Final Exam

Date: May 13th 2016

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. *You will not reveal the contents of this exam to others who are taking the makeup thereby giving them an undue advantage*:

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.***

Problem 1	10	
Problem 2	15	
Problem 3	10	
Problem 4	15	
Problem 5	15	
Problem 6	10	
Problem 7	15	
Problem 8	10	
Total	100	

(10) Question 1. Please place **one word or short phrase** that best answers each question in the box.

(1) Part a) Which registers are NOT pushed on the stack when an interrupt occurs?

(1) Part b) Why did we use the 7406 driver for interfacing the LED? ……

(1) Part c) If the average rate at which you call **Fifo_Put** to enter data into a FIFO is less than the average rate at which you call **Fifo_Get** to remove data, …… could the FIFO ever become full?

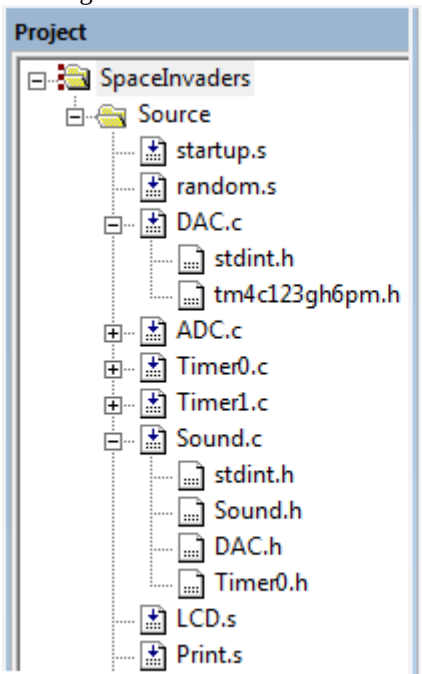
(1) Part d) Why do we add **`static`** to an otherwise local variable? …

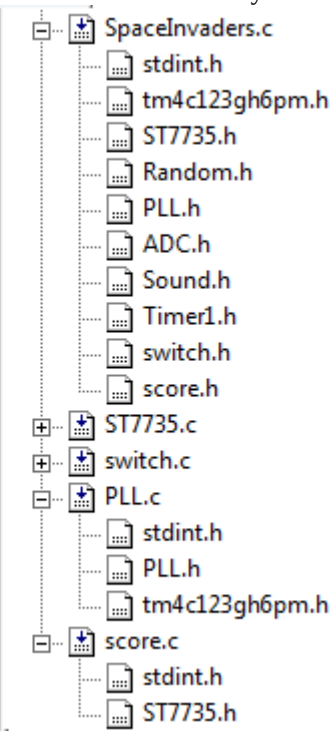
(1) Part e) Why do we add **`const`** to an otherwise global variable? ….

(1) Part f) Why do we add **`static`** to an otherwise global variable? …

(1) Part g) Why would you ever wish to use the PLL and slow down … the bus clock so the software runs slower?

(3) Part h) The following is the project window for my Lab 10 game. Draw a subset of the **call graph** of this system showing all the arrows into and out of the **Sound** module. Put your answer in the box.





(15) Question 2: *SysTick and Fixed-point*

(6) Part a) Write a function in C or assembly that uses SysTick delay for 1 millisecond. Assume that the SysTick is already initialized with **`NVIC_ST_RELOAD=0x00FFFFFF; NVIC_ST_CTRL=0x05;`** Assume that the clock is running at 50 MHz, such that each bus cycle is 20 ns.

```
void Delay1ms(){
```

(6) Part b) Write C code initialization that initializes SysTick to interrupt every 100 us. In this initialization, in addition to enabling and arming SysTick interrupts you should also enable interrupts in the processor. Run at priority level 1. Assume the bus clock is 80 MHz. You do not need to write the SysTick ISR.

```
void SysTick_Init100us(void){
```

(3) Part c) Consider the use of decimal fixed-point representation with a resolution of 0.01cm. What is the area of a rectangle in cm^2 whose width and length are represented by fixed-point integer values 250 and 110 respectively?

(10) Question 3: *FSM*

(5) Part a) A Moore FSM has 10 states, a 3-bit input (PB5-7), a 5-bit output (PB0-4) and a state dwell (or wait) time that can vary between 300ms to 1800ms. Complete the missing pieces in the definition of the struct for the State and FSM array declaration needed to implement the FSM.

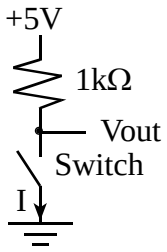
```
#define Init 0
struct State{




}
typedef struct State State_t;
```

```
[          ] FSM[    ]= { … contents are defined for you …}
uint8_t CS; // Index into the FSM array indicating current state
```

(5) Part a) Complete the FSM engine loop inside the main below. Assume you are given a function **`Delay1ms(unit32_t count)`** that delays for count milliseconds.

```
int main(void){
  PORTB_Init(); // Port B Initialization is done for you
  CS = Init;
  while(1){









  }
}
```

(15) Question 4: *Interfacing*

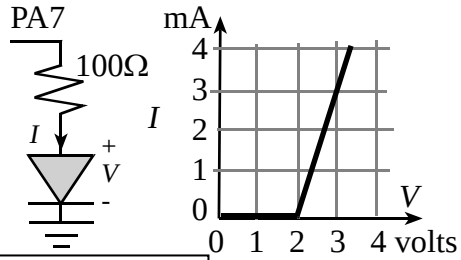


(3) Part a) Consider this negative logic switch circuit used in a +5V digital system. Do not consider the switch to be ideal. Rather, assume the resistance of the switch when the switch is open is 1 MΩ, and the resistance of the switch when the switch is closed is 1 Ω. How much current flows in mA through the switch when the switch is not pressed?

How much current in mA flows through the switch when the switch is pressed?

(6) Part b) The output on PA7 controls this LED. For LED voltages less than 2 volts, the LED current is 0. Assume the output high voltage of PA7 is 3.3V. For voltages above 2 volts, the LED current is

$I = 3 * (V - 2)$, where I is in mA, V is in volts.



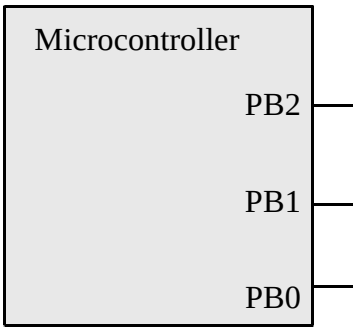
What are the **current, voltage,** and **power** to the LED when it is on?

V =

I =

P =

(6) Part c) Design a 3-bit DAC using multiple 10k resistors. No values other than 10k are allowed.



DACout

(15) Question 5: *FIFO queue*
You are asked to implement a FIFO that can handle up to **SIZE** elements. You cannot add additional global variables. You cannot change the function prototypes.
(3) Part a) Write the routine that initializes the FIFO.

```
#define SIZE 10
uint8_t FIFO[SIZE],GetI,PutI,Count;
void Fifo_Init(void){ // Initialize FIFO


}
```

(6) Part b) Write the routine that puts data into the FIFO. If the FIFO is full, this routine should spin (i.e., wait) until there is room in FIFO for the data. You can add local variables.

```
// enter one byte into the FIFO
void Fifo_Put(uint8_t data){









}
```

(6) Part c) Write the routine that gets data from the FIFO. If the FIFO is empty, this routine should spin (i.e., wait) until there is data in FIFO to return. You can add local variables.

```
// return one byte of data
// if empty spin until there is data in FIFO
uint8_t Fifo_Get(void){









}
```

(10) Question 6: *Local variables*

(3) Part a) What does **LCD_OutDec** print to screen?

```
void add1(uint32_t *in){
  *in = (*in)+1;
}
void func(void){
  uint32_t a = 10;
  add1(&a);
  LCD_OutDec(a);
}
```

Answer (select one option):

a) 11
b) 10
c) Some address of the stack region
d) Address of **a**
e) Unknown, unable to determine

(7) Part b) Implement both subroutines **func** and **add1** in assembly. For the subroutine **func**, use binding, allocation, access and deallocation of the local variable **a** on the stack. Implement each line of C explicitly in assembly. Use AAPCS. You must clearly identify each of the different stages. Use C statements as comments.

add1

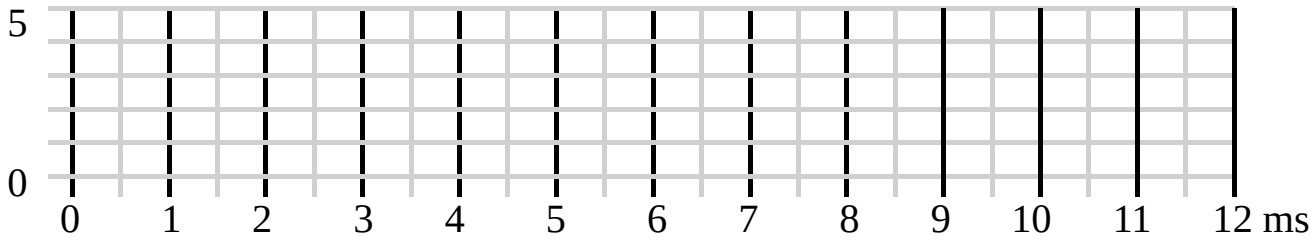
func

(15) Question 7: *UART and ADC*

(3) Part a) A programmer set the **`UART0_IBRD_R`** to 50 and **`UART0_FBRD_R`** to 0. If the bus clock frequency were 80 MHz, what would be the baud rate? Pick one answer.

i. 80 kbps
ii. 100 kbps
iii. 1 Mbps
iv. 120 kbps
v. 16 kbps
vi. None of the above

(6) Part b) Assume a serial port operating with a baud rate of 2000 bits per second. Draw the UART waveform when the decimal value 140 is transmitted. You may assume the channel is idle before and after the frame. Time flows from left to right. The dark vertical black lines correspond to 1-ms boundaries. Assume the frame begins at time = 1 ms, and show the waveform from 0 to 12 ms.



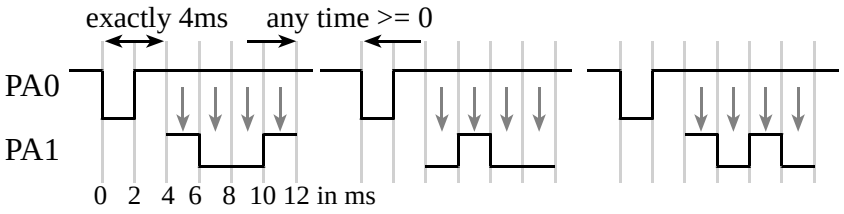
(3) Part c) You have a 12-bit, 0 to 3V range ADC. If the ADC input is 1.25 V, what will be the digital value **in hex** returned by this ADC?

(3) Part d) You are attempting to capture a sinusoidal sound with a frequency of 8 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?

(10) Question 8: *Design Problem*

You are to implement one-directional communication between two microcontrollers (sender and receiver) using two pins PA0 and PA1. The sender microcontroller programs these pins as output and the receiver programs them as input. PA0 is used as the control and PA1 is used to transfer the actual data. The bit-level protocol is as follows: when it has data to transmit, the sender sends a pulse on PA0, which is a low for 2 ms, followed by a high for 2 ms. The first bit of transmission (on PA1) immediately follows, with each bit sent lasting for exactly 2 ms. Each 4-bit transmission is preceded by a pulse on PA0. The time between transmissions can be any value greater than or equal to zero. See in the timeline below that the first byte transferred is 0x29. Note that each byte of data is transmitted as two 4-bit nibbles with the bit order as 0,1,2,3 then 4,5,6,7. The vertical arrows mark when you should read the input data. You may assume PA1 and PA0 are initialized to inputs.

(8) Part a) The overall goal of the communication is to transfer data from the sender to the receiver. In this section you will write a function that receives one byte of data. You may assume that you are given a function **`Delay1ms().`**



You may assume the software execution time is negligible compared to the Delay function

```
uint8_t ReceiveChar(void){
```

(2) Part b) What is maximum achievable **bandwidth** in bits/sec for this communication scenario?

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
```

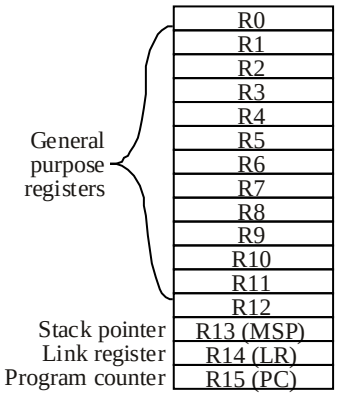
Notes `Ra Rd Rm Rn Rt represent 32-bit registers`

`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>`

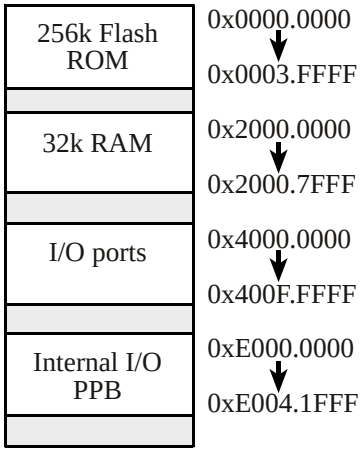
Examples of flexible operand **<op2>** creating the 32-bit number. E.g., **Rd = Rn+op2**

```
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.

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. LM3S has 10-bit data.

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).

	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.