Final Exam Solutions

Date: May 14, 2015

UT EID: ________________________ Circle one: MT, NT, JV, RY, VJR

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	10	
Problem 3	10	
Problem 4	10	
Problem 5	10	
Problem 6	10	
Problem 7	12	
Problem 8	8	
Problem 9	20	
Total	100	

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

Part i) Why do we sometimes use the phase lock loop? .... Y

Part ii) Why did we use the open collector 7406 gate to interface the LED? .... M

Part iii) Why did we use fixed-point to represent measured distance? .... G

Part iv) Why did we dump input/output data into buffers in Lab 4? .... J

Part v) Why do we put programs in flash memory? 3

Part vi) Why does the UART use start and stop bits? .... T

Part Vii) Why do we specify a global variable as `static`? .... D

Part viii) Why do we specify a local variable as `static`? 2

Part ix) Why do the I/O definitions have `volatile` in the definitions? 4

Part x) Why do we specify a function parameter as `const`? 5

A) The Cortex M has a Harvard Architecture.
B) The PC always fetches instructions from flash memory in a von Neumann architecture.
C) Some machine instructions are 16 bits and others are 32 bits.
D) It reduces the scope of the data making the data private to the file.
E) The Cortex M processor on the TM4C123 does not support floating point operations.
F) The left/right shift is faster than multiply/divide.
G) In order to represent non-integer values.
H) To create bounded latency and provide for real-time operation.
I) It is nonintrusive debugging.
J) It is minimally intrusive debugging.
K) The interface must control both voltage and current so the LED is the proper brightness.
L) The LED needs more than 3.3 V.
M) The LED needs more than 8 mA.
N) Buffers can store more data than can be printed using the UART.
O) It creates a negative logic interface.
P) To satisfy the Nyquist Theorem.
Q) It illustrates to our client how the program works.
R) Because the UART sends a data bit value 0 as 0V and a data bit value 1 at 3.3V.
S) Message can vary in length and it is used signify the end of the message.
T) The receiver uses it to synchronize timing with the transmitter.
U) It provides a mechanism to minimize bandwidth.
V) Black box testing is more detailed than white box testing.
W) It decouples the production of data from the consumption of data.
X) It provides for ceiling and floor.
Y) If we slow down processor execution, it will save power. If we execute faster, we do more processing.
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) To allocate it in RAM, making it persistent across subroutine calls.
3) To allocate it in ROM, and ROM is nonvolatile.
4) To tell the compiler to fetch a new value each time it is accessed.
5) To tell the compiler the subroutine should not change its value.
6) Specifies it as an address or a pointer.

(10) Question 2

(2) Part a) What are the addressing modes used in the following ARM instructions?

Instructions	**Addressing Modes**
`MOV R0, #10`	Immediate mode
`LDR R0, [R1]`	Indexed addressing
`BL sublabel`	PC-relative
`ADD R2, R1`	Register
`PUSH {R4-R11, LR}`	Register list

(2) Part b) In order to specify the desired baud rate for a bus clock frequency is 80 MHz, the divider has been correctly calculated as 50.125. What values should the **UART0_IBRD_R** and the **UART0_FBRD_R** registers be initialized to?

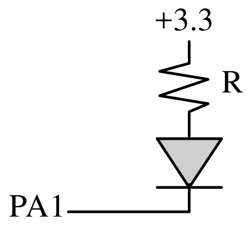
`UART0_IBRD_R =` 50

`UART_FBRD_R =` 8

(2) Part c) If the ADC sampling frequency is 100 Hz, what range of frequencies in the analog input can safely be represented in the digital samples?

Nyquist Theorem, 0 to 50 Hz

(2) Part d) Consider an LED with a desired operating point of (I_d,V_d). Let V_{OL} V_{OH} I_{OL} and I_{OH} be the operating parameters of the digital output on PA1. What is the design equation needed to calculate the desired resistance *R* for this circuit?

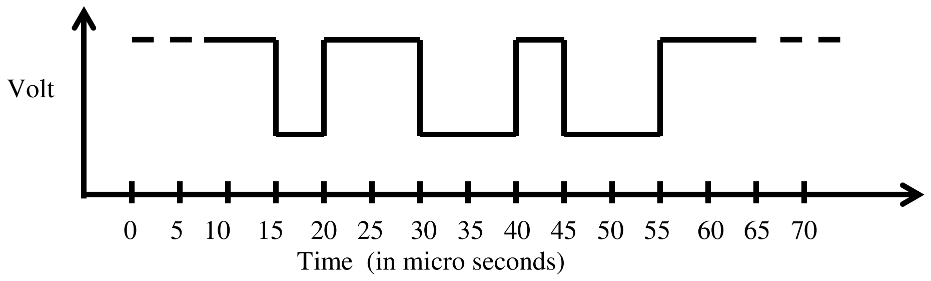


$R = (3.3 - V_d - V_{OL})/I_d$

(2) Part e) What is the relationship between the range, precision and resolution of an ADC, given that the sampling frequency is *f*?

Range = precision * resolution

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



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

1001_0011 -> 0x93

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

1/5us = 200k bit/s

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

0.8*200/8 =20k byte/s

(3) Part d) Assume the UART has been initialized with busy wait synchronization. Write a C function that reads one character from the UART.

```
char UART_Read(void){
  while((UART0_FR_R & 0x0010) == 0);  // RXFE
  return ((char)(UART0_DR_R & 0xFF));}
}
```

(2) Part e) Assume the receiver software uses busy-wait synchronization. The main program reads all the data available from the UART and then processes the data. The maximum time required to process the data is 125us. Is it possible to lose data? If so, explain how to change the UART so no data is lost. If no data can be lost, explain how the UART works so no data are lost.

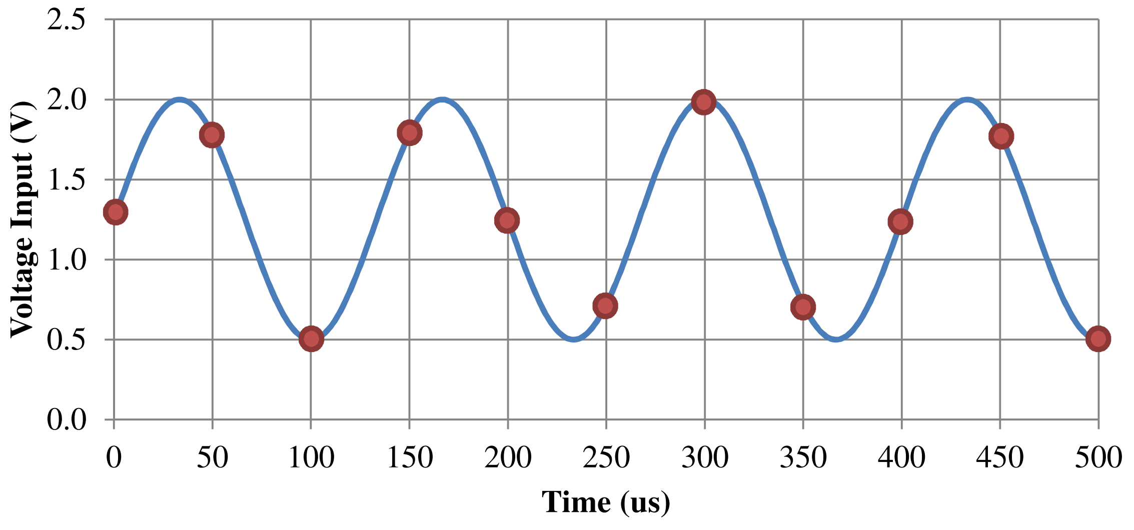
Add FIFOs for rate matching. At least 3 bytes. The TM4C123 will not lose data because it has a 16-element FIFO

(10) Question 4. Analog Devices AD7641 is an 18-bit, 0 to 2.5V range, 2MSPS SAR ADC. A student is attempting to capture a sinusoid signal of frequency 7.5 kHz using the AD7641. Using the 18-bit ADC and periodic interrupt, he programs the system to interrupt at a frequency of 20 kHz. Each time the system interrupts, he calls **`AD7641_In()`** to get one sample of the signal from the AD7641.

(2) Part a) If the AD7641 input is 1.25 V, what will be the digital value in hex returned by this ADC?

2^{17}=0x10000

(4) Part b) Assuming the first sample is taken at time t=0, mark the (time, voltage) points on the plot below specifying the data collected by the ADC. Red dots are the digital samples

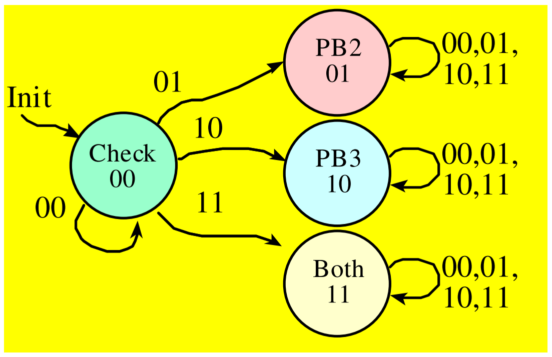


(4) Part c) Is it possible to recreate the original signal from the captured samples? If your answer is *yes*, explain how. If your answer is *no*, what is the term used to refer to this loss of information?

Yes, it is possible because 7.5 kHz is less than ½ f_s according to the Nyquist Theorem

(10) Question 5. You will design an embedded system using a Moore FSM. There are two inputs (PB3, PB2) and two outputs (PB1, PB0). The FSM runs in the background with 1 kHz SysTick periodic interrupts. Initially both outputs will be low, and you may also assume both inputs are initially low. *If PB3 rises before PB2 rises, then set PB1 high. If PB2 rises before PB3 rises, then set PB0 high. If both rise during the same 1-ms window, set both PB1 and PB0 high. After either or both PB1 and/or PB0 become high, let the output remain fixed.* The initial state is s=0.

(4) Part a) Show the FSM graph in Moore format. Full credit for the solution with the fewest states.



(6) Part b) The **`struct`** and the main program are fixed. Show the C code that places the FSM in ROM, and write the SysTick ISR. **`PORTB_Init`** initializes PB3-PB0 and makes the outputs low. **`SysTick_Init`** initializes interrupts at 1 kHz. **`PORTB_Init`** and **`SysTick_Init`** are given. Full credit awarded for friendly access and good programming style. The initial state will be s=0.

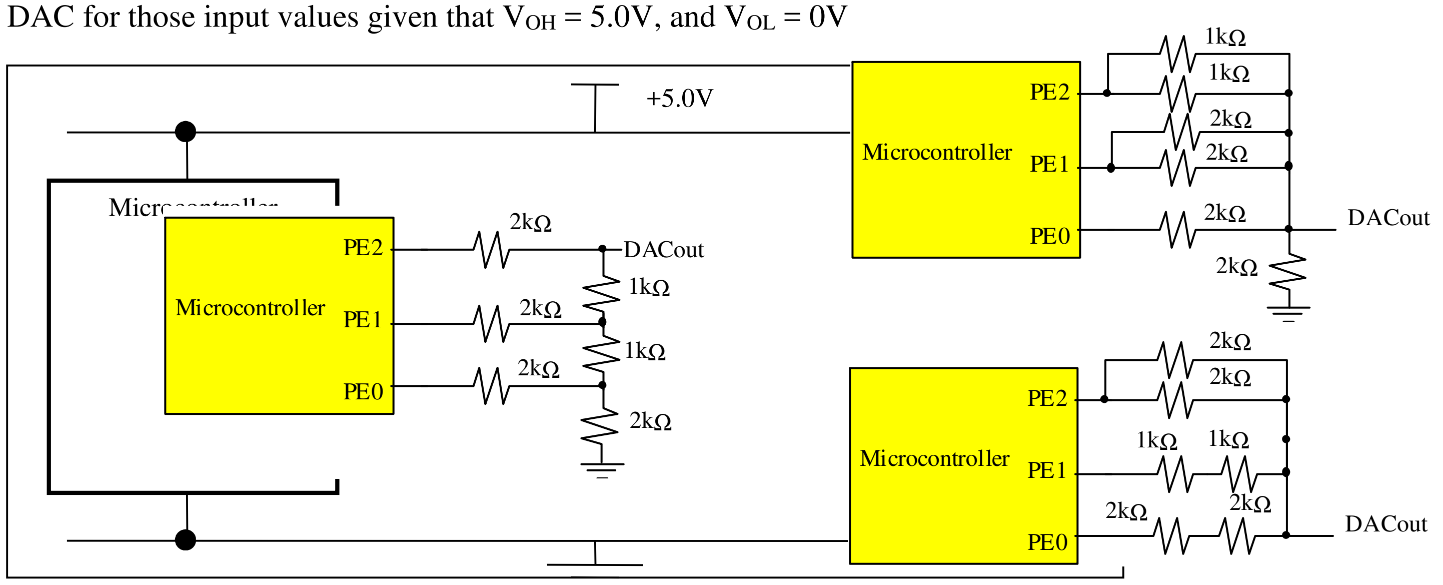
```
const struct State{
  uint32_t out;
  uint32_t next[4];
};
typedef const struct State State_t;
uint32_t s;  // state number
```

```
// Initialize array of states
#define Check  0
#define PB2    1
#define PB3    2
#define Both   3
State_t FSM[4]={
 {0,  {   Check,PB2, PB3,Both }},
 {1,  {   PB2,  PB2, PB2, PB2 }},
 {2,  {   PB3,  PB3, PB3, PB3 }},
 {3,  {   Both, Both,Both,Both}}};
```

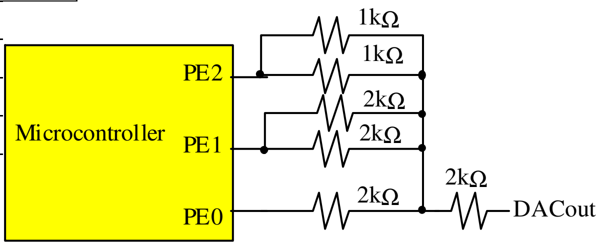
```
void main(void){ PORTB_Init();
  s = 0; // initial state
  SysTick_Init();
  EnableInterrupts();
  while(1){}}
void SysTick_Handler(void){
```

```
// read input

  in = GPIO_PORTB_DATA_R&0x0C)>>2;

// change state

  s = FSM[s].next[in];

// friendly write output
  out = GPIO_PORTB_DATA_R&(~0x03);
  out |= FSM[s].out;
  GPIO_PORTB_DATA_R = out;

}
```

(10) Question 6. (a): You are given two 1 kΩ resistors and four 2 kΩ resistors. Build a 3-bit DAC circuit (connected to PE2, PE1, PE0) using *all* resistors. Complete the table below where a few of the input logic voltage values at PE2, PE1, PE0 are shown. Calculate the output voltage Vout of the DAC for those input values given that V_{OH} = 5.0V, and V_{OL} = 0V



PE2	PE1	PE0	Vout
0	0	0	0V
0	0	1	0.625V
0	1	0	1.25V
1	0	0	2.5V



(b) The output of the DAC circuit you built in part (a) is now connected to a speaker whose resistance is very low and can be approximated to be 0 Ω. Calculate the current through the speaker when the logic voltage values at PE2, PE1, PE0 are 100. Show your work

For the R-2R ladder circuit:
Current = 5/2k = 2.5 mA

For the weighted DAC circuit:
Current = 5/1k = 5mA

(12) Question 7: FIFO

(2) Part a) What is the difference between *FIFO* and *Mailbox*?

Mailbox holds one piece of data, while a FIFO can hold multiple data in a first in first out manner.

(10) Part b) Write a C program that implements FIFO using two stack data structures. You have to implement the **Fifo_Get** function using two stacks. **Fifo_Put** is already given to you. Return a value of -1 if the FIFO is empty. The stack data functions are given to you, having *push*, *pop* and *empty* functions that you must use. The function prototypes for these functions are given below.

```
// Prototypes of the stack functions that you can use
// Assume stacks do not overflow (infinite size)
int  pop1();        // Gets the element at the top of the stack1
void push1(int);    // Puts the element at the top of the stack1
int  empty1();      // Returns 1 if stack1 is empty, 0 otherwise
int  pop2();        // Gets the element at the top of the stack2
void push2(int);    // Puts the element at the top of the stack2
int  empty2();      // Returns 1 if stack2 is empty, 0 otherwise

// Put an element into the back of the FIFO.
// data is never -1 (the error code)
void Fifo_Put(int data) {
  push1(data); // pushes element data onto stack1
}
```

```
// Get the element at the head of the FIFO
int Fifo_Get(void) {
     int value;
     if (!empty2()) {
          return pop2();
     }

     while (!empty1()) {
          value = pop1();
          push2(value);
     }

     if (!empty2() {
          return pop2();
     }
     return -1;

}
```

(8) Question 8: Convert the C code into assembly, using local variable allocation phases. Remember, local variables use the stack, not registers. Put exactly one assembly line into each box.

```
; *****binding phase***************
sum   EQU   0   ;16-bit unsigned number
n     EQU   2   ;16-bit unsigned number
; 1)*****allocation phase *********
calc PUSH {R4,LR}
      SUB   SP,#4          ;allocate 4 bytes
; 2)******access phase ************
      MOV   R0,#0
      STRH  R0,[SP,#sum]       ;sum=0
      MOV   R1,#255
      STRH  R1,[SP,#n]         ;n=255
loop  LDRH  R1,[SP,#n]         ;R1=n
      LDRH  R0,[SP,#sum]       ;R0=sum
      ADD   R0,R1              ;R0=sum+n
      STRH  R0,[SP,#sum]       ;sum=sum+n
      LDRH  R1,[SP,#n]          ;R1=n
      SUBS  R1,#1              ;n-1
      STRH  R1,[SP,#n]         ;n=n-1
      BNE   loop
; 3)******deallocation phase *****
      ADD   SP,#4     ;deallocation
      POP   {R4,PC}   ;R0=sum
```

```
uint16_t calc(void){
  uint16_t sum;

  uint16_t n;

  sum = 0;

  for(n=255; n>0; n--) {
    sum=sum+n;
  }

  return sum;
}
```

(20) Question 9: **(Program)** Many of you could not play your ideal music for Lab 10. Valvano had enough with people asking for Full licenses on Keil, he knows sound files are the source of the problem, they are simply too big. You are told to change the coding of the sound files so the resulting array packs two 4-bit samples into one byte, resulting in a compression ratio of 2:1. All wav files are converted to 4-bit samples at 8 kHz. For example the first ten 4-bit samples of the **start** sound (see below) are 8,9,9,10,11,11,12,14,15,15. Notice 8,9 are "packed" into the byte 0x89. During play time, the packed values are decompressed into their original form and sent to the DAC. The game has 4 sounds, each in a different array named **startS, shootS, deathS** and **quietS**, each of a different length. The following code declares the constants, variables and structure used in the solution. Read the code carefully and answer the below questions.

```
#define start 0
#define shoot 1
#define death 2
#define quiet 3

const uint8_t startS[450] = {0x89,0x9A,0xBB,0xCE,0xFF … };
const uint8_t shootS[280] = { … };
const uint8_t deathS[8]   = {0x8B,0xDE,0xFE,0xDB,0x85,0x32,0x12,0x35};
const uint8_t quietS[1]   = {0x88};
struct sound{
      uint32_t length;          // number of bytes in the array
      const uint8_t *samples;   // pointer to the array
};
typedef struct sound Sound_t;

// sounds is the array of structs one per sound
Sound_t sounds[4] = {{450,startS},{280,shootS},{8,deathS},{1,quietS}};

uint32_t cSound; // holds the current sound number (0,1,2,or 3)

// Declare any other globals you need here
```

```
uint8_t hi;     // hi=1 means high byte or 0 to indicate low byte
uint32_t Index; // index into the sound array
```

Your task is to write the following three routines, along with any globals you need above:

```
// Setup SysTick so it interrupts periodically at 8 kHz, bus=80MHz
void SysTick_Init(void){
  NVIC_ST_CTRL_R = 0;
  NVIC_ST_RELOAD_R = 9999; // 80MHz/8kHz = 10000
  NVIC_ST_CURRENT_R = 0;
  cSound = quietS;  Index = 0; hi = 1;


  NVIC_SYS_PRI3_R = (NVIC_SYS_PRI3_R&0x00FFFFFF)|0x40000000;// Priority 2

  NVIC_ST_CTRL_R = 0x07; // CS=1, IEN=1, EN=1
}
```

```
// Sound is always playing. Call this function to change the sound
// Called with a single input which is 0-3 specifying which sound to play
// Start playing the new sound from the beginning after switching.
void ChangeSound(uint8_t soundNum){

      cSound = soundNum;
      Index = 0;
      NVIC_ST_CURRENT_R = 0;
      hi = 1;


}
```

```
// SysTick_Handler calls DAC_Out output one 4-bit value
// cSound specifies the sound to play
// loop current sound if the end is reached
// DAC_Out is given, you do not need to write it
void SysTickHandler(void){

    if(hi){
      DAC_Out((sounds[cSound]).samples[Index]>>4);
      hi = 0;
    }else{
      DAC_Out(sounds[cSound].samples[Index]&0x0F);
      hi = 1;
      Index++;  // increment every other output
    }
    if(Index==sounds[cSound].length){
      Index = 0;
    }

}
```

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`

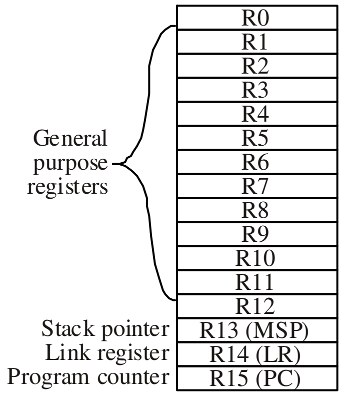
```
     value   any 32-bit value: signed, unsigned, or address
     {S}     if S is present, instruction will set condition codes
     #im12   any value from 0 to 4095
     #im16   any value from 0 to 65535
     {Rd,}   if Rd is present Rd is destination, otherwise Rn
     #n      any value from 0 to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

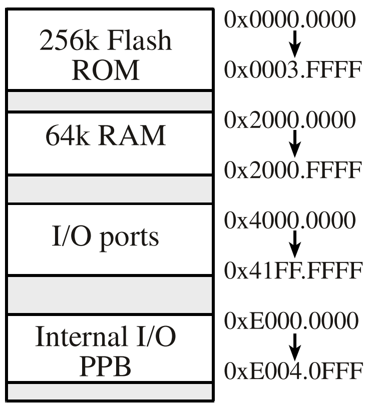
```
   ADD Rd, Rn, Rm         ; op2 = Rm
   ADD Rd, Rn, Rm, LSL #n ; op2 = Rm<<n  Rm is signed, unsigned
   ADD Rd, Rn, Rm, LSR #n ; op2 = Rm>>n  Rm is unsigned
   ADD Rd, Rn, Rm, ASR #n ; op2 = Rm>>n  Rm is signed
   ADD Rd, Rn, #constant  ; op2 = constant
```

, 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.E108			GPIOF	GPIOE	GPIOD	GPIOC	GPIOB	GPIOA	SYSCTL_RCGCGPIO_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 TM4C123/LM4F120 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 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

	31-12	11-0	
$4003.80A8		12-bit DATA	ADC_SSFIFO3

Table 10.3. The TM4C123/LM4F120ADC 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 11 on the TM4C123/LM4F120. 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.