Final Exam

Date: December 15, 2017

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:

- ***Write your UT EID on all pages (at the top) and circle your instructor's name at the bottom.***
- 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	15	
Problem 5	15	
Problem 6	15	
Problem 7	15	
Problem 8	10	
Total	100	

(10) Question 1. Place one letter in the box for each question that represents the best answer.

A) To force the compiler to not optimize the access
B) To place the variable in nonvolatile ROM
C) To force the variable to be placed in a register
D) To place the variable in volatile RAM
E) To make the variable private to the file
F) To make the variable private to the function
G) Because of Arm Architecture Procedure Call Standard
H) To force the variable to be placed on the stack

(1) Part a) Why do we add `static` to an otherwise local variable? ..... D

(1) Part b) Why do we add `const` to an otherwise global variable? ..... B

(1) Part c) Why do we add `static` to an otherwise global variable? ..... E

(1) Part d) Why do we use a local variable? ..... F

I) To execute instructions faster
J) Because of the Central Limit Theorem
K) To reduce the latency of other interrupts
L) To save power, making the battery last longer
M) To prevent Aliasing
N) Because of the Nyquist Theorem
O) To decouple the execution of the ISR with the main program
P) To reduce noise and improve signal to noise ratio
Q) To synchronize one computer to another
R) To improve bandwidth

(1) Part e) Why do we place a FIFO queue between an ISR that reads data from an input port..... O
and the main program that processes the data?

(1) Part f) Why does a Harvard architecture have two (or more) buses? I

(1) Part g) Why should the time to execute an ISR be as short as possible? ..... K

(1) Part h) Why would you ever wish to use the PLL and slow down the bus clock L
so the software runs slower?

(1) Part i) Why would we use hardware averaging on the ADC? ..... P

(1) Part j) Why does the UART protocol use start and stop bits? ..... Q

(10) Question 2: For parts a) to f), please place one **numerical value** in the box for each question.

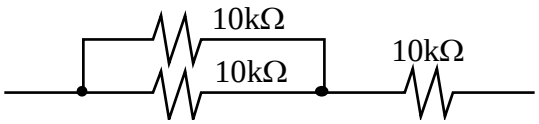
(2) Part a) You are recording a bioelectric signal with frequency components from 1 to 10 Hz. The ADC precision is 8 bits. What is the slowest sampling rate possible? …… 20 Hz

(1) Part b) What is the 16-bit hexadecimal representation of decimal 1000? …… 0x03E8

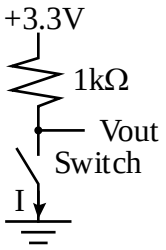
(1) Part c) UART0_IBRD_R equals 2 and UART0_FBRD_R equals 32. The bus cycle is 16 MHz? The UART is using its hardware FIFO. What is the baud rate? …… 400,000 bps

(1) Part d) A fixed-point number system has a resolution of 0.25 cm. What integer do we use to represent -3.75 cm? …… -15

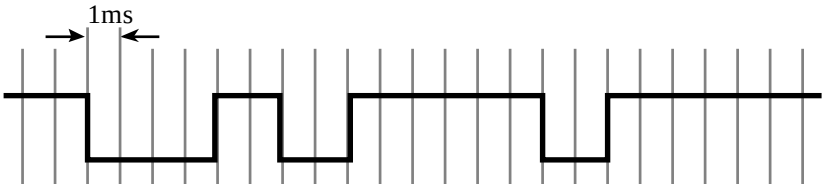
(1) Part e) These three resistors could be replaced by one resistor. Give the resistance value of this one equivalent resistor? …… 15kΩ



(1) Part f) How much current (I) flows when the switch is closed, include units …… 3.3 mA



Reverse-engineer UART parameters from the trace observed at a receiver below.



(2) Part g) What is the *data value* transferred over the UART in **hexadecimal**? …… 10111010_2 = 0xBA =186

(2) Part h) What is the *baud rate* in **bits/sec**? …… Bit time = 2ms, BR=500 bps

(10) Question 3: *Interrupt*

(6) Part a) Complete the assembly subroutine that initializes SysTick to interrupt every 1 sec. The bus clock is 16 MHz; that each bus cycle is 62.5 ns. Set the interrupt priority to 3. ARM and enable interrupts. The goal is to toggle PB1 every 3600 seconds. You may assume PB1 is already initialized to be a GPIO output. Fill in the blanks as needed

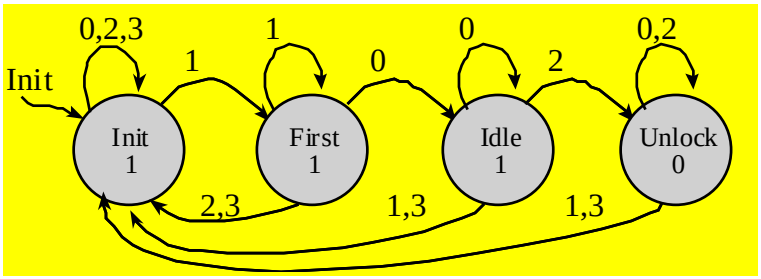
```
        THUMB
        AREA    DATA, ALIGN=4
Count SPACE 4

        AREA    |.text|, CODE, READONLY, ALIGN=2
Init    LDR R1,=NVIC_ST_RELOAD_R
        LDR R0,=15999999        ; 16,000,000 clock/sec

        STR R0,[R1]             ; establish interrupt period
        LDR R1,=NVIC_SYS_PRI3_R
        LDR R2,[R1]
        AND R2,R2,#0x00FFFFFF   ; priority is
        ORR R2,R2,#0x60000000   ;in bits 31-29

        STR R2,[R1]             ; set SysTick to priority 3
        LDR R1,=NVIC_ST_CTRL_R
        MOV R2,#7

        STR R2,[R1]             ; arm and enable SysTick
        LDR R1,=Count
        MOV R2,#3600
        STR R2,[R1]             ; count=3600
        CPSIE  I                ; enable interrupts




        BX  LR
```

(4) Part b) Write the SysTick ISR in assembly that toggles PB1 every 3600 seconds.

```
SysTick_Handler
     LDR  R1,=Count
     LDR  R2,[R1]
     SUBS R2,R2,#1
     BNE  skip
     LDR  R0,=GPIO_PORTB_DATA_R
     LDR  R2,[R0]
     EOR  R2,R2,#0x02    ; toggle PB1
     STR  R2,[R0]
     MOV  R2,#3600
Skip STR  R2,[R1]        ; set count
     BX   LR
```

(15) Question 4: *FSM*

(5) Part a) Consider a Moore finite state machine with 2 inputs (SW1, SW0) and one output (LOCK). Initially the LOCK is high (1). Any time both inputs are simultaneously high, the machine reverts to the initial state with the LOCK high. Call this initial state **Init**. The sequence just SW0 high (input=1), both switches low (input=0), and then just SW1 high (input=2) will cause the LOCK to go low (0). Assume the FSM runs at a fixed rate of 100 Hz periodically performing the usual output, wait 10ms, input, next operations. Use pointer addressing to access the next state.



(10) Part b) Show the C code, including the struct that defines your finite state graph in ROM. Define a state pointer **Pt**.

```
struct state {
  uint32_t Out;
  const struct state *Next[4];
};
typedef const struct State state_t;
state_t *Pt;
#define Init   &FSM[0]
#define First  &FSM[1]
#define Idle   &FSM[2]
#define Unlock &FSM[3]
state_t FSM[4]={
 {1, {  Init, First,   Init, Init}},  // Init state
 {1, {  Idle, First,   Init, Init}},  // First state
 {1, {  Idle,  Init, Unlock, Init}},  // Idle state
 {0, {Unlock,  Init, Unlock, Init}}   // Unlock state
};
```

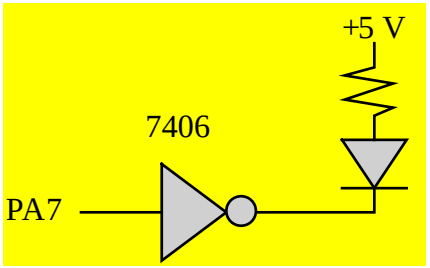
The execution engine in the main program is given and cannot be changed.

```
int main(void){uint32_t input;
  Port_Init();    // Port B and Port E initializations are given (not to write)
  SysTick_Init(); // SysTick initialization is given
  Pt = Init;      // initial state (not to write)
  while(1){
    GPIO_PORTE_DATA_R = Pt->Out;      // set LOCK to 0 or 1
    SysTick_Wait10ms(1);              // fixed delay
    input = GPIO_PORTB_DATA_R&0x03;   // read switches: 0,1,2,3
    Pt = Pt->Next[input];  }}}
```

(15) Question 5: *Interfacing*

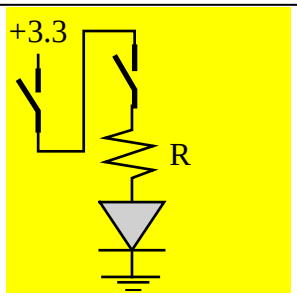
(5) Part a) Interface an LED to the microcontroller PA7 output. If PA7 is high the LED should be on. Assume the desired operating point is 3.5V 20mA. Assume the output low voltage of the TM4C123 is 0.2V. Assume the output high voltage of the TM4C123 is 3.1V. Assume the output low voltage of the 7406 is 0.5V. Show the circuit, including resistor values.

R = (5V-3.5-0.5V)/20 mA
= 1V/20mA = 50 ohms

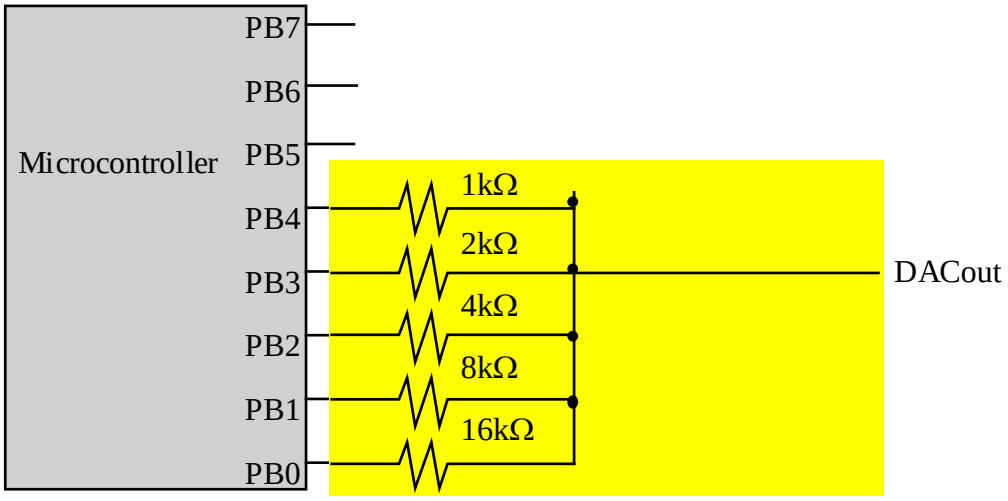


(5) Part b) Design a circuit with two switches and one LED. Assume the switches are ideal, and the LED operating point is 1.3V 2 mA. There is no microcontroller in this solution. The LED should be on if both switches are pressed; otherwise the LED should be off. Show the circuit, including the switches, the LED, and resistors as needed (include resistance values).

R = (3.3V-1.3V)/2mA = 1000 ohms



(5) Part c) Design a 5-bit DAC using multiple resistors. Show the circuit, including resistor values.



(15) Question 6: *FIFO queue* You are asked to implement a 16-bit FIFO that can handle up to 19 elements. You cannot add additional global variables. You cannot change the function prototypes. You must use pointers to access the FIFO

```
uint16_t FIFO[20];
uint16_t *Gpt,*Ppt;
```

(3) Part a) Write the routine that initializes the FIFO.

```
void Fifo_Init(void){ // Initialize FIFO
  Gpt = FIFO;
  Ppt = FIFO;
}
```

(6) Part b) Write the routine that puts data into the FIFO. If the FIFO is full, this routine should wait until there is room in FIFO for the data. You can add local variables, but no statics or globals.

```
void Fifo_Put(uint16_t data){ // enter data into FIFO
  uint16_t *pt = Ppt;
  pt++;
  if(pt == &FIFO[20]){
    pt = FIFO; // wrap
  }
  while(pt == Gpt){}; // full, so wait
  *Ppt = data;
  Ppt = pt;
}
```

(6) Part c) Write the routine that gets data from the FIFO. If the FIFO is empty, this routine should wait until there is data in FIFO to return. You can add local variables, but no statics or globals.

```
uint16_t Fifo_Get(void){ // if empty spin until there is data
  uint16_t data;
  while(Ppt == Gpt){}; // empty, so wait
  data = *Gpt; // remove data
  Gpt ++;
  if(Gpt == &FIFO[20]){
    Gpt = FIFO; // wrap
  }
  return data;
}
```

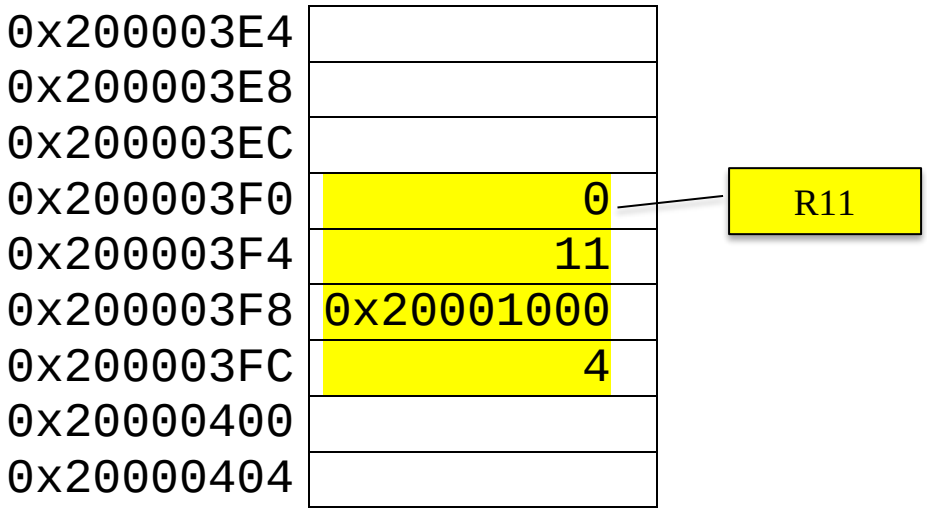
(15) Question 7: *Local variables*
(3) Part a) What does the function **func** do?
Calculates the exclusive or of all half-words in an array.

(7) Part b) Implement the operation performed by this assembly function in C. The assembly is written in AAPCS. Add C comments.

```
uint16_t Func(uint16_t *pt, uint8_t size){
  uint16_t result=0; // EOR of all data
  do{
    result ^= *pt; // access array
    pt++;          // pointer to next
    size--;        // number of elements
  }
  while(size);
  return result;
}
```

```
;R0 points to an array
;R1 is the size of array
Func PUSH {R0,R1}
     MOV  R2,#0
     PUSH {R2,R11}
     MOV  R11,SP
;***this point for part c)***
loop LDR  R0,[R11,#8]
     LDRH R1,[R0]
     ADD  R0,#2
     STR  R0,[R11,#8]
     LDRH R0,[R11,#0]
     EOR  R0,R0,R1
     STRH R0,[R11,#0]
     LDRB R0,[R11,#12]
     SUBS R0,#1
     STRB R0,[R11,#12]
     BNE  loop
     LDRH R0,[R11,#0]
     ADD  SP,#4
     POP  {R11}
     ADD  SP,#8
     BX   LR
```

(5) Part c) Assume at the beginning of the execution of the function **Func**, the stack is empty, SP equals 0x20000400, R0 equals 0x20001000, R1 equals 4, R2 equals 2, and R11 equals 11. Show the contents of the stack at the point in the execution of **Func** signified by the *****.** Also specify the value of R11 using a pointer into the stack.



(10) Question 8: *UART interrupt*

Assume you have the SSI interface to the LCD you used in Lab 6. In C, this LCD output data function is

```
void writedata(uint8_t c) {
  while((SSI0_SR_R&0x00000002)==0){};   // wait until transmit FIFO not full
  GPIO_PORTA_DATA_R |= 0x40;            // DC is data
  SSI0_DR_R = c;                        // data out
}
```

The overall goal of the communication is to transfer data from the UART0 receiver (PA0) to the LCD. Assume the UART0 is initialized for receiver interrupts. In particular, when there is data in the receiver FIFO, RXRIS is set and a UART interrupt is triggered.

(6) Part a) Show the UART ISR that reads data from the receiver, acknowledges the interrupt and sends the data to the LCD using the SSI port.

```
void UART0_Handler(void){
uint8_t data;
  data = UART0_DR_R; // read from UART receiver
  UART0_IC_R = 0x10; // acknowledge
  writedata(data);
}
```

(2) Part b) The hardware automatically pushes R0, R1, R2, R3, R12, LR, PC, and PSW on the stack when the interrupt is triggered. These registers are automatically popped at the end of the ISR by the BX LR instruction. Is the ISR code allowed to use the other registers R4-R11? Choose the best answer, placing A-F in the box.

A) No, since thee registers are not saved, using these registers would cause errors in the main program.
B) Yes, R4-R11 are reserved for interrupts, so they can be freely used.
C) Yes, according to AAPCS any function can use R4-R11 if it first saves the registers and then restores them afterward.
D) No, R4-R11 are reserved for the main program, so they cannot be used in an ISR.
E) No, according to AAPCS the stack must be aligned to 8 bytes.
F) Yes, R4-R11 are also automatically saved by the compiler for every function.

C

(1) Part c) Does the hardware automatically disable interrupts during the execution of the ISR? Choose the best answer, placing A-F in the box.

F

A) Yes, the execution of interrupts is more important than the execution of the main program.
B) Yes, disabling interrupts prevents the execution of one ISR from being interrupt by itself
C) Yes, according to AAPCS, the hardware automatically disables interrupts during the execution of the ISRs.
D) No, but according to AAPCS, the software automatically disables interrupts during the execution of the ISRs.
E) No, interrupts are not disabled to allow an interrupt with a lower priority (higher value in priority register) to interrupt.
F) No, interrupts are not disabled to allow an interrupt with a higher priority (lower value in priority register) to interrupt.

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
```

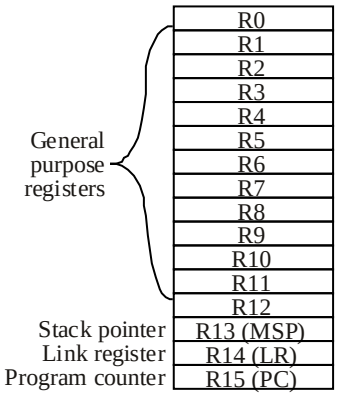
Notes `Ra Rd Rm Rn Rt represent 32-bit registers`

`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>

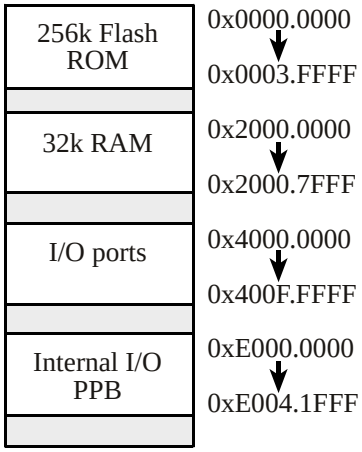
Examples of flexible operand **<op2>** creating the 32-bit number. E.g., **Rd = Rn+op2**

```
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.E608			GPIOF	GPIOE	GPIOD	GPIOC	GPIOB	GPIOA	SYSCTL_RCGCGPIO_R
$4000.53FC	DATA	DATA	DATA	DATA	DATA	DATA	DATA	DATA	GPIO_PORTB_DATA_R
$4000.5400	DIR	DIR	DIR	DIR	DIR	DIR	DIR	DIR	GPIO_PORTB_DIR_R
$4000.5420	SEL	SEL	SEL	SEL	SEL	SEL	SEL	SEL	GPIO_PORTB_AFSEL_R
$4000.551C	DEN	DEN	DEN	DEN	DEN	DEN	DEN	DEN	GPIO_PORTB_DEN_R

Table 4.5. TM4C123 Port B 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. We set **INTEN** to arm SysTick interrupts. The standard name for the SysTick ISR is **SysTick_Handler**.

Address	31-2	1	0	Name
$400F.E638		ADC1	ADC0	SYSCTL_RCGCADC_R

	31-14	13-12	11-10	9-8	7-6	5-4	3-2	1-0	
$4003.8020		SS3		SS2		SS1		SS0	ADC0_SSPRI_R

	31-16	15-12	11-8	7-4	3-0	
$4003.8014		EM3	EM2	EM1	EM0	ADC0_EMUX_R

	31-4	3	2	1	0	
$4003.8000		ASEN3	ASEN2	ASEN1	ASEN0	ADC0_ACTSS_R
$4003.80A0		MUX0				ADC0_SSMUX3_R
$4003.80A4		TS0	IE0	END0	D0	ADC0_SSCTL3_R
$4003.8028		SS3	SS2	SS1	SS0	ADC0_PSSI_R
$4003.8004		INR3	INR2	INR1	INR0	ADC0_RIS_R
$4003.8008		MASK3	MASK2	MASK1	MASK0	ADC0_IM_R
$4003.8FC4		Speed				ADC0_PC_R

	31-12	11-0	
$4003.80A8		DATA	ADC0_SSFIFO3_R

Table 10.3. The TM4C ADC registers. Each register is 32 bits wide. LM3S has 10-bit data.

Set Speed to 0001 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_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.