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 _____

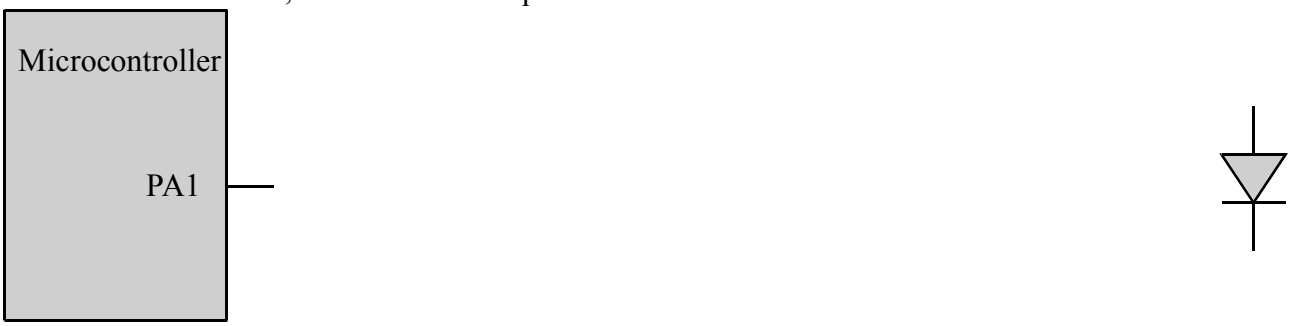
(10) Question 1. Consider a game that has 100 bouncing balls. There is an array of specifying the current status of each ball. Each ball has an (x,y) coordinate, a velocity, a direction and a color. You may assume the **Ball** array has been populated with data.

```
struct thing {
  unsigned char x;            // x coordinate, in cm
  unsigned char y;            // y coordinate, in cm
  short velocity;             // velocity, in cm/sec
  unsigned short angle;       // direction, in degrees
  unsigned long color;};      // RGB color
typedef thing thingType;
thingType Ball[100];
```

Write a C function that searches to see if two balls are occupying the same space (the x coordinates are equal and the y coordinates are equal). If two balls are occupying the same space, add 90 degrees to the angle of both balls, making sure the angles remain in the range of 0 to 359. E.g. 300+90 is 390, so set the angle to 30 degrees. Do not worry about 3 or more balls occupying the same space.

(5) Question 2. Interface a single-pole double-throw (SPDT) switch to input port PA0. If the switch is **not pressed**, then pin A is connected to pin B. If the switch is **pressed**, then pin C is connected to pin B. Pin A will never be connected to pin C, and pin B is always connected to either pin A or pin C. Implement the interface such that if the switch is pressed PA0 is high, and if the switch is not pressed PA0 is low. Do not use internal resistors. Show hardware connections; no software is required.

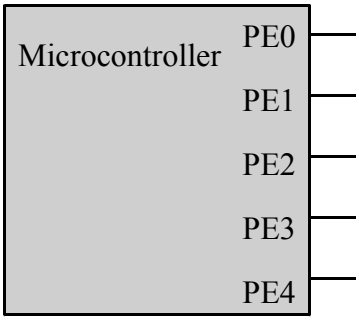
(5) Question 3. Interface an LED to PA1. Implement the interface in negative logic. The desired LED operating point is 1.2V 2mA. The V_{OH} is 3.0V and V_{OL} = 0.1V. Minimize cost of the interface. Show hardware connections; no software is required.



(8) Problem 4. Assume the UART0 has been initialized. Use busy-wait synchronization to implement a C function outputs a string to UART0. In C, strings are variable-length with null termination; your function uses call by reference. If you wish to call **`UART_OutChar`**, you must show the implementation of this subfunction.

```
void UART_OutString(unsigned char *pt){
```

(8) Question 5. Design a 5-bit DAC using the binary-weighted configuration. The DAC is controlled by five output port pins, PE4-0. Carefully label the signal which is the DAC output.



(6) Question 6. Add C code to define the following variables
v1 should be a public permanently-allocated 32-bit signed variable
v2 should be a temporary 32-bit unsigned variable private to the function **Fun_Init**
v3 should be a permanently-allocated 16-bit signed variable private to the function **Fun_Init**
v4 should be a permanently-allocated 16-bit signed variable, private to the file **Fun.c**.

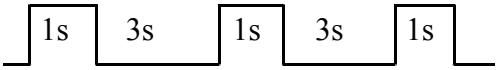
```
// This is the first line of the Fun.c code file



void Fun_Init(int in){     // code



}
// this is the last line of the Fun.c code file
```

(10) Question 7. Assume there is a buffer is defined in assembly with the equivalent size and type as the one shown in C on the right.

;assembly	// C
Buffer SPACE 400	long Buffer[100];

Show an **assembly subroutine** that sets each element of the buffer to its index value. Assuming **i** varies from 0 to 99, set **Buffer[i] = i;**

(10) Question 8. Write C or assembly code that creates this output on PA2 using SysTick interrupts. Assume the bus clock is 50 MHz. The pattern of high for 1 second and low for 3 seconds should repeat over and over. *Hint: since you do not have a calculator run the SysTick period at a convenient value, such as every 10ms or every 100ms.*



Part a) Show the initialization code that runs once

Part b) Show the SysTick ISR

(10) Question 9. State the term that is best described by each definition.
Part a) An address that specifies the location of an interrupt service routine.

Part b) A type of computer architecture where data is read from memory in the same way machine codes are fetched from memory.

Part c) The theorem that says the frequency at which the ADC is sampled must be higher than the frequency of the signal being sampled.

Part d) An interfacing approach where the hardware causes a specific software routine to be executed.

Part e) A debugging technique that stores strategic information into an array at run time, and the contents of the array are observed afterwards.

Part f) A term that describes a variable specifying whether some or all of the software has access to the variable. Hint: the answer is not private, and the answer is not public.

Part g) A measure of software size, specifying how many bytes of memory are required for the software.

Part h) A software step that explicitly clears the trigger flag. ----------

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

Part j) A type of digital logic where the output is either zero or off. ----------

(4) Question 10. The Stellaris LM3S1968 has a 0 to 3V 10-bit ADC. What will be the digital output of the ADC if the input voltage is 0.75 V? Give the answer in decimal.

(2) Question 11. If R0 equals -10, what will be in register R0 after executing these instructions?

```
LSL  R1,R0,#3
ADD  R0,R0,R1
```

(6) Question 12. Consider a SysTick ISR.
Part a) During the context switch from main program to ISR, which registers get pushed on the stack? Do not include any registers pushed by software as it executes.

Part b) The last instruction of an ISR is **`BX LR`**. For a regular subroutine return, **`BX LR`** simply puts LR into PC. For an interrupt return **`BX LR`** does something different. How does **`BX LR`** know not to put LR into PC, and instead what does **`BX LR`** do?

(10) Question 13. A distance is represented as unsigned binary fixed-point number with resolution of 2^{-4} cm. Assume the variable integer is 32 bits and unsigned. Assume the variable integer is passed by value into a subroutine using Register R0. Calculate the *cost* = (1.5 dollars/cm)**distance*. The cost is represented as an unsigned decimal fixed-point number with resolution of $0.01. The function should return the variable integer representing cost in Register R0. For example if the distance is 1.25 cm. The cost will be (1.5 dollars/cm)*1.25 cm = $1.87 (or $1.88 depending on how you round).
Part a) Let *I* be the variable integer representing *distance*. Give an equation relating *distance* and *I*?

Part b) Let *J* be the variable integer representing *cost*. Give an equation relating *cost* and *J*?

Part c) Write the assembly subroutine that converts distance to cost. Start with the desired operation
cost = (1.5 dollars/cm)**distance*
and then derive a function that can be used to calculate *J* from *I*. Optimize for speed, eliminate overflow, and minimize dropout. The input I is passed by value in R0, and the output J is returned by value in R0.

(6) Question 14. Consider this FIFO get function. There are no bugs in either implementation, but there are three missing values in the assembly version. Fill in the three values on the answer sheet.

```
pt        EQU  ??(a)??
Fifo_Get  PUSH {R0} ;allocate local
          PUSH {R4,R5}
          LDR  R0,=PutPt
          LDR  R0,[R0]
          LDR  R1,=GetPt
          LDR  R2,[R1]
          CMP  R2,R0
          BNE  NotEmpty
          MOV  R0,#0
          B    done
NotEmpty  LDRSB R3,[R2]
          LDR  R4,[SP,#pt]
          STRB R3,[R4]
          ADD  R2,R2,#??(b)??
          LDR  R5,=Fifo+FIFOSIZE
          CMP  R2,R5
          BNE  NoWrap
          LDR  R2,=Fifo
NoWrap    STR  R2,[R1]
done      POP  {R4,R5}
          ADD  SP,SP,#??(c)??
          BX   LR
```

```
#define FIFOSIZE 10
char volatile *PutPt;
char volatile *GetPt;
char static Fifo[FIFOSIZE];
int Fifo_Get(char *pt){
  if(PutPt == GetPt){
    return(0);
  }
  *pt = *(GetPt);
  GetPt++;
  if(GetPt== &Fifo[FIFOSIZE]){
     GetPt = &Fifo[0];
  }
  return(1);
}
```

Part a) What is **??(a)??**

Part b) What is **??(b)??**

Part c) What is **??(c)??**

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
```

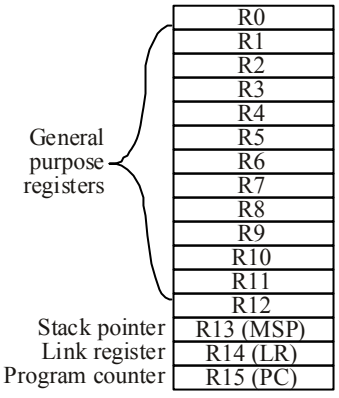
Notes `Ra Rd Rm Rn Rt represent 32-bit registers`

```
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>
```

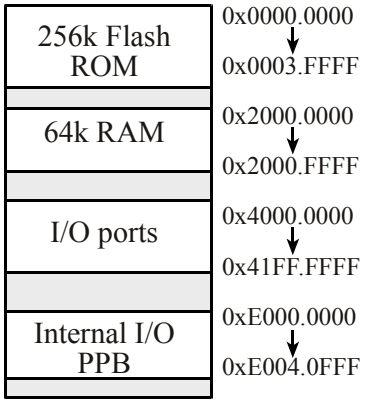
Examples of flexible operand **<op2>** creating the 32-bit number. E.g., **Rd = Rn+op2**

```
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

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.