The University of Texas at Austin
Department of Electrical and Computer Engineering

Introduction to Embedded Systems

EE319K (Gerstlauer), Spring 2013

Final Exam

Date: May 11, 2013

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:

Signature: _____

Instructions:

- Closed book and closed notes.
- No calculators or any electronic devices (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.*
- For all questions, unless otherwise stated, find the most efficient (time, resources) solution.

Problem 1	15	
Problem 2	25	
Problem 3	20	
Problem 4	15	
Problem 5	25	
Total	100	

Problem 1 (15 points): Subroutines

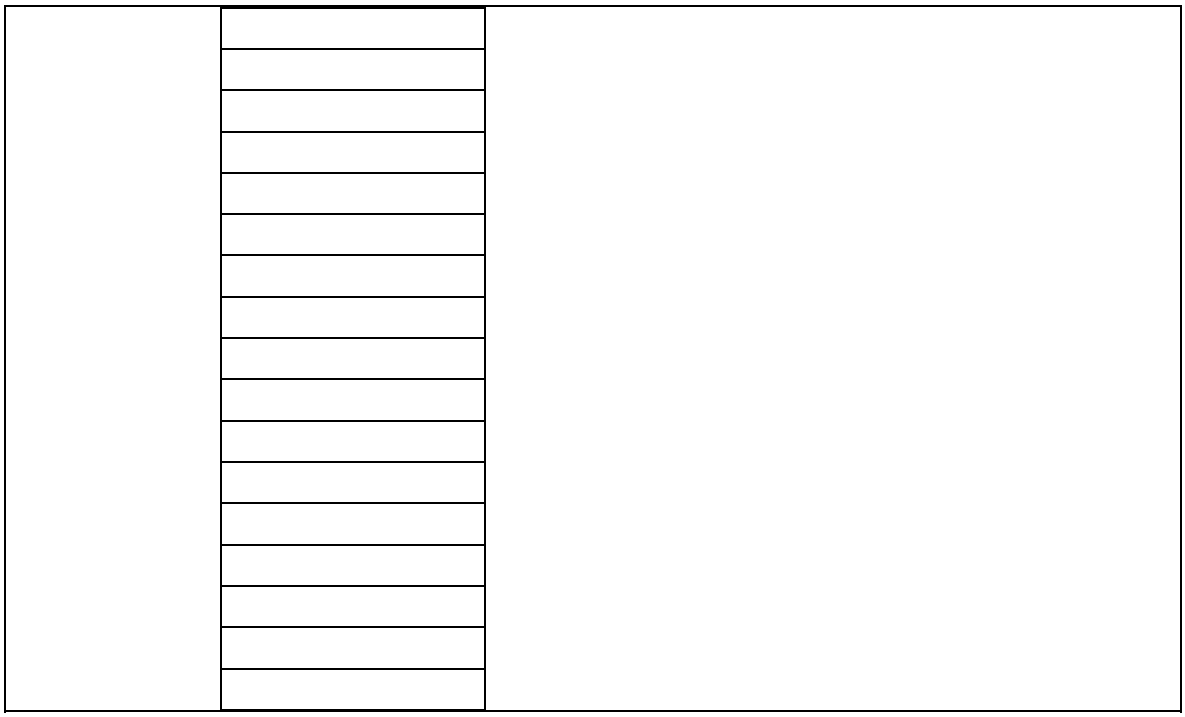
Given the following C functions:

```
long max1(long a[2]) {
  long m;
  m = a[0];
  if (a[1] > m) m = a[1];
  return m;
}

void max2(long a, long b) {
  long data[2];
  data[0] = a; data[1] = b;
  return max1(data);
}
```

(a) Assume a calling convention in which only the first parameter of a function is passed via register R0. All other parameters are passed via the stack and functions can freely use registers R0-R3. Complete the partial assembly code generated for these two functions:

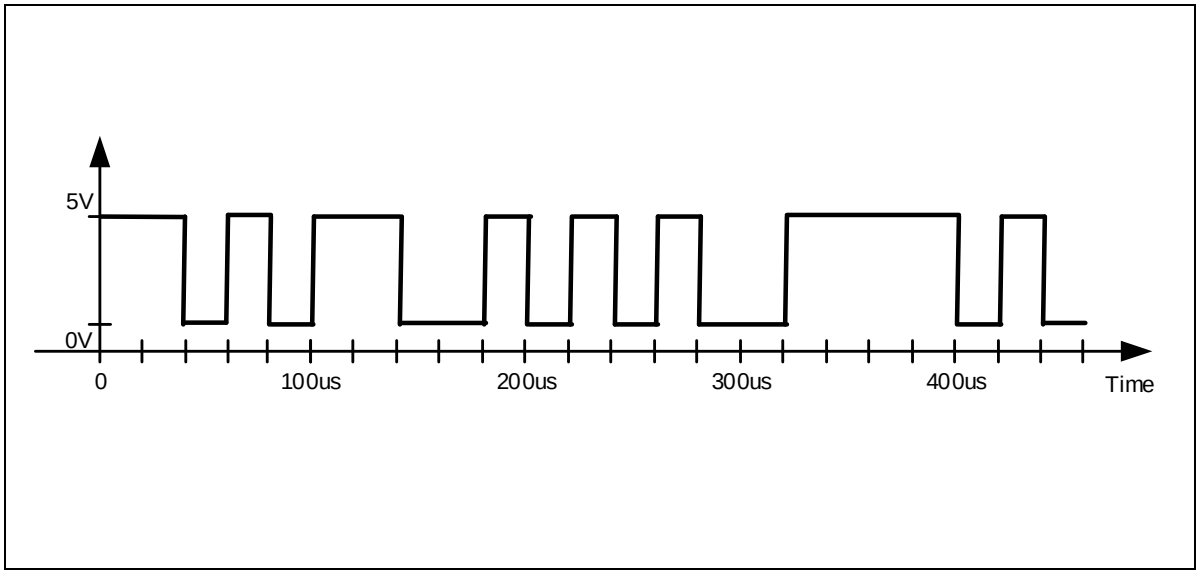
```
     AREA    |.text|, CODE, READONLY, ALIGN=2

max1
     LDR       R1,[R0]
     LDR       R2,[R0,_________]
     CMP       R1,R2
     _______   n1
     MOV       R1,R2
n1   MOV       R0,R1
     BX        LR

b    EQU  __________          ; input parameter 'b'
dat0 EQU  __________          ; local variable 'data[0]'
dat1 EQU  __________          ; local variable 'data[1]'

max2
     _________________________ ; save registers
     _________________________ ; allocate 'data'
     MOV       R11,SP
     STR       R0,[R11,#dat0] ; store first 'data' element
     LDR       R0,[R11,#c]    ; load 'b'
     STR       R0,[R11,#dat1] ; store second 'data' element
     ADD       R0,R11,#dat0   ; pass address to 'data'
     BL        max1
     _________________________ ; de-allocate 'data'

     _________________________ ; restore registers
     BX        LR
```

(b) Assume that *max2* is called with parameters *max2(42,-14)*. Show the contents of the stack frame of *max2* at the point right before the `BL max1` instruction gets executed. Mark any allocated but uninitialized stack items with a '?'. Indicate the location of both stack and frame pointer. Each entry below corresponds to a 32-bit word.



Problem 2 (25 points): Serial Communication

(a) Assume you are observing the following waveform on an oscilloscope attached to a serial communication line. Assuming that the line was idle before, mark the frame boundaries and indicate the start, stop and data bits within each observed frame



(b) What is the baud rate and bandwidth of the observed communication channel?

Baud rate	Bandwidth (bytes/s)

(c) Assuming that ASCII characters are transmitted over the line, what partial message have we observed so far?

(d) Assume that you want to implement a receiver that taps into the communication line to observe and decode transmitted messages. Given the code template for the UART initialization function below, fill in the blanks to complete the initialization code such that it matches communication requirements and enables the receive FIFO with interrupts on ¼ full and when idle. Assume that the systems is running at an 8MHz bus clock.

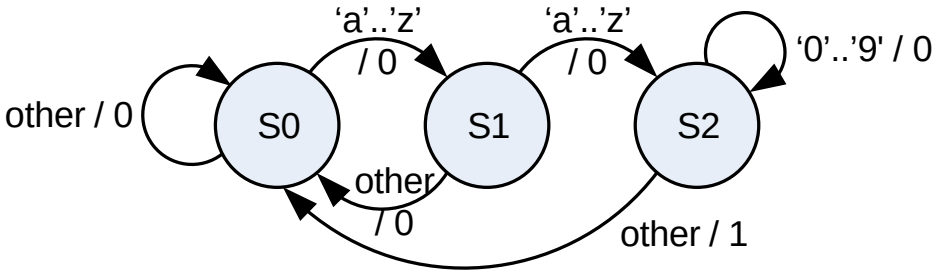
```
void UART0_Init(void) {
  SYSCTL_RCGC1_R |= 0x0001;
  SYSCTL_RCGC2_R |= 0x0001;
  UART0_CTL_R &= ~0x0001;
  UART0_IBRD_R _____________________
  UART0_FBRD_R _____________________
  UART0_LCRH_R = 0x0070;
  UART0_IFLS_R &= ~0x38;
  UART0_IFLS_R |= 0x80;
  UART0_IM_R  ______________________
  UART0_CTL_R ______________________
  GPIO_PORTA_AFSEL_R |= 0x03;
  GPIO_PORTA_DEN_R |= 0x03;
  NVIC_PRI1_R = (NVIC_PRI1_R&0xFFFF00FF)|0x00004000;
  NVIC_EN0_R |= NVIC_EN0_INT5;
}
```

(e) Write the code for the UART interrupt handler. The handler is supposed to read ASCII characters from the UART and put them into a global software FIFO until the UART receive FIFO is empty. You can ignore errors (full conditions) of the software FIFO.

```
void UART0_Handler(void) {









        FIFO_Put(data);









}
```

Problem 3 (20 points): Finite State Machine

You are asked to implement a Mealy FSM that recognizes a certain pattern of ASCII characters received over the serial port (see Problem 2). The FSM processes characters by reading from the software FIFO at the beginning of each state. It outputs a '1' for one character duration every time the pattern is received in the input stream. The output of the FSM should be written to an LED attached to PA0. Given the following state diagram:



(a) What input pattern does the FSM recognize?

(b) Define the C state structure to use in the FSM as well as the FSM array to encode the given machine. Hint: there is a naïve and a smart way to encode input dependencies.

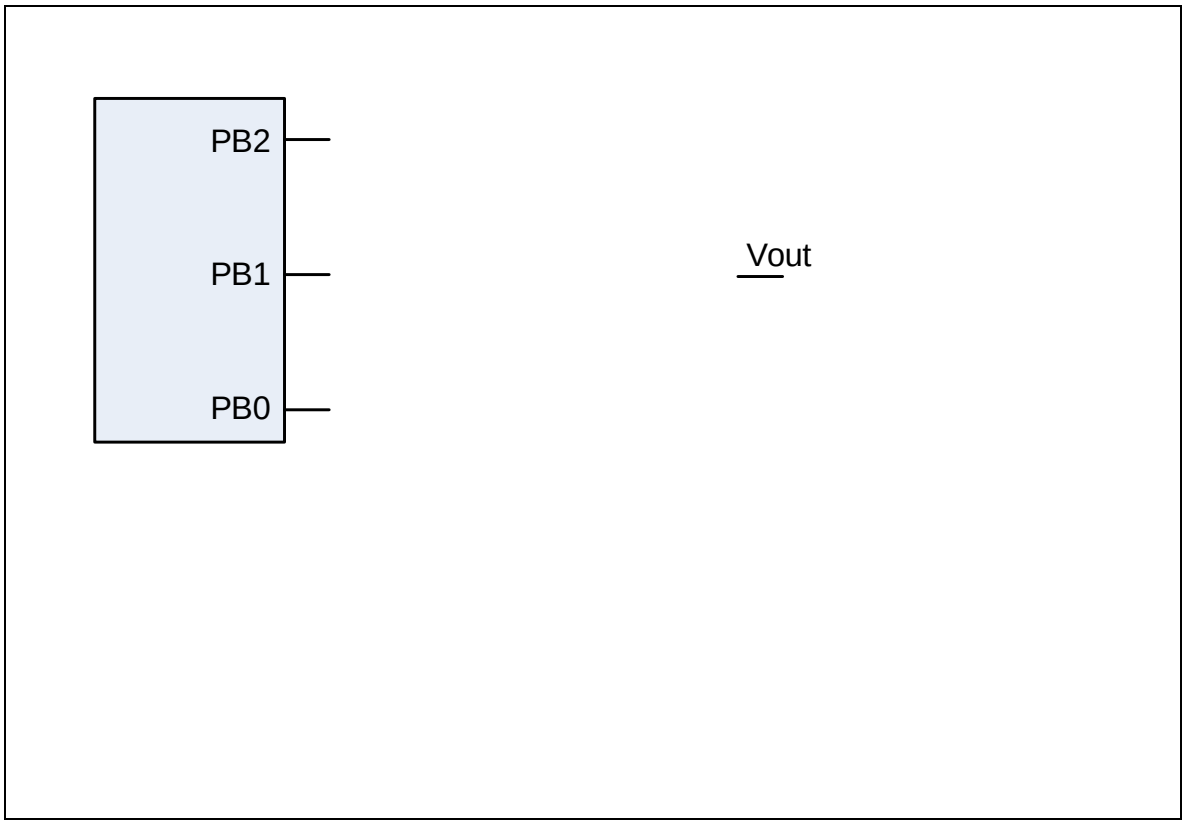
```
struct state {




};

typedef const struct state_stateType;

#define S0 _____________________
#define S1 _____________________
#define S2 _____________________

stateType FSM[3] = {




};
```

(c) Fill in the blanks to write the main program that implements the recognizer.

```
#define PA0 (*((volatile unsigned long *)0x40004004))

stateType *current = S0;

void main(void) {



   PLL_Init();
   UART0_Init();
   PortA_Init();

   while(1) {
      FIFO_Get(&input);




   }
}
```

Problem 4 (15 points): Digital to Analog Conversion

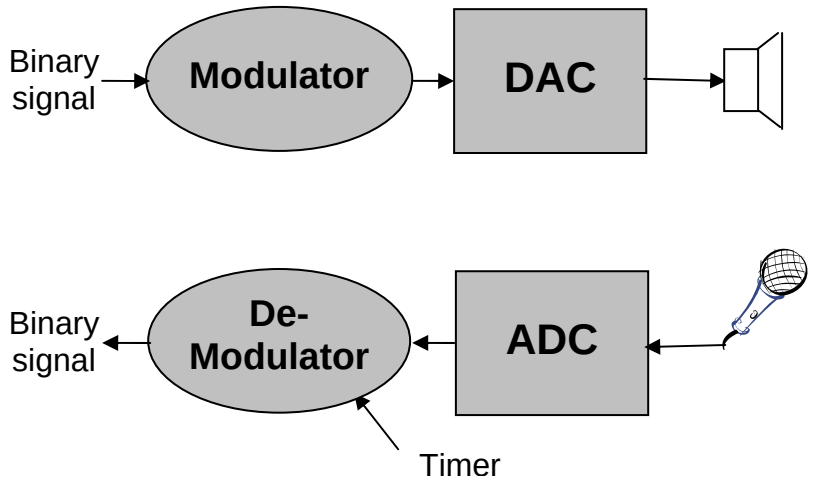
(a) Design a 3-bit binary-weighted DAC interfaced to PB2 through PB0 using 3kΩ, 6kΩ and 12kΩ resistors. What is the output voltage Vout when applying bit patterns 111 and 101?



(b) What is the voltage at Vout for the bit pattern 111 if a load of 12kΩ is applied between Vout and ground?

Problem 6 (25 points): Communication System

You are asked to implement the receiver side of a communication system that uses sound to wirelessly transmit a binary signal. The signal is transmitted by modulating a sound wave at 1000Hz such that the sound is on when the binary digit being transmitted is one and off otherwise (so-called Binary Amplitude Shift Keying, BASK modulation).



(a) Assume that the microphone can convert air pressure changes of sound waves into a voltage between 0 and 3V with a precision corresponding to only 4 bits. What is the resolution of the system and what 4-bit value will get sampled for a microphone voltage of 2.25V?

Resolution (V)	Sampled value at 2.25V

(b) What is the minimum rate at which the receiver needs to sample the sound signal in order to be able to properly decode it?

(c) In order to improve decoding and demodulation, we want to oversample the received sound signal at a rate of 32000Hz. Given the template below for a system running at a bus clock of 8MHz, fill in the blanks to complete the SysTick initialization routine to set the SysTick interrupt priority to 2 and trigger SysTick interrupts at the desired sampling rate. Make sure your code is friendly.

```
SysTick_Init
    LDR R1, =NVIC_ST_CTRL_R
    MOV R0, #0
    STR R0, [R1]
    LDR R1, =NVIC_ST_RELOAD_R


    ____________________________________
    STR R0, [R1]
    LDR R1, =NVIC_SYS_PRI3_R
    LDR R0, [R1]


    ____________________________________
    STR R0, [R1]
    LDR R1, =NVIC_ST_CTRL_R


    ____________________________________
    STR R0, [R1]
    BX  LR
```

(d) Finally, write the *SysTick_Handler* that calls an *ADC_In* routine to acquire 4-bit samples and collect it into a global *Samples* array. Every time 32 new samples have been collected, the handler is supposed to call a *Demodulate* function and put a zero or one into a global mailbox depending on whether the *Demodulate* result was greater than the predefined THRESHOLD. *ADC_In* and *Demodulate* functions are declared externally as follows:

ADC.h:

```
// Acquire 4-bit sample
unsigned char ADC_In(void);
```

Demodulate.h:

```
// Demodulate signal of 32 samples into amplitude
unsigned short Demodulate(unsigned char signal[32]);
```

Make sure that your code defines and uses the following variables:

- All mailbox-related variables should be public and permanently allocated.
- *Samples* should be a permanently allocated array that is private to the *SysTick.c* file.
- *Cur* should be a permanently allocated variable that is private to the *SysTick_Handler*. It should be initialized to zero and used to index into the *Samples* array.
- Any other necessary variables should be temporary and private to the handler.

SysTick.h

```
void SysTick_Handler(void);
```

SysTick.c

```
#define THRESHOLD  6000

#include “ADC.h”
#include “Demodulate.h”
#include “SysTick.h”

void SysTick_Handler(void)
{

}
```

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 requal,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
```

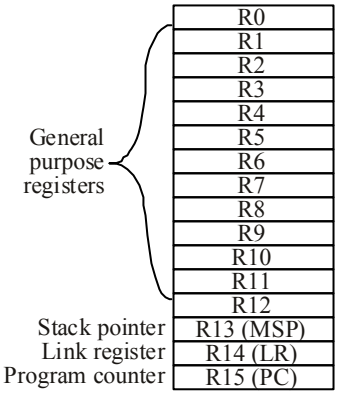
Notes `Ra Rd Rm Rn Rt represent 32-bit registers`

```
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>
```

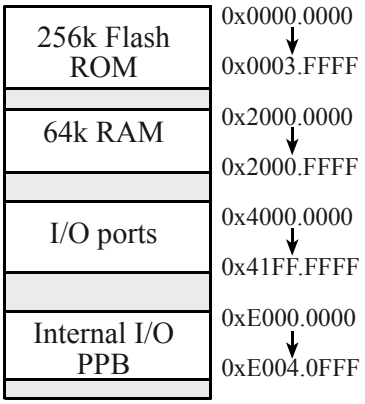
Examples of flexible operand **<op2>** creating the 32-bit number. E.g., **Rd = Rn+op2**

```
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.

ASCII Table

BITS 4 to 6	0	1	2	3	4	5	6	7
	0	NUL	DLE	SP	0	@	P	`	p
B	1	SOH	DC1	!	1	A	Q	a	q
I	2	STX	DC2	"	2	B	R	b	r
T	3	ETX	DC3	#	3	C	S	c	s
S	4	EOT	DC4	$	4	D	T	d	t
	5	ENQ	NAK	%	5	E	U	e	u
0	6	ACK	SYN	&	6	F	V	f	v
	7	BEL	ETB	'	7	G	W	g	w
T	8	BS	CAN	(	8	H	X	h	x
O	9	HT	EM	)	9	I	Y	i	y
	A	LF	SUB	*	:	J	Z	j	z
3	B	VT	ESC	+	;	K	[	k	{
	C	FF	FS	,	<	L	\	l	;
	D	CR	GS	-	=	M	]	m	}
	E	SO	RS	.	>	N	^	n	~