Final Exam

Date: Dec 10, 2015

UT EID: _____

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:

- 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	12	
Problem 3	12	
Problem 4	12	
Problem 5	12	
Problem 6	12	
Problem 7	10	
Problem 8	20	
Total	100	

(10) Question 1. Please place **one letter/number** for each box. Choose the best answer to each question.

Part i) What the consequence of running the microcontroller at a higher speed? …… [ ] …….

Part ii) When is the 7406 driver needed in interfacing? …… [ ]

Part iii) What is the advantage of using binary vs. decimal fixed point? …… [ ]

Part iv) What does the prototype **void fun(uint32_t *)** say? …… [ ]

Part v) What memory does code get placed in? …… [ ]

Part vi) Why is the bandwidth always lower than the baud-rate? …… [ ]

Part Vii) How do we make a declaration be placed in ROM? …… [ ]

Part viii) The AAPCS convention requires that we maintain the contents of these registers inside a subroutine? …… [ ]

Part ix) Where are local variables allocated their space? …… [ ]

Part x) Which two shift operations are the same? …… [ ]

A) The Cortex M has a Harvard Architecture.
B) The PC always fetches instructions from flash memory in a von Neumann architecture.
C) ASL and LSL are the same.
D) ASR and LSR are the same.
E) In RAM because code needs to be able to grow the stack.
F) Registers R0 through R3.
G) Registers R4 through R11.
H) In ROM because it does not get modified.
I) UART uses start and stop bits in addition to the 8-bits of data.
J) Local variables are stored on the stack.
K) Local variables are stored in registers.
L) The LED needs more than 3.3 V.
M) The LED needs more than 10 mA.
N) Binary takes less space than decimal.
O) It creates a negative logic interface.
P) To satisfy the Nyquist Theorem.
Q) Binary fixed point math is simpler than decimal.
R) Because the UART sends a data bit value 0 as 0V and a data bit value 1 at 3.3V.
S) The function does not accept an 8 or 16 bit unsigned integer.
T) The receiver uses it to synchronize timing with the transmitter.
U) The function expects a pointer to a 32-bit unsigned integer as input.
V) Black box testing is more detailed than white box testing.
W) It decouples the production of data from the consumption of data.
X) The switch needs more than 10mA current.
Y) If running on battery we drain the battery faster.
Z) It provides for debugging, allowing you to download code and debug your software.
1) In order to handle either positive or negative values.
2) By using a static modifier in the declaration.
3) By using a const modifier in the declaration.
4) By using a volatile modifier in the declaration.
5) To tell the compiler the subroutine should not change its value.
6) Specifies it as an address or a pointer.

(12) Question 2

(5) Part a) What are the Condition code bits after each of the following ARM instructions are executed sequence?

Instructions	CC Bits
`MOV R0, #-1`	N=0; Z=0;V=0; C=0
`LSRS R0,#30`	
`SUBS R0,#1`	
`CMP R0,#4`	
`ADD R0,#1`	
`CMP R0,#3`	

(2) Part b) Calculate the divider and the Integer and fractional part of the Baud-rate assuming we want a 100kbps baud-rate. The clock rate is 50MHz.

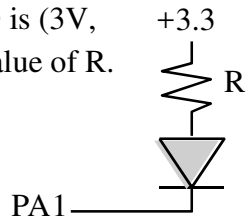
`UART0_IBRD_R =` ☐

`UART_FBRD_R =` ☐

(2) Part c) A repeating pulse signal that is ON for 5ms and OFF for 5ms is being sampled in an application. What should the sampling frequency be to faithfully capture the signal?

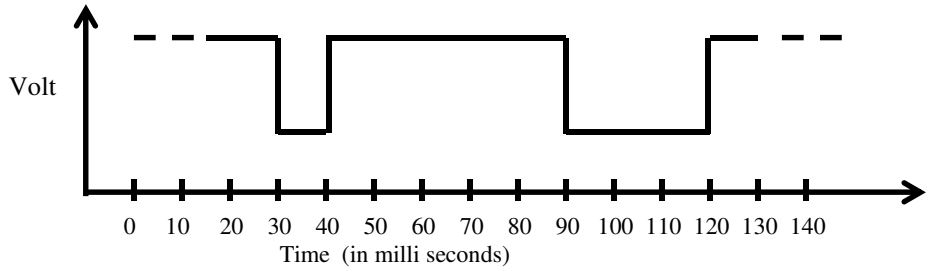
☐

(3) Part d) In the LED circuit interface to the right the operating point of the LED is (3V, 5mA) and the V_{OL}=0.5V. Will the interface work? If it works then calculate the value of R. If it does not work, explain why?



☐

(12) Question 3. Reverse-engineer UART parameters from the trace observed at a receiver below.



(2) Part a) What is the **integer** *data value* transferred over the UART?

(2) Part b) What is the *baud rate* in **bits/sec**?

(2) Part c) What is the *maximum bandwidth* in **bytes per second**?

(4) Part d) Assume the UART has been initialized with busy wait synchronization. Write a C function that writes one character to the UART.

(2) Part e) Assuming the clock is at 80MHz, what value was written to the IBRD register to achieve the baud-rate calculated above?

(12) Question 4.

a) (4 points) For a 8-bit ADC with an analog input voltage range of 0 to2.55V, what are the following:

(i) ADC precision

(ii) ADC range

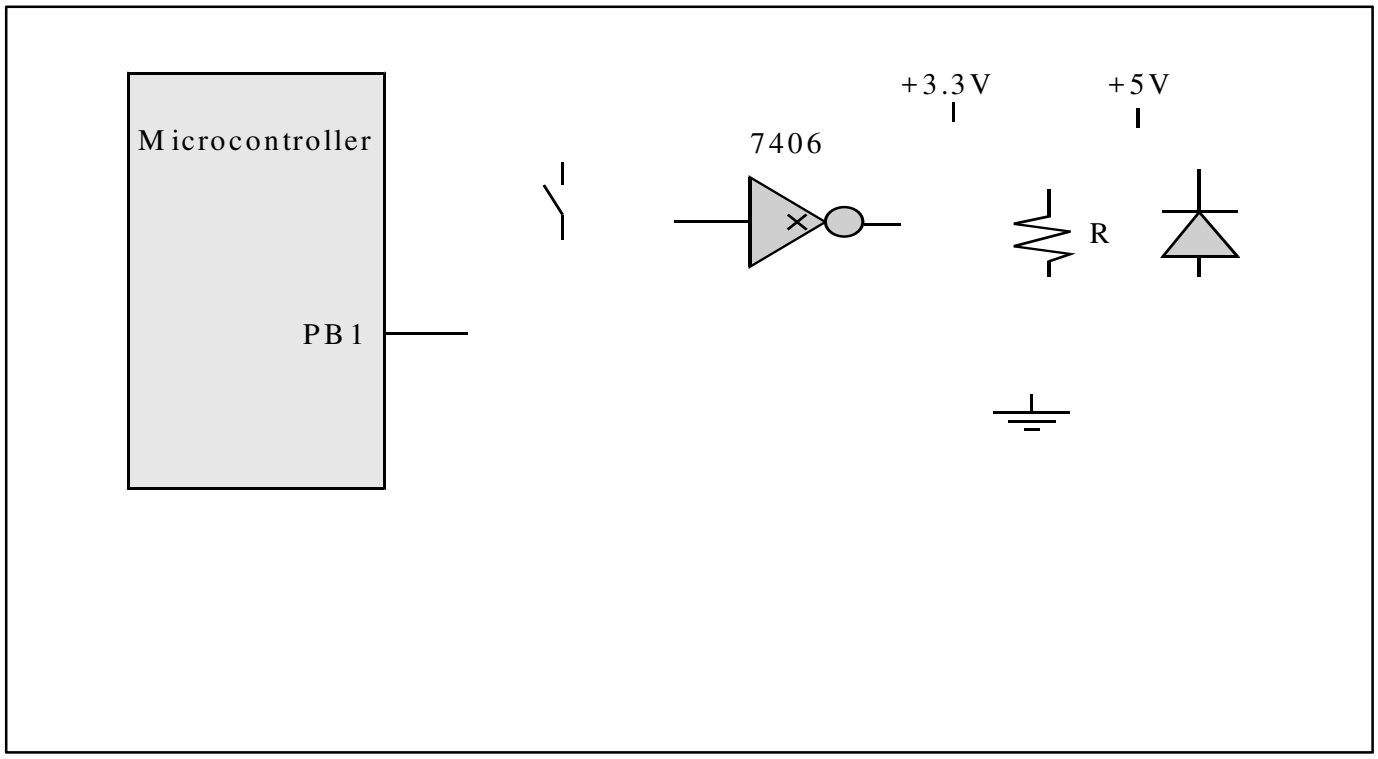
(iii) ADC resolution

b) (2 points) What will the above 8-bit ADC return if the input voltage is 1.0V?

c) (6 points) Write an ***ADC0_In*** function (in C) that uses busy-wait synchronization to sample the ADC. The function reads the ADC output, and returns the 8-bit binary number. Assume the ADC has already been initialized to use sequencer 3 with a software trigger and channel 1. See supplement pages for ADC registers.

```
uint8_t ADC0_In(void) {
```

(12) Question 5.
The desired operating point of an LED is 1V, 10mA. Interface this LED to PB12 using positive logic. Assume the V_{OL} of the 7406 is 0.5V. Assume the microcontroller output voltages are V_{OH} = 3.3V and V_{OL} = 0.2V. Specify values for any resistors needed. Show equations of your calculations used to select resistor values.
Draw the circuit along with any additional components needed.



(12) Question 7: FIFO
Write a C program that implements a FIFO data structure with exactly two elements. You have to implement the **`Fifo_Init, FiFo_Get`** and **`Fifo_Put`** functions. The FiFo declaration and function prototypes for the functions are given below. Feel free to add any global variables you need:

```
// Declarations
char FiFo[2];
// Add any other declarations here
```

```
//Prototypes of the stack functions that you can use
// Assume stacks do not overflow (infinite size)
void FiFo_Init();        // Initializes the FiFo variables
unit8_t FiFo_Put(char data);  // Adds data to Fifo returns 0/1
                              // for failure/success
unit8_t FiFo_Get(char *data); // Copies to *data from Fifo
                              //returns 0/1 for failure/success
```

(10) Question 8: Convert the C code into assembly, assuming the AAPCS parameter passing convention and local variables. Remember, local variables use the stack, not registers.

```
uint16_t pow(unit8_t base, uint8_t exp){
  uint16_t prod;
  uint16_t n;

  prod = 1;
  for(n=exp; n>0; n--) {
    prod = prod * base;
  }
  return prod;
}
```

(20) Question 9: **(Program)** A stepper motor can be controlled by writing a 4-bit number to it. The repeating sequence 5,6,10,9,5,6,10,9… moves it clockwise (CW) and the repeating sequence 5,9,10,6,5,9,10,6… moves it counter clockwise (CCW). The delay between writes determines the speed of the motor. Assume a constant speed of the motor with time between writes of 50ms. Design a Moore FSM with four states that takes a single input PA0 (0: CW; 1:CCW) and four outputs (PD0-3).

a) (10 points) Give the FSM state graph for the stepper motor.

b) (10 points) Complete the code below by adding state #defines and FSM array entries and the FSM loop.

```
struct State{
  uint8_t out;   // output produced in this state
  uint32_t wait; // delay in 10µs units
  uint8_t next[4]; // list of next states
};
typedef struct State SType;
SType FSM[4] = {




}
SType curState = ; //set the initial state here
int main() {
      // All Port Initialization done for you - Ccomplete the FSM loop below
      …
      while(1){




     }
}
```

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
```

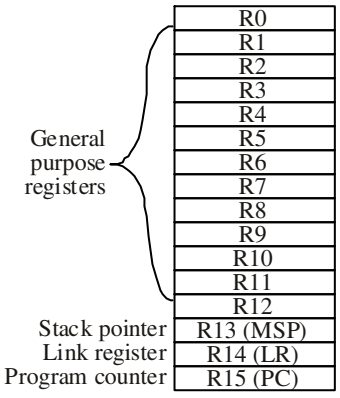
Notes `Ra Rd Rm Rn Rt represent 32-bit registers`

`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>

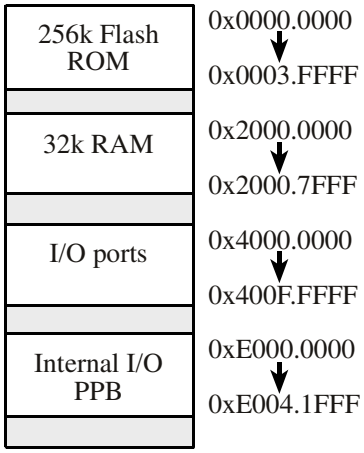
Examples of flexible operand **<op2>** creating the 32-bit number. E.g., **Rd = Rn+op2**

```
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 enable 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 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. 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.