Final Exam

Date: December 11, 2013

UT EID: ________________________ Circle one: Gerstlauer or Valvano+Yerraballi

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 books, no papers, no data sheets (other than the last two 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.*

Problem 1	10	
Problem 2	10	
Problem 3	15	
Problem 4	10	
Problem 5	10	
Problem 6	15	
Problem 7	10	
Problem 8	10	
Problem 9	10	
Total	100	

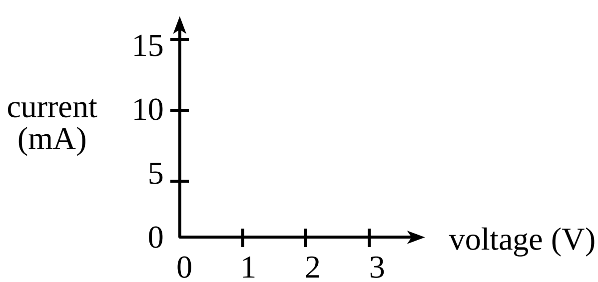
(10) Question 1 (Equations/relations you should know).

a) Consider a UART with one start bit, 1 stop bit, *n* data bits and no parity bits. The bus frequency is 80 MHz. Give the relationship between baud rate (*BR*) and maximum possible bandwidth (*BW*), assuming both are in bits/sec.

b) Consider a resistor used to build your DAC in Lab 6. Avogadro's Number is about $6.022*10^{23}$. Give the relationship between resistance (*R*, in kΩ), voltage (*V* in volts), and current (*I* in mA).

c) Consider the sampling rate chosen for the ADC in Lab 8. Give the relationship for the slowest possible sampling rate (f_s, in Hz), given these parameters: ADC resolution (ΔV, in volts), number of ADC bits (*n*, in bits, e.g., 12 bits) and rate at which one moves the slide pot (*r*, in oscillations per sec).

d) Sketch the current versus voltage curve of an LED like the ones used in lab. Include the (2V,10mA) operating point, but roughing sketch the other points from 0 to 3 volts



e) Consider a two-dimensional array of half words (16 bits each), with *n* rows and *m* columns. The bus frequency is 80 MHz. The base address of the array is *b*, and the array is defined in row major order What is the address of the element in row *i* and column *j*?

(10) Question 2 (Local Variables). The assembly subroutine below uses ***three*** local variables. Demonstrate your understanding of local variables in assembly by answering the following questions. You may assume the initial stack pointer is 0x20001008, no registers other than R0-R3 are used, and all three local variables are allocated on the stack.

Assembly	C Equivalent

`xxx equ aa`
`yyy equ bb`
`zzz equ cc`
`Locals`
`      ; Body of subroutine`
 	
`      BX LR`
	`void Locals(void) {`
	`    long xxx;`
	`    long yyy;`
	`    long zzz;`
	`// Body of subroutine`
	
	`}`

a) (2 points) Which of the following is ***not*** relevant to the use of local variables?

- **i.** Binding using equ pseudo-ops
- **ii.** Allocation on stack
- **iii.** Parameter-passing to the subroutine
- **iv.** Indexed access of the stack with SP as the base.
- **v.** Deallocation by restoring the SP

b) (2 points) What will the value of the SP be after allocating space for all three?

c) (3 points) Assuming that the three variables are allocated space in the order in which they appear **xxx** at higher address, **yyy** in the middle and **zzz** at the lower address. Lower address means smaller value than higher address. The values of **aa**, **bb** and **cc** are:

- **i.** **aa** is 8; **bb** is 4; **cc** is 0
- **ii.** **aa** is 0; **bb** is 1; **cc** is 2
- **iii.** **aa** is 2; **bb** is 1; **cc** is 0
- **iv.** **aa** is 0; **bb** is 4; **cc** is 8
- **v.** **aa** is 1; **bb** is 2; **cc** is 4
- **vi.** None of the above

d) (3 points) Assuming that the correct values for **bb** is set. Which of the sequences of instructions will add 1 to the local variable **yyy**.

- **i.** `LDRSB R0,[SP,#yyy]; ADD R0,R0,#1;  STRB R0,[SP,#yyy]`
- **ii.** `LDR   R0,[SP,yyy];  ADD R0,R0,#1;  STR  R0,[SP,yyy]`
- **iii.** `ADD   [SP,#yyy],#1;`
- **iv.** `LDRSH R0,[SP,yyy]; ADD R0,R0,#1;  STRH R0,[SP,yyy]`
- **v.** None of the above

(15) Question 3 (C Programming with `struct`). Given the following `struct` declaration for a student, complete the subroutine which (a) calculates each student's grade as 'P' or 'F' depending on whether the score is higher than or equal to 75, and (b) returns the average class score;

```
#define SIZE 64
struct Student {
   unsigned long id;
   unsigned long score;
   unsigned char grade; // you will enter 'P' or 'F'
};
typedef struct Student STyp;

unsigned long Grades(STyp class[SIZE]){

}
```

(10) Question 4 (Interrupts).

a) (3 points) An Interrupt Service Routine executes the last line of its code, a return statement (`BX LR`). Which of the following registers are ***not*** popped from the stack? Put all the letters in the box that apply. For example, if you think i,ii are not popped, but iii,iv,v are popped, enter i+ii.

i. R0-R3
ii. R12
iii. LR
iv. PC, SP, PSR
v. R4-R11

b) (5 points) Assume the bus clock is operating at 80 MHz. The SysTick initialization executes these instructions. SysTick will be used to generate a periodic interrupt with an interrupt period of 100µs (which is 10 kHz.) What assembly instructions go in the **`????(a)????`** and **`????(b)????`** places?

```
SysTick_Init
    LDR R1,=NVIC_ST_RELOAD_R
                                        ????(a)????
    STR R0,[R1]
    LDR R1,=NVIC_ST_CTRL_R
                                        ????(b)????
    STR R2,[R1]
    BX  LR
```

c) (2 points) All Interrupt Service Routines with the exception of **`SysTick_Handler`** must do this:

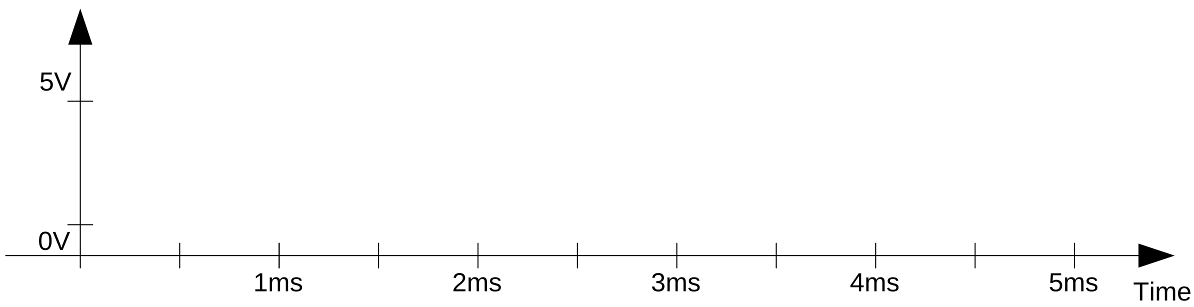
i. Explicitly pop the SP before returning from the interrupt
ii. Not use the Stack
iii. Write to a FIF0
iv. Explicitly acknowledge the Interrupt
v. Write to a mailbox
vi. None of the above

(10) Question 5 (UART).

a) (2 points) A programmer set the **`UART0_IBRD_R`** to 50 and **`UART0_FBRD_R`** to 0. If the Bus clock frequency is 80MHz, what is the baud rate? (bps stands for bits per second)

i. 80 kbps
ii. 100 kbps
iii. 1 Mbps
iv. 120 kbps
v. 16 kbps
vi. None of the above

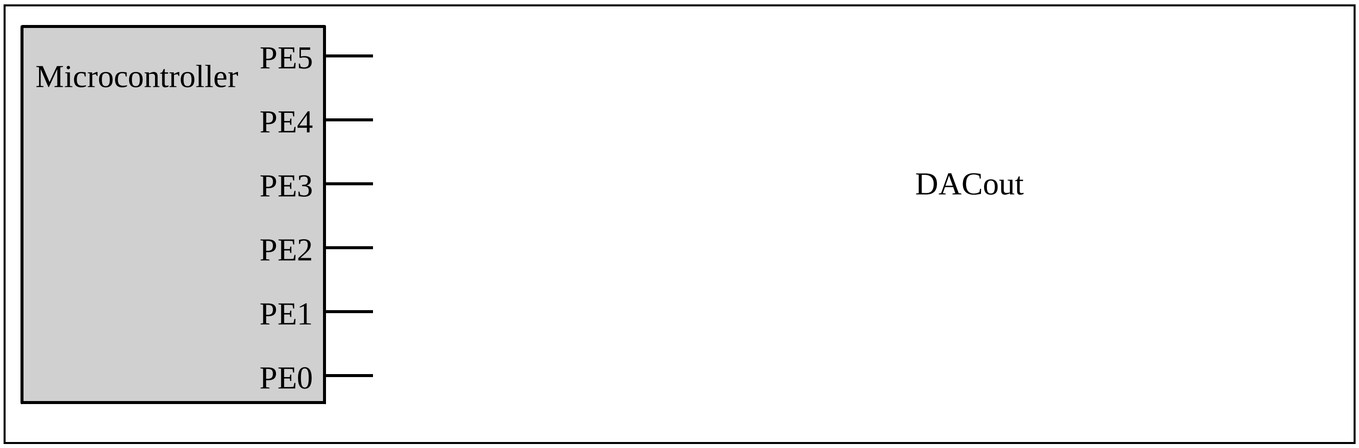
b) (5 points) Assuming now a serial port operating with a baud rate of 2000 bits per second. The protocol is 1 start, 8 data and 1 stop bit. Draw the waveform when the decimal value 204 is transmitted. You may assume the channel is idle before and after the frame. Time flows from left to right.



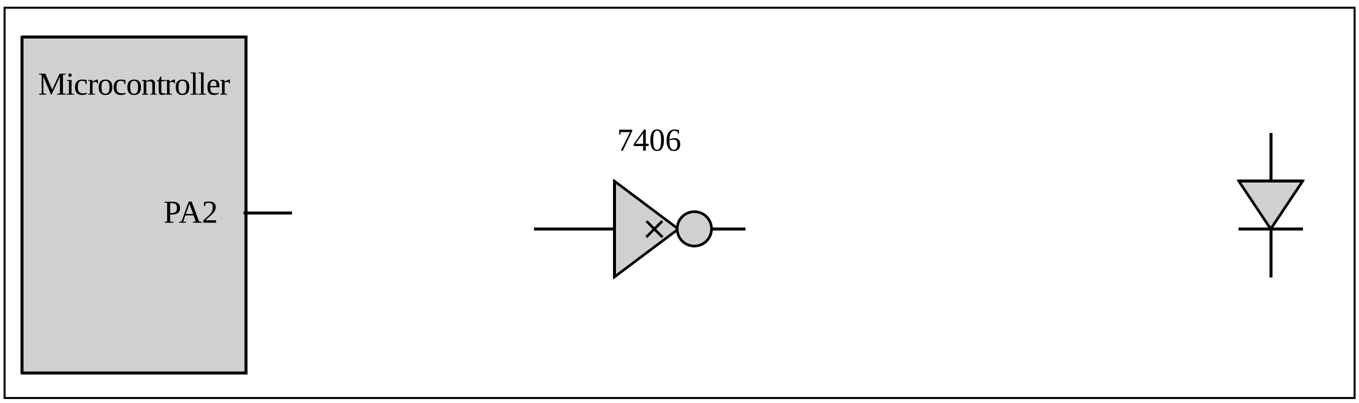
c) (3 points) Assume the serial port is setup as in part b) (2000 bps, 1 stop, 8 data, 1 start bit) and the serial receive interrupt is set to trigger when the UART receive FIFO is half full. Furthermore, the receive interrupt handler empties the UART FIFO every time it is invoked. How long can interrupts at most be disabled to guarantee that no UART overflow (OE bit set) will occur.

(15) Question 6. (Hardware)

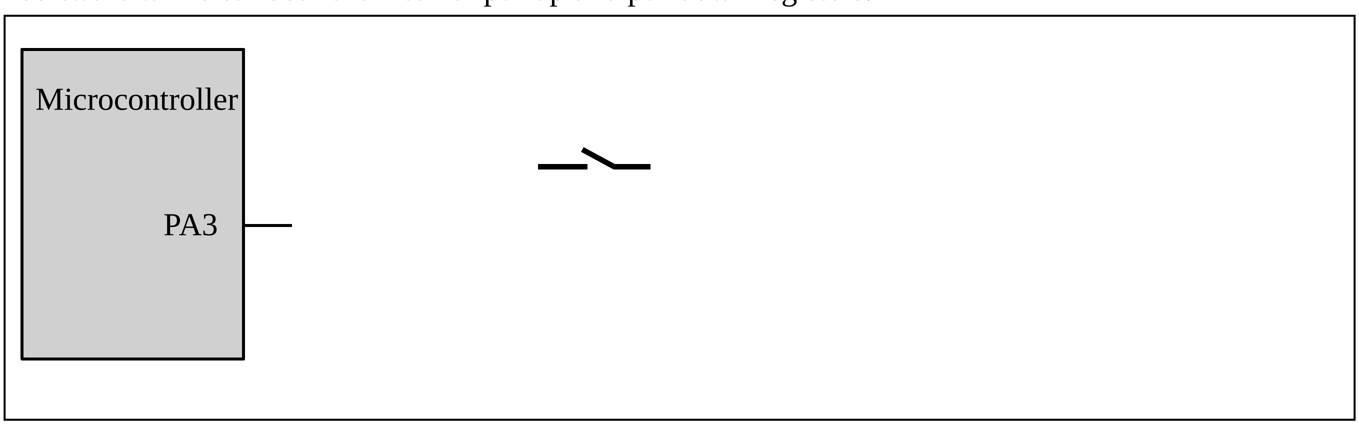
a) (5 points) Design a 6-bit DAC using the binary-weighted configuration. The DAC is controlled by six output port pins, PE5-0, where PE0 is the least significant and PE5 is the most significant bit. Carefully label the signal which is the DAC output, and specify the values for any resistors used



b) (5 points) The desired LED operating point is 1V, 1mA. Assume the V_{OL} of the 7406 is 0.5V. Assume the microcontroller output voltages are V_{OH} = 3.1V and V_{OL} = 0.2V. Interface this LED to PA2 using positive logic. Full credit for the solution uses the fewest components, and partial credit if it works. Specify values for any resistors needed. Show equations of your calculations used to select resistor values.



c) (5 points) Interface a switch to PA3 such that if the switch is pressed the software sees a logic 0 and if the switch is not pressed the software sees a logic one. Specify values for any resistors needed. The software will clear both the internal pullup and pulldown registers.



(10) Question 7 (ADC). Assume the ADC has already been initialized to use sequencer 3 with a software trigger and channel 1. Write a C function that starts the ADC, waits for it to complete, reads the 12-bit result, clears the flag and returns the measured voltage (in the ADC's range of 0-3V) as a value with units of mV. For example if the input is 1.234V then the software will return 1234. The prototype for this function is

```
unsigned long ADC0_InSeq3(void);
```

(10) Question 8 (FIFO).

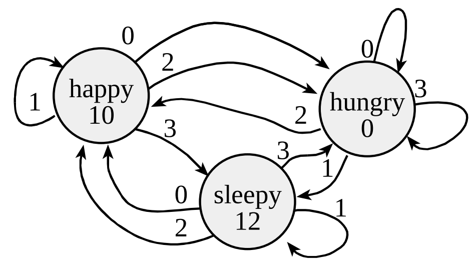
a) (2 points) Why is the **first in first out** (FIFO) queue really important for interfacing I/O devices?
i) They can store data permanently, which is important because embedded systems are used in safety critical situations, and we need to know what the data was during operation.
ii) They are a way to store data in the cloud. FIFOs provide backup and sharing.
iii) The software and hardware can operate at variable speeds and data are temporarily spooled into the FIFO as it passes between them.
iv) It can store an arbitrarily large amount of data. This is important because the size and complexity of embedded systems is growing.
v) None of the above.
vi) All of the above.

b) (5 points) Circle all the bugs in this FIFO implementation, and show the corrections needed to make this FIFO functional.

```
unsigned char static PutI;
unsigned char static GetI;
short static FIFO[16]; // 16 halfwords or 32 bytes of data
void Fifo_Init(void){
  PutI = GetI = 5;
}
int Fifo_Put(short data){
  if(((PutI+1)&0x1F) == GetI) return 0;
  FIFO[PutI] = data;
  PutI = (PutI+1)&0x1F;
  return 1;
}
int Fifo_Get(short *datapt){
  if(PutI == GetI) return 0;
  *datapt = FIFO[GetI];
  GetI = (GetI+1)&0x1F;
  return 1;
}
```

c) (3 points) Assuming a SysTick handler calls `Fifo_Put()` every 1 ms and the `main()` is able to process one item every 2 ms. If the `main()` is processing the first item right after the first SysTick puts it into the FIFO at time 0, does the FIFO ever overflow and if so, at what time?

(10) Question 9 (FSM).



a) (4 points) Assume we start in the happy state. The input starts and remains 3. What sequence of outputs will occur?

i) start and remain at 10

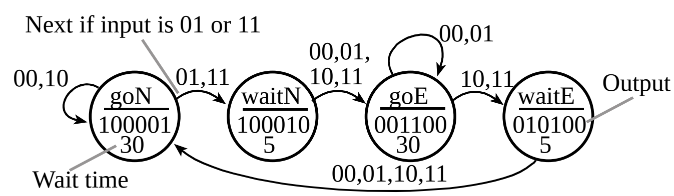
ii) 10, 12, 0 (and remain at 0)

iii) 10, 0, 10, 0, 10, 0, 10, 0, 10, over and over

iv) 10, 12, 10, 12, 10, 12, 10, 12, 10, over and over

v) None of the above

b) (6 points) Consider this FSM, the 6-bit output is on Port B (PB5-0) and the 2-bit input on Port E (PE1-0). You may call the SysTick function `SysTick_Wait10ms(n);` to wait *n**10msec.




```
struct State {
  unsigned long Out;
  unsigned long Time;
  const struct State *Next[4];};
typedef const struct State STyp;
#define goN   &FSM[0]
#define waitN &FSM[1]
#define goE   &FSM[2]
#define waitE &FSM[3]
STyp FSM[4]={
 {0x21,3000,{goN,waitN,goN,waitN}},
 {0x22, 500,{goE,goE,goE,goE}},
 {0x0C,3000,{goE,goE,waitE,waitE}},
 {0x14, 500,{goN,goN,goN,goN}}};
STyp *Pt;
int main(void){
  SysTick_Init(); // this function is given which initializes SysTick
  Port_Init();    // this function is given which initializes B E
  Pt = goN;
```

Write C code that completes this main program such that the FSM runs in the foreground. Write friendly code. You can assume that **`GPIO_PORTB_DATA_R`** and **`GPIO_PORTE_DATA_R`** have been defined.

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`

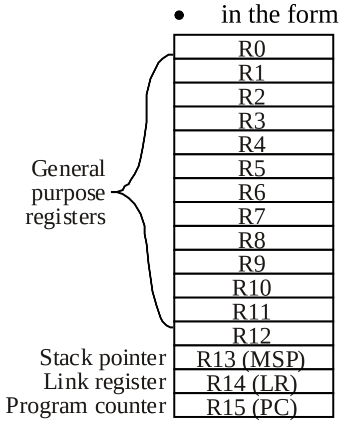
```
value   any 32-bit value: signed, unsigned, or address
{S}     if S is present, instruction will set condition codes
#im12   any value from 0 to 4095
#im16   any value from 0 to 65535
{Rd,}   if Rd is present Rd is destination, otherwise Rn
#n      any value from 0 to 31
#off    any value from -255 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
```

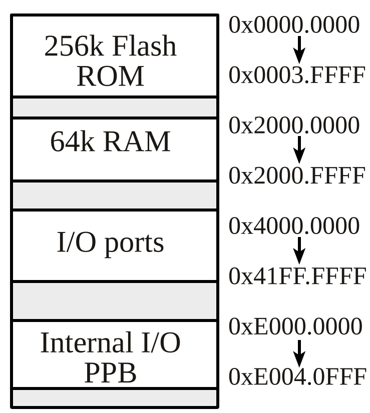
, where **X** and **Y** are hexadecimal digits:

- 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_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 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

	31-10	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.

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.