Final Exam

Date: December 11, 2014

UT EID: ________________________ Circle one: ME, JV, RY

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	20	
Problem 2	10	
Problem 3	10	
Problem 4	10	
Problem 5	10	
Problem 6	10	
Problem 7	20	
Problem 8	10	
Total	100	

(20) Question 1 (Miscellaneous)

(3) Part a) This *Interrupt Service Routine* modifies registers R0 and R1 but the ISR does not save R0 and R1, why?

```
SysTick_Handler
    LDR R1, =GPIO_PORTF_DATA_R
    LDR R0, [R1]
    EOR R0, R0, #0x04
    STR R0, [R1]
    BX  LR
```

(5) Part b) A DAC is used to output a *sine wave* using SysTick Interrupts and a sine-wave table. Assume the DAC has 7 bits, the DAC output is connected to a speaker, the SysTick ISR executes at 32kHz, the sine table has 256 elements, and one DAC output occurs each interrupt. The DAC output range is 0 to 3.3V. The bus clock is 80 MHz. The ADC maximum rate is 125 kHz. What frequency sound is produced, in Hz?

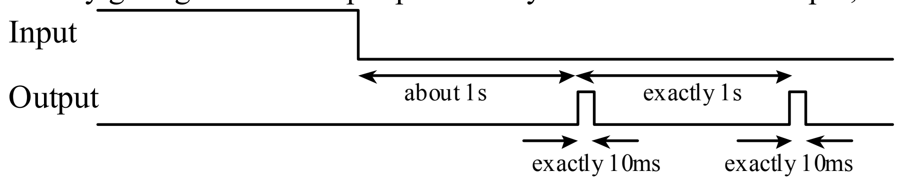
(3) Part c) A DAC has a *range* of 0 to 3V and needs a *resolution* of 1mV. How many bits are required? In other words, what is the *smallest number of DAC bits* that would satisfy the requirements?

(3) Part d) An embedded system will use an ADC to capture electrocardiogram (EKG) data. The frequency range of the human EKG spans from 0.1 Hz to 100 Hz. What is the *slowest rate* at which we could sample the ADC and still have a faithful representation of the EKG in the digital samples? Give your answer as the time between samples. (*Hint: this is not the Valvano Postulate*.)

(3) Part e) An 8-bit ADC (different from the TM4C123) has an input range of 0 to +10 volts and an output range of 0 to 255. What *digital value* will be returned when an input of +7.5 volts is sampled? Give your answer as a decimal number.

(3) Part f) A serial port (UART1) is configured with one start, 8 data bits, one stop and a baud rate of 50,000 bits/sec. What is the *maximum possible bandwidth* of this port in bytes/sec?

(10) Question 2 (FSM). You will design a pacemaker using a Moore FSM. There is one input and one output. The input will be high if the heart is beating on its own. The input will be low if the heart is not beating on its own. If the heart is not beating your machine should pace the heart. If the heart is beating on its own, the input will be high and your output should be low. However, if the input is low, you should pace the heart by giving a 10 ms output pulse every 1000 ms. PB0 is output, PB1 is input.



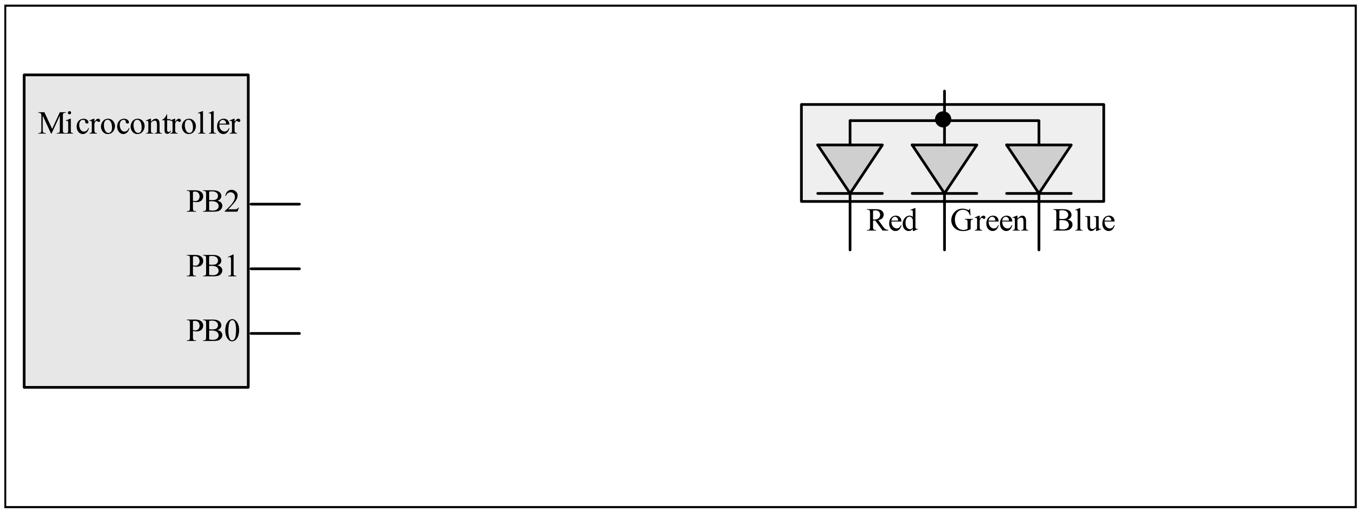
(5) Part a) Show the FSM graph in Moore format. Full credit for the solution with the fewest states.

(5) Part b) The structure and the main program are fixed. Show the C code that places the FSM in ROM, and specify the initial state in the box.

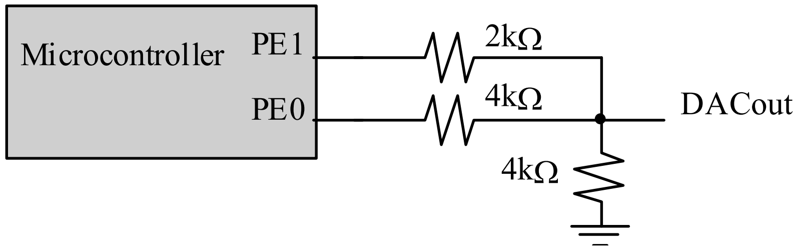
```
const struct State{
      uint32_t out;
      uint32_t wait;
      uint32_t next[2];
};
typedef const struct State State_t;
uint32_t s;
```

```
void main(void){ PORTB_Init();
  SysTick_Init();
  s =                 ;
  while(1){
    GPIO_PORTB_DATA_R = FSM[s].out;
    SysTick_Wait1ms(FSM[s].wait);
    Input = (GPIO_PORTB_DATA_R&0x02)>>1;
    s = FSM[s].next[Input];
  }}
```

(10) Question 3. Interface a multicolor LED to the microcontroller. Each color is controlled by a separate diode with an operating point of 2V, 25mA. You can use any number of 7406 inverters, and any number of resistors. Assume the V_{OL} of the 7406 is 0.5V. Assume the microcontroller output voltages are V_{OH} = 3.0V and V_{OL} = 0.1V. Specify values for any resistors needed. Show equations of your calculations used to select resistor values. Make each output control one color, positive logic.



(10) Question 4 (DAC). What are the maximum voltage, precision, and resolution of this DAC? Assume the microcontroller output voltages are V_{OH} = 3.2V and V_{OL} = 0.0V.



Maximum voltage	Precision	Resolution

(10) Question 5 (UART).

(5) Part a) Write two C functions that send an 8-byte message using UART1. Assume the UART1 is already initialized for busy-wait synchronization. The 7 bytes of payload are passed by reference to your function. You will send an 8th byte that is an error-checking code (ECC), which will be the bit-wise **exclusive or** of the 7-byte data.

ECC = str[0] ^ str[1] ^ str[2] ^ str[3] ^ str[4] ^ str[5] ^ str[6]

Your UART1_OutMessage function should call your UART1_ OutChar function.

```
// Input: str is a pointer to a 7-byte array of data to be transmitted
void UART1_OutMessage(const uint8_t str[7]){









// Input: 8-bit data to be transmitted
void UART1_OutChar(const uint8_t data){
```

(5) Part b) Assume you have received the 8-byte message from the UART1 on the other microcontroller, and the message has been placed in this array of 8 bytes:

```
Message SPACE 8
```

Write an *assembly* subroutine that checks the ECC to determine if an error has occurred. Return R0=0 if the message is ok, and return R0≠0 if the ECC does not match. The subroutine will access the global array called **`Message`**. Hint: what should the following calculation be if there is no error?

Message[0]^Message[1]^Message [2]^Message [3]^Message [4]^Message [5]^Message [6]^Message[7]

```
CheckMessage
```

(10) Question 6 (debugging). Consider the FIFO code, which declares one global data structure and implements two functions that manipulate the structure. The compiler will initialize all variables to 0. Note that the code has many bugs.

(8) Part a) Write down as many bugs as you can find and for each bug propose a solution. Use the boxes below to describe each bug, the lines affected by it (possibly multiple lines with same type of bug), and a solution. If a bug is missing lines of code, mark down the two line numbers between which your solution code needs to be inserted and just write the extra code in the "Solution" column. For example, lines 6 and 11 are missing a semi-colon as marked below.

```
1:  #include <stdint.h>
2:  struct fifo {
3:    char    data[512];
4:    uint8_t x, y, z;};
5:  typedef struct fifo fifo_t;
6:  fifo_t myData
7:  int8_t Fifo_Put(char c) {
8:    if (myData.z == 512) {
9:      return(-1);
10:   }
11:   myData.data[myData.x] = c
12:   myData.x = [myData.x + 1] % 511;
13:   myData.z = myData.z + 1;
14: }
15: int8_t Fifo_Get(char* c) {
16:   c = myData ->data[myData->y];
17:   myData->y = (myData->y + 1) % 511;
18:   myData->z = myData->z + 1;
19: }
```

Line(s)	Description	Solution
6, 11	Missing ;	Add ;

Lines(s)	Description	Solution

(2) Part b) What is the purpose of the return value of the function **`Fifo_Put()`** and the function **`Fifo_Get()`**?

(20) Question 7 (Local Variables, AAPCS and Parameter-passing)
Answer the questions that follow with reference to the code given below. Assume initially that R3=3, R4=4, R5=5, R11=11, and LR = 0x30F.

```
// Main.c
extern uint32_t Func(                                ); //[1]

int main(void){
  uint16_t glob;
  uint32_t param = 14;

  glob=42;
  param = Func(param,&glob);    //[A]
  glob += param;                //[C]
}
```

```
main PUSH  {r3-r5,lr}
     MOVS  r4,#0x0E
     MOVS  r0,#0x2A
     STR   r0,[sp,#0x00]
     MOV   r1,sp
     MOV   r0,r4
     BL    Func
     MOV   r4,r0
     LDRH  r0,[sp,#0x00]
     ADD   r0,r0,r4
     UXTH  r0,r0 ;clear 31-16
     STR   r0,[sp,#0x00]
     MOVS  r0,#0x00
     POP   {r3-r5,pc}
```

```
; Func.s
     EXPORT                ;[2]
     AREA    |.text|, CODE, READONLY, ALIGN=2
     THUMB

loc equ                      ; Binding [3]

Func
    SUB  SP,#4
    PUSH {R2,R11}
    LDRH R2,[R1]
    ADD  R2,#1
    STRH R2,[R1]
    STR  R2,[SP,#loc]
    ADD  R0,R2
    STR  R0,[SP,#loc]
    POP  {R2,R11}              ;[B]
    ADD  SP,#4
    BX   LR

    ALIGN
    END
```

a) (5 points) Compete the three missing blanks in lines labeled [1], [2] and [3].
b) (2 points) Circle the **Allocation** and **Deallocation** steps for the local variable **`loc`**.
c) (3 points) Assuming the SP is initialized to `0x20000400`. What are the contents of the Stack (and the value of the SP) after **`main`** calls **`Func`** and the instruction at [B] has been completed.

0x200003E4	
0x200003E8	
0x200003EC	
0x200003F0	
0x200003F4	
0x200003F8	
0x200003FC	
0x20000400	
0x20000004	

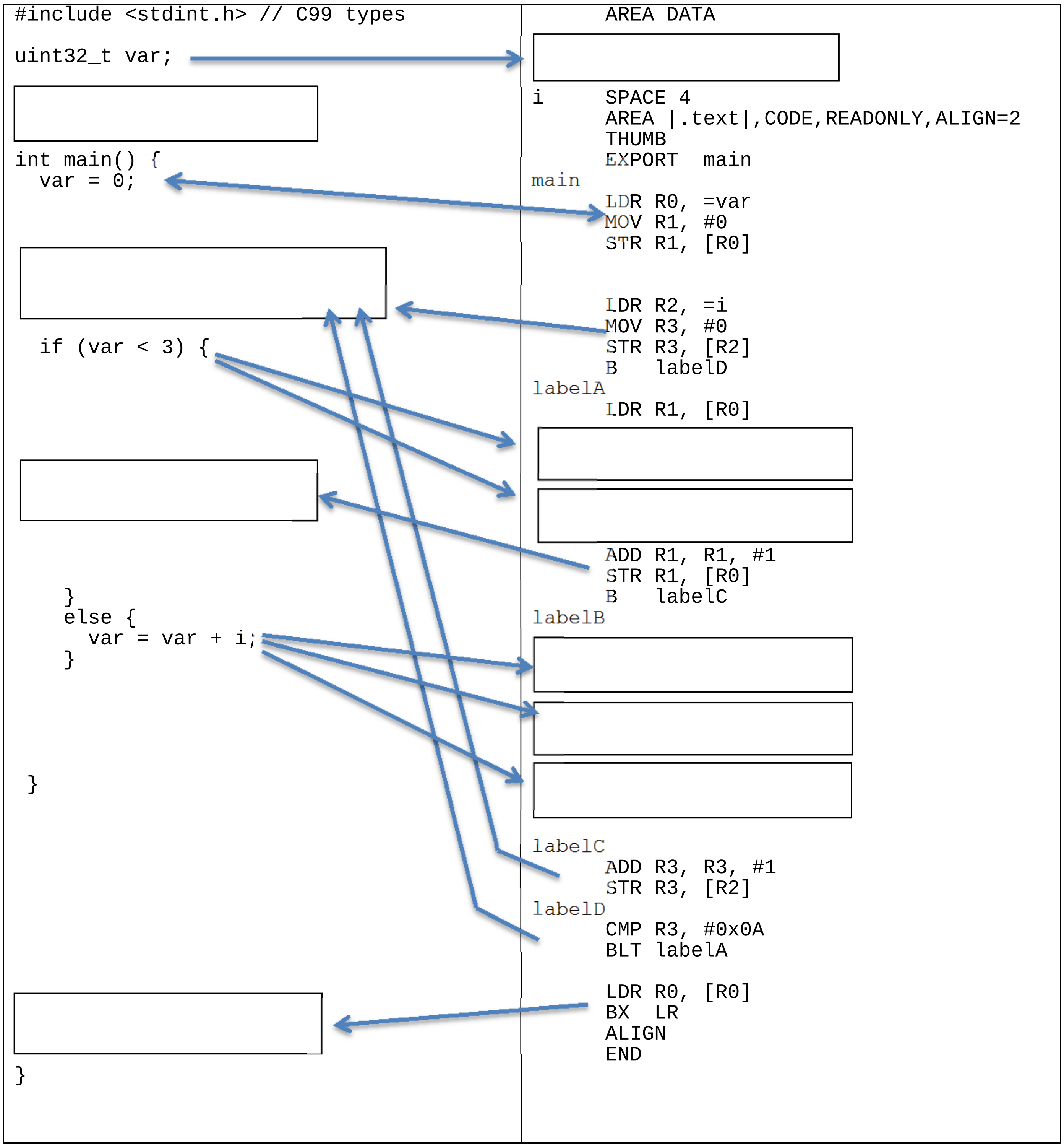
d) (5 points) What is the value of the variable **`glob`**:

I. After instruction at [A] is completed?

II. After instruction at [C] is completed?

e) (5 points) Give the C equivalent of the assembly code corresponding to the subroutine **`Func`**.

(10) Question 8: **(assembly/C)** The left and right sides represent corresponding C and ARM assembly (think of the assembly as the compiler-produced code of the C part). Both sides contain a few missing lines, which you are to fill in. Each box below represents exactly one missing line of code (in either C or assembly). Note arrows are placed to roughly correspond to lines of assembly and lines of C.



```
#include <stdint.h> // C99 types

uint32_t var;



int main() {
  var = 0;




  if (var < 3) {




    }
    else {
      var = var + i;
    }




 }




}
```

```
       AREA DATA

i      SPACE 4
       AREA |.text|,CODE,READONLY,ALIGN=2
       THUMB
       EXPORT  main
main
       LDR R0, =var
       MOV R1, #0
       STR R1, [R0]

       LDR R2, =i
       MOV R3, #0
       STR R3, [R2]
       B   labelD
labelA
       LDR R1, [R0]


       ADD R1, R1, #1
       STR R1, [R0]
       B   labelC
labelB



labelC
       ADD R3, R3, #1
       STR R3, [R2]
labelD
       CMP R3, #0x0A
       BLT labelA

       LDR R0, [R0]
       BX  LR
       ALIGN
       END
```

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`

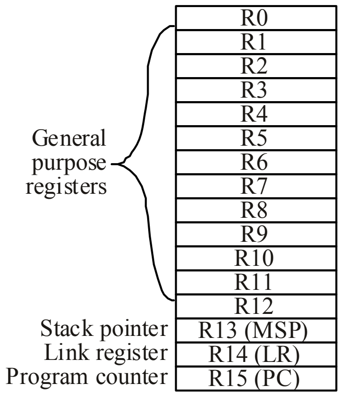
```
     value    any 32-bit value: signed, unsigned, or address
     {S}      if S is present, instruction will set condition codes
     #im12    any value from 0 to 4095
     #im16    any value from 0 to 65535
     {Rd,}    if Rd is present Rd is destination, otherwise Rn
     #n       any value from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

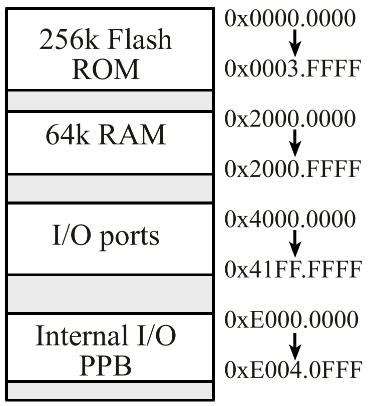
```
   ADD Rd, Rn, Rm         ; op2 = Rm
   ADD Rd, Rn, Rm, LSL #n ; op2 = Rm<<n  Rm is signed, unsigned
   ADD Rd, Rn, Rm, LSR #n ; op2 = Rm>>n  Rm is unsigned
   ADD Rd, Rn, Rm, ASR #n ; op2 = Rm>>n  Rm is signed
   ADD Rd, Rn, #constant  ; op2 = constant
```

, 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.