) solution.
- Unless otherwise stated, make all I/O accesses friendly.
- *Please read the entire exam before starting.* ***See supplement pages for Device I/O registers.***

(5) Problem 1. Variables. Consider the following C program.

```
uint8_t A=5;
const uint8_t B=5;
static uint8_t C=5;
volatile uint8_t D=5;
void func(const int32_t E, int32_t F){
  int32_t G=5;
  int32_t static H=5;
}
```

For each question list all possible variable names. Specify names **A B C D E F G** and/or **H**. If there are no possible answers, specific NONE.

(1) Part a) Which variable is allocated in R1? (for this question give the one answer) …………………………

(1) Part b) Which variables may be allocated on the stack? (for this question give NONE, one, or more answers) ……………………………………………………………………………………………..……

(1) Part c) Which variables are private to (have scope limited to) the function `func`? (for this question give NONE, one, or more answers) ………………………………………..……………

(1) Part d) Which variables are initialized to 5 when the you **download** object code to the TM4C123, before any software has started? (for this question give NONE, one, or more answers) ………………………

(1) Part e) Which variable is the best one to use to share information between the main program and software running in an ISR? (for this question give the one answer) ……………………………………….

(15) Problem 2. Equations. Give the relationships in terms of these parameters: (V_{OL}, output low voltage of TM4C123 in volts), (V, voltage in volts), (R, resistance in ohms), (n, number of bits in the ADC, e.g., 12 bits), (b, baud rate of the UART in bits/sec, e.g., 115200 bps), (*max*, the maximum possible ADC voltage in volts, e.g., 3.3V), (*min*, the minimum possible ADC voltage in volts, e.g., 0V), (r, rate at which one moves the slide pot in oscillations per sec, e.g., 10 Hz), (R, the SysTick RELOAD value), (f, the TM4C123 bus frequency in Hz, e.g., 80,000,000 Hz).

(4) Part a) Give the relationship for the **power** dissipated in a resistor.

P =

Units of power =

(4) Part b) Give the relationship for the maximum **bandwidth** possible on a UART.

BW =

Units of bandwidth =

(4) Part c) Give the relationship for the ADC **resolution**.

Resolution =

Units of resolution =

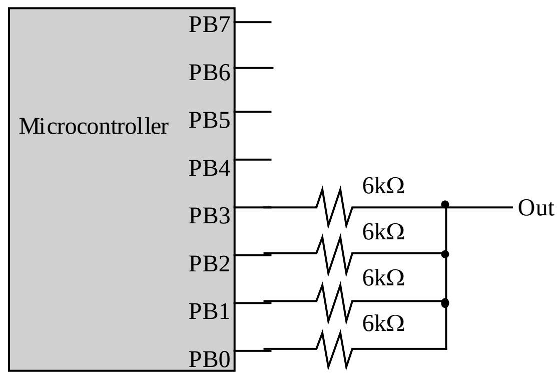
(3) Part d) Give the relationship for SysTick **interrupt period**.

Period =

Units of period =

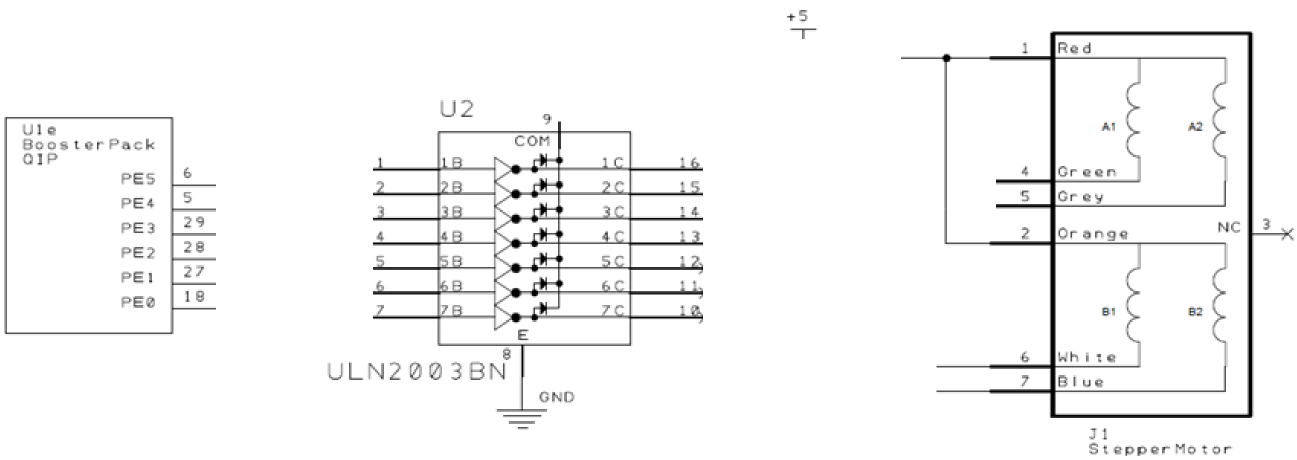
Integer (binary)	Out (volts)
0000	0.0V
0001	
0010	
0011	
0100	
0101	
0110	
0111	
1000	
1001	
1010	
1011	
1100	
1101	
1110	
1111	4.0 V

(10) Problem 3. Circuit. Consider this interface circuit. Assume PB3, PB2, PB1, PB0 are digital output representing a binary integer from 0 to 15. Notice all the resistors are the same value. To make the math easier, assume V_{OH} of the microcontroller is 4V, and assume V_{OL} is 0V. Some of the values are filled in. Complete the table showing the relationship between output voltage Out, and the binary integer. Show your work



(5) Question 4. Write an assembly language subroutine that stores a value into an array. The array is a global called `Buffer`. Each element of the array is a signed 16-bit integer. The size of the array is 256 elements. There are two parameters to your subroutine. R0 contains the signed 16-bit integer, and R1 contains the index (0 to 255). If R1 is less than or equal to 255, then store the one value into the array at that index. If R1 is greater than 255, do not store into the array.

(5) Problem 5. Stepper motor interface. The stepper motor we had in Lab 5 had five coils and the software output the pattern 1,2,4,8,16 to spin the motor. In this problem you will interface a stepper motor with four coils (labelled A1 A2 B1 and B2) to Port E, and the software output the pattern 5,6,10,9 to spin the motor. The desired operating point of the one coil is anywhere from 4 to 5 volts with a current of 80 to 100 mA. Assume the ULN2003B has an output low voltage of 0.5V. The maximum current of one output of the ULN2003B is 500 mA. Show the circuit and label all resistors, capacitors and interface chips needed. Just show the circuit, not software is required.



(10) Problem 6. Draw the state transition graph for a Moore FSM used to control an LED. There are two inputs and one output. Consider the two inputs as a binary integer, *I*, from 0 to 3. The input will determine the brightness of the LED. More specifically, the duty cycle of the LED should be 100**I*/3 in percent. The time constant of the human's visual processing is about 100 ms. The switch input and LED output are both in positive logic. Each state has a name, an output, a dwell time, and multiple arrows to next states. Just show the graph, no software is required. You may use X to specify 0,1,2 or 3.

(10) Question 7: You are asked to implement a FIFO queue using the following variables. These variable names and types are fixed and cannot be changed. You cannot add additional global or static variables. You can add local variables.

```
int16_t *GetPt;     // pointer to oldest (next to Get)
int16_t *PutPt;     // pointer to free space (next place to Put)
int16_t Buffer[10]; // FIFO can store up to 9 elements in this Buffer
void Fifo_Init(void){
  GetPt = PutPt = Buffer;
}
```

```
// Gets an element from the FIFO
// Input: Pointer to a place that will get
// Output: 1 for success and 0 for failure
//  failure is when the FIFO is empty
uint32_t Fifo_Get(int16_t *pt){
```

```
// Adds an element to the FIFO
// Input: value to be inserted
// Output: 1 for success and 0 for failure
//  failure is when the FIFO is full
uint32_t Fifo_Put(int16_t data){
```

(5) Problem 8. Assume the UART0 has been initialized for busy-wait synchronization. Design **an assembly function** to implement OutChar with these two steps

1) Wait for UART to be ready to accept another data for transmission
2) Write data to the UART that causes the data to be transmitted

The C prototype for the function is **`void OutChar(char data);`**

(5) Question 9. Translate the following C code to assembly

```
void (*Task)(void);


void SysTick_Init(void(*t)(void)){
  Task = t;
  NVIC_ST_RELOAD_R = 79999
  NVIC_ST_CTRL_R = 7;
  EnableInterrupts(); // I=0
}


void SysTick_Handler(void){
  (*Task)();
}
```

```
      THUMB
      AREA   DATA, ALIGN=2



      AREA   |.text|, CODE, READONLY, ALIGN=2
```

(10) Question 10. The subroutine **mySub** has one call by value parameter. There are no return parameters. The one call by value input parameter is AAPCS compliant. A typical calling sequence is

```
      AREA    |.text|, CODE, READONLY, ALIGN=2
stuff DCD  123          ;32-bit constant
start LDR  R0,=stuff
      LDR  R0,[R0]
      BL   mySub
```

The subroutine allocates two 32-bit local variables, **i** and **j** and uses SP stack pointer addressing to access the local variables. The binding for these two are

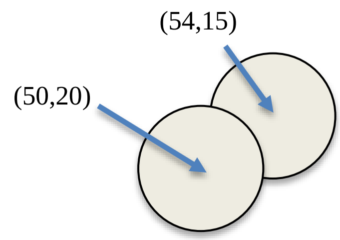
```
i  EQU  [          ] ;binding for 32-bit local variable
j  EQU  [          ] ;binding for 32-bit local variable
mySub
     [                ] ;allocate i,j
     PUSH {R4,LR}
;---------start of body------------------
     [                ] ;set i = input parameter
    LDR  R4,[SP,#i]  ;Reg R4 is the input parameter value
    STR  R4,[SP,#j]  ;save parameter into local j
;---------end of body--------------------
    POP  {R4,LR}
     [                ] ;deallocate i,j
    BX LR
```

In the boxes provided, show the binding for the two local variables, the assembly code to allocate the two local variables, the assembly code to set i equal to the input parameter, and the assembly code to deallocate the two local variables.

(5) Question 11: You are attempting to capture a sinusoidal sound with a frequency of 1 kHz. The **ADC0_PC_R** is set to 0001, which supports a maximum of 125k samples/sec. Using the 12-bit ADC and periodic interrupt, you have programmed the SysTick to interrupt at a frequency of 12 kHz. During the SysTick ISR you collect one ADC sample. Is it possible to recreate the original signal from the captured samples? If your answer is *yes*, explain how. If your answer is *no*, what is the term used to refer to this loss of information?

(15) Problem 12. Consider a game that has 32 circles. There is an array of sprites (**`Balls`**) specifying the current status of each circle. Each circle has a radius of 4 pixels, and has an (x,y) coordinate of the center of the circle, two velocities, and a life parameter. The circles are moving according to the two velocities. You may assume the **`Balls`** array has been populated with data before your function is called. Two circles are touching if the distance from one center to the other center is less than or equal to 8 pixels. The figure on the right shows one example with two circles at (x,y)=(50,20) and (54,15). These circles are touching because sqrt(4*4+5*5) = sqrt(41) is less than or equal to 8 pixels. Hint: you do not need floating point or square root to solve this problem.

```
typedef enum {dead,alive} status_t;
struct sprite {
  int16_t x;        // x coordinate, in pixels
  int16_t y;        // y coordinate, in pixels
  int16_t vx;       // x velocity, in pixels/frame
  int16_t vy;       // y velocity, in pixels/frame
  status_t life;};  // dead or alive
typedef sprite sprite_t;
sprite_t Balls[32]
```



Implement a C function that searches to see if two alive circles are touching. If two **`alive`** circles are touching, invert the sign of the x velocities of both circles. Do not worry about 3 or more circles touching at the same time.

```
void Collisions(void){
```

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 or 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

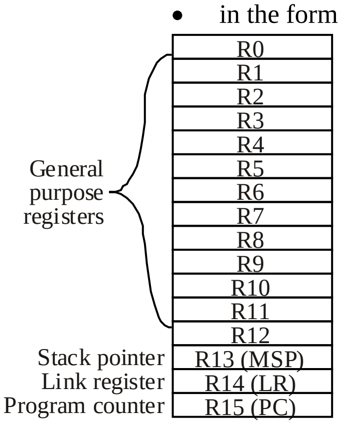
```
value   any 32-bit value: signed, unsigned, or address
{S}     if S is present, instruction will set condition codes
#im12   any value from 0 to 4095
#im16   any value from 0 to 65535
{Rd,}   if Rd is present Rd is destination, otherwise Rn
#n      any value from 0 to 31
#off    any value from -255 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
```

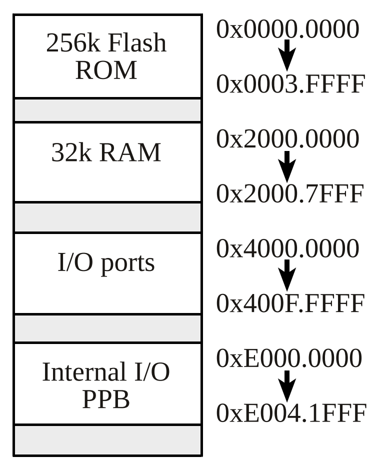
, where **X** and **Y** are hexadecimal digits:

- 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




```
DCB   1,2,3 ; allocates three 8-bit byte(s)
DCW   1,2,3 ; allocates three 16-bit halfwords
DCD   1,2,3 ; allocates three 32-bit words
SPACE 4     ; reserves 4 bytes
```

Address	7	6	5	4	3	2	1	0	Name
$400F.E608			GPIOF	GPIOE	GPIOD	GPIOC	GPIOB	GPIOA	SYSCTL_RCGCGPIO_R
$4000.53FC	DATA	DATA	DATA	DATA	DATA	DATA	DATA	DATA	GPIO_PORTB_DATA_R
$4000.5400	DIR	DIR	DIR	DIR	DIR	DIR	DIR	DIR	GPIO_PORTB_DIR_R
$4000.5420	SEL	SEL	SEL	SEL	SEL	SEL	SEL	SEL	GPIO_PORTB_AFSEL_R
$4000.551C	DEN	DEN	DEN	DEN	DEN	DEN	DEN	DEN	GPIO_PORTB_DEN_R

Table 4.5. TM4C123 Port B parallel ports. Each register is 32 bits wide. Bits 31 – 8 are zero.

Address	31	30	29-7	6	5	4	3	2	1	0	Name
0xE000E100		F	…	UART1	UART0	E	D	C	B	A	NVIC_EN0_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	SYSTICK	0	PENDSV	0	DEBUG	0	NVIC_SYS_PRI3_R

Table 9.6. SysTick registers. Note: 2^24=16,777,216

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. We set **INTEN** to arm SysTick interrupts. The standard name for the SysTick ISR is **SysTick_Handler**.

Address	31-2	1	0	Name
$400F.E638		ADC1	ADC0	SYSCTL_RCGCADC_R

	31-14	13-12	11-10	9-8	7-6	5-4	3-2	1-0	
$4003.8020		SS3		SS2		SS1		SS0	ADC0_SSPRI_R

	31-16	15-12	11-8	7-4	3-0	
$4003.8014		EM3	EM2	EM1	EM0	ADC0_EMUX_R

	31-4	3	2	1	0	
$4003.8000		ASEN3	ASEN2	ASEN1	ASEN0	ADC0_ACTSS_R
$4003.80A0		MUX0				ADC0_SSMUX3_R
$4003.80A4		TS0	IE0	END0	D0	ADC0_SSCTL3_R
$4003.8028		SS3	SS2	SS1	SS0	ADC0_PSSI_R
$4003.8004		INR3	INR2	INR1	INR0	ADC0_RIS_R
$4003.8008		MASK3	MASK2	MASK1	MASK0	ADC0_IM_R
$4003.8FC4		Speed				ADC0_PC_R

	31-12	11-0	
$4003.80A8		DATA	ADC0_SSFIFO3_R

Table 10.3. The TM4C ADC registers. Each register is 32 bits wide.

Set Speed to 0001 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. 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. Write one to **ADC_ISC_R** to clear the corresponding bit in the **ADC_RIS_R** register.

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

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**. RXIFLSEL specifies the receive FIFO level that causes an interrupt (010 means interrupt on ≥ ½ full, or 7 to 8 characters). TXIFLSEL specifies the transmit FIFO level that causes an interrupt (010 means interrupt on ≤ ½ full, or 9 to 8 characters).

Address	31–12	11	10	9	8	7–0	Name
$4000.C000		OE	BE	PE	FE	DATA	UART0_DR_R

Address	31–3	3	2	1	0	Name
$4000.C004		OE	BE	PE	FE	UART0_RSR_R

Address	31–8	7	6	5	4	3	2–0	Name
$4000.C018		TXFE	RXFF	TXFF	RXFE	BUSY		UART0_FR_R

Address	31–16	15–0	Name
$4000.C024		DIVINT	UART0_IBRD_R

Address	31–6	5–0	Name
$4000.C028		DIVFRAC	UART0_FBRD_R

Address	31–8	7	6 – 5	4	3	2	1	0	Name
$4000.C02C		SPS	WPEN	FEN	STP2	EPS	PEN	BRK	UART0_LCRH_R

Address	31–10	9	8	7	6–3	2	1	0	Name
$4000.C030		RXE	TXE	LBE		SIRLP	SIREN	UARTEN	UART0_CTL_R

Address	31–6	5-3	2-0	Name
$4000.C034		RXIFLSEL	TXIFLSEL	UART0_IFLS_R

Address	31-11	10	9	8	7	6	5	4		Name
$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.