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.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem 1</td>
<td>10</td>
</tr>
<tr>
<td>Problem 2</td>
<td>12</td>
</tr>
<tr>
<td>Problem 3</td>
<td>12</td>
</tr>
<tr>
<td>Problem 4</td>
<td>12</td>
</tr>
<tr>
<td>Problem 5</td>
<td>12</td>
</tr>
<tr>
<td>Problem 6</td>
<td>12</td>
</tr>
<tr>
<td>Problem 7</td>
<td>10</td>
</tr>
<tr>
<td>Problem 8</td>
<td>20</td>
</tr>
<tr>
<td>Total</td>
<td>100</td>
</tr>
</tbody>
</table>
(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?

<table>
<thead>
<tr>
<th>Instructions</th>
<th>CC Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV R0, #−1</td>
<td>N=0; Z=0; V=0; C=0</td>
</tr>
<tr>
<td>LSR S R0, #30</td>
<td></td>
</tr>
<tr>
<td>SUBS R0, #1</td>
<td></td>
</tr>
<tr>
<td>CMP R0, #4</td>
<td></td>
</tr>
<tr>
<td>ADD R0, #1</td>
<td></td>
</tr>
<tr>
<td>CMP R0, #3</td>
<td></td>
</tr>
</tbody>
</table>

(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.

\[
\text{UART0_IBRD_R} = \underline{ } \\
\text{UART_FBRD_R} = \underline{ }
\]

(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?

[Diagram of LED circuit]
(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 to 2.55V, 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.

```c
t uint8_t ADC0_In(void) {
```
**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:

```c
// 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.

```c
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.

```c
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 - Complete the FSM loop below
    ...
    while(1){
        ...
    }
}
```
Memory access instructions

\text{LDR} \hspace{1em} \text{Rd, [Rn]} \hspace{1em}; \text{load 32-bit number at [Rn] to Rd}
\text{LDR} \hspace{1em} \text{Rd, [Rn,#off]} \hspace{1em}; \text{load 32-bit number at [Rn+off] to Rd}
\text{LDR} \hspace{1em} \text{Rd, =value} \hspace{1em}; \text{set Rd equal to any 32-bit value (PC rel)}
\text{LDRH} \hspace{1em} \text{Rd, [Rn]} \hspace{1em}; \text{load unsigned 16-bit at [Rn] to Rd}
\text{LDRH} \hspace{1em} \text{Rd, [Rn,#off]} \hspace{1em}; \text{load unsigned 16-bit at [Rn+off] to Rd}
\text{LDRSH} \hspace{1em} \text{Rd, [Rn]} \hspace{1em}; \text{load signed 16-bit at [Rn] to Rd}
\text{LDRSH} \hspace{1em} \text{Rd, [Rn,#off]} \hspace{1em}; \text{load signed 16-bit at [Rn+off] to Rd}
\text{LDRB} \hspace{1em} \text{Rd, [Rn]} \hspace{1em}; \text{load unsigned 8-bit at [Rn] to Rd}
\text{LDRB} \hspace{1em} \text{Rd, [Rn,#off]} \hspace{1em}; \text{load unsigned 8-bit at [Rn+off] to Rd}
\text{LDRSB} \hspace{1em} \text{Rd, [Rn]} \hspace{1em}; \text{load signed 8-bit at [Rn] to Rd}
\text{LDRSB} \hspace{1em} \text{Rd, [Rn,#off]} \hspace{1em}; \text{load signed 8-bit at [Rn+off] to Rd}
\text{STR} \hspace{1em} \text{Rt, [Rn]} \hspace{1em}; \text{store 32-bit Rt to [Rn]}
\text{STR} \hspace{1em} \text{Rt, [Rn,#off]} \hspace{1em}; \text{store 32-bit Rt to [Rn+off]}
\text{STRH} \hspace{1em} \text{Rt, [Rn]} \hspace{1em}; \text{store least sig. 16-bit Rt to [Rn]}
\text{STRH} \hspace{1em} \text{Rt, [Rn,#off]} \hspace{1em}; \text{store least sig. 16-bit Rt to [Rn+off]}
\text{STRB} \hspace{1em} \text{Rt, [Rn]} \hspace{1em}; \text{store least sig. 8-bit Rt to [Rn]}
\text{STRB} \hspace{1em} \text{Rt, [Rn,#off]} \hspace{1em}; \text{store least sig. 8-bit Rt to [Rn+off]}
\text{PUSH} \hspace{1em} \{\text{Rt}\} \hspace{1em}; \text{push 32-bit Rt onto stack}
\text{POP} \hspace{1em} \{\text{Rd}\} \hspace{1em}; \text{pop 32-bit number from stack into Rd}
\text{ADR} \hspace{1em} \text{Rd, label} \hspace{1em}; \text{set Rd equal to the address at label}
\text{MOV(S)} \hspace{1em} \text{Rd, \langle op2\rangle} \hspace{1em}; \text{set Rd equal to op2}
\text{MVN(S)} \hspace{1em} \text{Rd, \langle op2\rangle} \hspace{1em}; \text{set Rd equal to -op2}

Branch instructions

\text{B} \hspace{1em} \langle \text{label} \rangle \hspace{1em}; \text{branch to label} \hspace{1em} \text{Always}
\text{BEQ} \hspace{1em} \langle \text{label} \rangle \hspace{1em}; \text{branch if Z == 1} \hspace{1em} \text{Equal}
\text{BNE} \hspace{1em} \langle \text{label} \rangle \hspace{1em}; \text{branch if Z == 0} \hspace{1em} \text{Not equal}
\text{BCS} \hspace{1em} \langle \text{label} \rangle \hspace{1em}; \text{branch if C == 1} \hspace{1em} \text{Higher or same, unsigned ≥}
\text{BHS} \hspace{1em} \langle \text{label} \rangle \hspace{1em}; \text{branch if C == 1} \hspace{1em} \text{Higher or same, unsigned ≥}
\text{BCC} \hspace{1em} \langle \text{label} \rangle \hspace{1em}; \text{branch if C == 0} \hspace{1em} \text{Lower, unsigned <}
\text{BLO} \hspace{1em} \langle \text{label} \rangle \hspace{1em}; \text{branch if C == 0} \hspace{1em} \text{Lower, unsigned <}
\text{BMI} \hspace{1em} \langle \text{label} \rangle \hspace{1em}; \text{branch if N == 1} \hspace{1em} \text{Negative}
\text{BPL} \hspace{1em} \langle \text{label} \rangle \hspace{1em}; \text{branch if N == 0} \hspace{1em} \text{Positive or zero}
\text{BVS} \hspace{1em} \langle \text{label} \rangle \hspace{1em}; \text{branch if V == 1} \hspace{1em} \text{Overflow}
\text{BVC} \hspace{1em} \langle \text{label} \rangle \hspace{1em}; \text{branch if V == 0} \hspace{1em} \text{No overflow}
\text{BHI} \hspace{1em} \langle \text{label} \rangle \hspace{1em}; \text{branch if C=1 and Z=0} \hspace{1em} \text{Higher, unsigned >}
\text{BLS} \hspace{1em} \langle \text{label} \rangle \hspace{1em}; \text{branch if C=0 or Z=1} \hspace{1em} \text{Lower or same, unsigned ≤}
\text{BGE} \hspace{1em} \langle \text{label} \rangle \hspace{1em}; \text{branch if N == V} \hspace{1em} \text{Greater than or equal, signed ≥}
\text{BLT} \hspace{1em} \langle \text{label} \rangle \hspace{1em}; \text{branch if N != V} \hspace{1em} \text{Less than, signed <}
\text{BGT} \hspace{1em} \langle \text{label} \rangle \hspace{1em}; \text{branch if Z=0 and N==V} \hspace{1em} \text{Greater than, signed >}
\text{BLE} \hspace{1em} \langle \text{label} \rangle \hspace{1em}; \text{branch if Z==1 or N!=V} \hspace{1em} \text{Less than or equal, signed ≤}
\text{BX} \hspace{1em} \text{Rm} \hspace{1em}; \text{branch indirect to location specified by Rm}
\text{BL} \hspace{1em} \langle \text{label} \rangle \hspace{1em}; \text{branch to subroutine at label}
\text{BLX} \hspace{1em} \text{Rm} \hspace{1em}; \text{branch to subroutine indirect specified by Rm}

Interrupt instructions

\text{CPSIE} \hspace{1em} \text{I} \hspace{1em}; \text{enable interrupts} \hspace{1em} \text{(I=0)}
\text{CPSID} \hspace{1em} \text{I} \hspace{1em}; \text{disable interrupts} \hspace{1em} \text{(I=1)}

Logical instructions

\text{AND(S)} \hspace{1em} \{\text{Rd,} \} \hspace{1em} \text{Rn, \langle op2\rangle} \hspace{1em}; \text{Rd=Rn&op2} \hspace{1em} \text{(op2 is 32 bits)}
\text{ORR(S)} \hspace{1em} \{\text{Rd,} \} \hspace{1em} \text{Rn, \langle op2\rangle} \hspace{1em}; \text{Rd=Rn|op2} \hspace{1em} \text{(op2 is 32 bits)}
\text{EOR(S)} \hspace{1em} \{\text{Rd,} \} \hspace{1em} \text{Rn, \langle op2\rangle} \hspace{1em}; \text{Rd=Rn^op2} \hspace{1em} \text{(op2 is 32 bits)}
\text{BIC(S)} \hspace{1em} \{\text{Rd,} \} \hspace{1em} \text{Rn, \langle op2\rangle} \hspace{1em}; \text{Rd=Rn&(~op2)} \hspace{1em} \text{(op2 is 32 bits)}
\text{ORN(S)} \hspace{1em} \{\text{Rd,} \} \hspace{1em} \text{Rn, \langle op2\rangle} \hspace{1em}; \text{Rd=Rn|(~op2)} \hspace{1em} \text{(op2 is 32 bits)}
\text{LSR(S)} \hspace{1em} \text{Rd, Rm, Rs} \hspace{1em}; \text{logical shift right Rd=Rm>>Rs} \hspace{1em} \text{(unsigned)}
\text{LSR(S)} \hspace{1em} \text{Rd, Rm, #n} \hspace{1em}; \text{logical shift right Rd=Rm>>n} \hspace{1em} \text{(unsigned)}
\text{ASR(S)} \hspace{1em} \text{Rd, Rm, Rs} \hspace{1em}; \text{arithmetic shift right Rd=Rm>>Rs} \hspace{1em} \text{(signed)}
Arithmetic instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASR(S) Rd, Rm, n</td>
<td>arithmetic shift right Rd=Rm&gt;&gt;n (signed)</td>
</tr>
<tr>
<td>LSL(S) Rd, Rm, Rs</td>
<td>shift left Rd=Rm&lt;&lt;Rs (signed, unsigned)</td>
</tr>
<tr>
<td>LSL(S) Rd, Rm, n</td>
<td>shift left Rd=Rm&lt;&lt;n (signed, unsigned)</td>
</tr>
</tbody>
</table>

Arithmetic instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD(S) {Rd,} Rn, &lt;op2&gt;</td>
<td>Rd = Rn + op2</td>
</tr>
<tr>
<td>ADD(S) {Rd,} Rn, #im12</td>
<td>Rd = Rn + im12, im12 is 0 to 4095</td>
</tr>
<tr>
<td>SUB(S) {Rd,} Rn, &lt;op2&gt;</td>
<td>Rd = Rn - op2</td>
</tr>
<tr>
<td>SUB(S) {Rd,} Rn, #im12</td>
<td>Rd = Rn - im12, im12 is 0 to 4095</td>
</tr>
<tr>
<td>RSB(S) {Rd,} Rn, &lt;op2&gt;</td>
<td>Rd = op2 - Rn</td>
</tr>
<tr>
<td>RSB(S) {Rd,} Rn, #im12</td>
<td>Rd = im12 - Rn</td>
</tr>
<tr>
<td>CMP Rn, &lt;op2&gt;</td>
<td>Rn - op2 sets the NZVC bits</td>
</tr>
<tr>
<td>CMN Rn, &lt;op2&gt;</td>
<td>Rn - (-op2) sets the NZVC bits</td>
</tr>
<tr>
<td>MUL(S) {Rd,} Rn, Rm</td>
<td>Rd = Rn * Rm signed or unsigned</td>
</tr>
<tr>
<td>MLA Rd, Rn, Rm, Ra</td>
<td>Rd = Ra + Rn*Rm signed or unsigned</td>
</tr>
<tr>
<td>MLS Rd, Rn, Rm, Ra</td>
<td>Rd = Ra - Rn*Rm signed or unsigned</td>
</tr>
<tr>
<td>UDIV {Rd,} Rn, Rm</td>
<td>Rd = Rn/Rm unsigned</td>
</tr>
<tr>
<td>SDIV {Rd,} Rn, Rm</td>
<td>Rd = Rn/Rm signed</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD Rd, Rn, Rm</td>
<td>op2 = Rm</td>
</tr>
<tr>
<td>ADD Rd, Rn, Rm, LSL #n</td>
<td>op2 = Rm&lt;&lt;n Rm is signed, unsigned</td>
</tr>
<tr>
<td>ADD Rd, Rn, Rm, LSR #n</td>
<td>op2 = Rm&gt;&gt;n Rm is unsigned</td>
</tr>
<tr>
<td>ADD Rd, Rn, Rm, ASR #n</td>
<td>op2 = Rm&gt;&gt;n Rm is signed</td>
</tr>
<tr>
<td>ADD Rd, Rn, #constant</td>
<td>op2 = constant where X and Y are hexadecimal digits:</td>
</tr>
<tr>
<td>• produced by shifting an 8-bit unsigned value left by any number of bits</td>
<td></td>
</tr>
<tr>
<td>• in the form 0x00XY00XY</td>
<td></td>
</tr>
<tr>
<td>• in the form 0X00XY00XY</td>
<td></td>
</tr>
<tr>
<td>• in the form 0xXYXYXYXY</td>
<td></td>
</tr>
</tbody>
</table>

Condition code bits

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

256k Flash ROM

32k RAM

I/O ports

Internal I/O PPB

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

<table>
<thead>
<tr>
<th>Address</th>
<th>31</th>
<th>30-7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>$400F.E608</td>
<td>GPIOF GPIOE GPIOD GPIOC GPIOB GPIOA SYSTCL_RCGCGPIO_R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$4000.5F3C</td>
<td>DATA DATA DATA DATA DATA DATA DATA GPIOPORTB_DATA_R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$4000.5F40</td>
<td>DIR DIR DIR DIR DIR DIR DIR GPIOPORTB_DIR_R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$4000.5F40</td>
<td>SEL SEL SEL SEL SEL SEL SEL GPIOPORTB_AFSSEL_R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$4000.5F5C</td>
<td>DEN DEN DEN DEN DEN DEN DEN GPIOPORTB_DEN_R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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_CTRL_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.

<table>
<thead>
<tr>
<th>Address</th>
<th>31-24</th>
<th>23-17</th>
<th>16</th>
<th>15-3</th>
<th>2</th>
<th>1</th>
<th>0</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>$4000.E010</td>
<td>0 0 0 COUNT 0 CLK_SRC INTEN ENABLE NVIC_EN0_R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$4000.E014</td>
<td>0 24-bit RELOAD value NVIC_ST_RELOAD_R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$4000.E018</td>
<td>0 24-bit CURRENT value of SysTick counter NVIC_ST_CURRENT_R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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_SSMUX3_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 \texttt{END0} will always be set, signifying this sample is the end of the sequence. Clear the \texttt{D0} bit. The \texttt{ADC\_RIS\_R} register has flags that are set when the conversion is complete, assuming the \texttt{IE0} bit is set. Do not set bits in the \texttt{ADC\_IM\_R} register because we do not want interrupts. Write one to \texttt{ADC\_ISC\_R} to clear the corresponding bit in the \texttt{ADC\_RIS\_R} register.

\texttt{UART0} pins are on PA1 (transmit) and PA0 (receive). The \texttt{UART0\_IBRD\_R} and \texttt{UART0\_FBRD\_R} registers specify the baud rate. The baud rate \texttt{divider} is a 22-bit binary fixed-point value with a resolution of $2^{-6}$. The \texttt{Baud16} clock is created from the system bus clock, with a frequency of (Bus clock frequency)/\texttt{divider}. The baud rate is

$$\text{Baud rate} = \text{Baud16/16} = \frac{(\text{Bus clock frequency})}{(16 \times \text{divider})}$$

We set bit 4 of the \texttt{UART0\_LCRH\_R} to enable the hardware FIFOs. We set both bits 5 and 6 of the \texttt{UART0\_LCRH\_R} to establish an 8-bit data frame. The \texttt{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 \texttt{UART0\_IM\_R} register. To acknowledge an interrupt (make the trigger flag become zero), software writes a 1 to the corresponding bit in the \texttt{UART0\_IC\_R} register.

We set bit 0 of the \texttt{UART0\_CTL\_R} to enable the UART. Writing to \texttt{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 placed in a 16-deep receive hardware FIFO. Reading from \texttt{UART0\_DR\_R} register will get one data from the receive hardware FIFO. The status of the two FIFOs can be seen in the \texttt{UART0\_FR\_R} register (FF is FIFO full, FE is FIFO empty). The standard name for the \texttt{UART0} ISR is \texttt{UART0\_Handler}. \texttt{RXIFLSEL} specifies the receive FIFO level that causes an interrupt (010 means interrupt on ≥½ full, or 7 to 8 characters). \texttt{TXIFLSEL} specifies the transmit FIFO level that causes an interrupt (010 means interrupt on ≤½ full, or 9 to 8 characters).