C2000 *Teaching Materials*

GETTING STARTED WITH THE TMS320F24x PROCESSOR

Tutorial 10: Storing Values in Buffers

New Instructions Introduced

`BANZ`
`DMOV`
`LTD`
`SPM`

New Flag Introduced

`PM`

Introduction

Within a digital signal processing (DSP) system, it is common practice to take readings using the ADC at regular intervals and to store the values in data memory. Normally a buffer is used to store the latest measurement and to hold a number of previous values.

Using a Buffer

A buffer is an area of data memory. It is used to store data that have been received or to hold data to be transmitted. A common example is the printer buffer of a computer. This contains formatted data to be sent to the printer. The printer can then take this data at its own speed. A buffer can be implemented C as follows:

Example 10-1.

```
int buffer[8] = {0,0,0,0,0,0,0,0};
```

In Example 10-1, the buffer is implemented as an *array* of eight words and has the name `buffer`. The elements within the buffer are numbered from 0 to 7. The first element is `buffer[0]`, the second element `buffer[1]` and the final element is `buffer[7]`. Here `{0,0,0,0,0,0,0,0}` assigns an initial value of zero to each element.

To implement Example 10-1 in assembly language we can write:

Example 10-2.

```
                .setsect ".data", 300h
                .data
buffer:         .word 0,0,0,0,0,0,0,0   ; No final comma
```

The assembler directive `.setsect` defines a segment in memory. Here we have a data segment (read/write) starting at address 300h. The label `buffer` provides us with a way to access the data elements.

Writing to a Particular Element of the Buffer

Example 10-3 shows how we would copy a value from the ADC to the fifth element of the buffer.

Example 10-3.

```
unsigned int buffer[8];     /* Array of 8 words */

buffer[4] = ADCFIFO1;       /* Copy value ADC reading to */
                            /* 5th element of the buffer */
```

To implement Example 10-3 in assembly language we can write:

Example 10-4.

```
                .setsect ".text",   8800h

                DP_ADC              .set 224
                DP_GP               .set 7      ; 380h to 3FFh.
                ADCFIFO1            .set 36
                buffer              .set 300h ; Buffer start address
                index               .set 0
                temp                .set 1      ; temporary storage

start:          LDP #DP_GP          ; Data page for GP variables.
                SPLK #4, index      ; Test purposes. Set index to 5th
                                    ; element.
                LACC #buffer        ; Accumulator contains start
                                    ; address of buffer.
                ADD index           ; Accumulator = 300h + index.
                SACL temp           ; Make copy in temp
                LAR AR3, temp       ; AR3 = 300h + index.
                MAR *, AR3          ; Use AR3 for instructions that
                                    ; use indirect addressing.

                LDP #DP_ADC         ; Page 224. Gain access to ADC
```

		; registers.
	LACC ADCFIFO1	; Copy the ADC measurement to ; the accumulator.
	SACL *	; Copy ADC measurement to data ; memory address = 300h + *index*
	B *start*	; Branch to label start.

In Example 10-4 we have used the `.set` directive to define the constant `BUFFER`, which has a value of 300h. This is the start address of the buffer array in data memory. To write to the buffer array, here we use indirect addressing with AR3 as the current auxiliary register. The value contained in AR3 is calculated from the sum of `BUFFER + index`.

When using a buffer, we would tend to use indirect addressing rather than direct addressing. There are two reasons for doing this. First, if the buffer were to contain more than 128 entries, it would straddle more than one data page. This makes the code more difficult to write and could lead to the wrong data page being worked upon.

Second, in many DSP applications there is a need to read each of the elements within the buffer. Indirect addressing is very convenient for this because we have the `*+` and `*-` operators, which are used to increment and decrement respectively the address

Using a Straight Buffer

One way in which to use a buffer to save a measurement is to put the newest measurement into one end of the buffer and the oldest at the other. For example, we may wish to use `buffer[0]` to hold the newest sample and `buffer[7]` to hold the oldest.

In order to insert a new measurement into the buffer, we discard the oldest reading then shuffle all the values along one place. For example, `buffer[6]` would be moved into `buffer[7]`. The process is shown in Figure 10-1 and assumes that the buffer already contains 1,2,3,4,5,6,7 and 8.

Figure 10-1. Storing Values in a Buffer

To carry out the operation shown in Figure 10-1 using the TMS320F24x, in C code we could write:

Example 10-5.

```
                                    
  unsigned int buffer[8];           /* Array of 8 elements */
  unsigned int x = 7;               /* General purpose variable  */
  do
  {
   x--;                             /* Decrement x. */
   buffer[x+1] = buffer[x];         /* Copy buffer[6] to */
                                    /* buffer[7] etc. */
  } while (x > 0);                  /* Terminate when x is zero */
                                    
  buffer[0] = ADCFIFO1;             /* Read in new value */
```

The `do-while` loop copies `buffer[6]` to `buffer[7]`, `buffer[5]` to `buffer[6]` etc. Note that in Example 10-3 we do not copy `buffer[7]` to `buffer[8]` - this would put it off the end of buffer and probably overwrite some other variable.

Also, we have to work from the highest to the lowest; otherwise we simply fill the buffer with all the same values.

In Example 10-5 a `do-while` construct has been deliberately used instead of a `for` loop or a `while` loop. This allows us to carry out an operation before terminating the loop at zero, which is an easy condition to implement using assembly language.

This may seem a rather complicated way to work, but does in fact suit the architecture of the TMS320F24x.

Using a Buffer in Assembly Language

The TMS320F24x provides the instruction Data Move in Data Memory (`DMOV`), which moves the contents of a data memory address to the next highest memory

location. We could apply the instruction DMOV multiple times, which is fine for a small buffer but impractical for a large one. In Example 10-6 we use a loop.

Example 10-6.

```
                    
        .setsect ".data",       300h ; Data memory.
        .data
buffer: .word 0,0,0,0           ; Fill buffer with zeroes.
        .word 0,0,0,0           ; No comma after numbers.
                    
        .setsect ".text",       8800h ; Executable code.
                    
DP_ADC  .set 224                ; Data Page for ADC.
ADCTRL1 .set 32h                ; ADC Control Register 1.
ADCTRL2 .set 34h                ; ADC Control Register 2.
ADCFIFO1 .set 36h               ; Result of ADC measurement.
                    
start:  LDP #DP_ADC             ; Data page for ADC.
        SPLK #0C01h, ADCTRL1    ; ADC channel 0.
        SPLK #0003h, ADCTRL2    ; Set conversion time.
        LAR AR4, #buffer+6      ; AR4 contains the address
                                ; of buffer[6].
        MAR *, AR4              ; Make AR4 the current
                                ; auxiliary register
        LAR AR5, #6             ; Set the number of repeats
                                ; to be used with the
                                ; instruction BANZ.
loop:   DMOV *-, AR5            ; Copy data to next higher
                                ; location and point to next
                                ; lower element using AR4 as
                                ; the pointer. Make AR5 the
                                ; current auxiliary
                                ; register.
        BANZ loop, *-, AR4      ; When AR5 is non-zero,
                                ; branch to the label loop.
                                ; Otherwise execute the
                                ; following instruction.
                                ; In either case, decrement
                                ; AR5 and make AR4 the
                                ; current auxiliary
                                ; register.
        MAR *+                  ; AR4 now points to 2FFh.
                                ; Increment to point to
                                ; first element of buffer
        LACC ADCFIFO1           ; Copy contents of ADC to
                                ; accumulator.
        SACL *                  ; Copy from accumulator to
                                ; buffer[0].
        B start                 ; Go round again.
```

In Example 10-6, the new instruction BANZ (Branch Auxiliary Register Non-Zero) has been introduced. This instruction is a code-efficient way in which to implement a loop counter. The first step is to assign one of the auxiliary registers AR0 to AR7 as the counter, and initialize it to the number of loops required.

The instruction BANZ first tests the current auxiliary register. If the current auxiliary register is non-zero, then a branch occurs to the label *loop*, otherwise program execution continues at the line below. The operator *- decrements the current auxiliary register. Finally, AR4 is made the current auxiliary register for use with the instruction DMOV.

The order of the three operands is: label, operation on the auxiliary register and optionally set a new current auxiliary register.

Relationship of Buffers to Z Transform

A straight buffer maps very closely to the Z transform.

Table 10-1.

Description	Where Stored in buffer	Equivalent Z value
Newest sample	buffer[0]	z^{0}
Second newest sample	buffer[1]	z^{-1}
Oldest sample	buffer[7]	z^{-2}

When the instruction DMOV is used to shift a value from say buffer[0] to buffer[1], it is delaying the newest sample.

Multiplication with Accumulation

A typical DSP application such as a finite impulse response (FIR) filter will multiply a series of input values by a series of constants. For eight elements, mathematically this can be represented by:

```
Output = a[0]*c[0] + a[1]*c[1] + a[2]*c[2] + a[3]*c[3] +
         a[4]*c[4] + a[5]*c[5] + a[6]*c[6] + a[7]*c[7]
```

where a[0] to a[7] are the input samples and c[0] to c[7] are the corresponding constants.

To implement this in C code we can write:

Example 10-7.

int a[8] = {0,0,0,0,0,0,0,0};	/* Inputs */
int c[8] = {1,2,3,4,5,6,7,8};	/* Constants */
int x = 8;	/* Counter */

```
long output = 0;                    /* Output */

 do
{
  x--;                              /* Decrement x */
  output +=  a[x] * c[x];
} while ( x > 0);
```

The `do-while` loop multiplies each element in the array `a[8]` by the corresponding constant in the array `c[8]`. Note the counter starts at a high value and counts downwards.

To implement Example 10-7 in assembly language, we would write:

Example 10-8.

```
                    .setsect ".data", 300h      ; Data memory.
                    .data
array:              .word 1, 2, 3, 4            ; Fill buffer with
                    .word 5, 6, 7, 8            ; known values.

constants:          .word 1, 2, 3, 4            ; Array of constants.
                    .word 5, 6, 7, 8

                    .setsect ".text",8800h      ; Executable code.
                    .text                       ; Necessary

start:              LACC #0                     ; Clear accumulator
                    LAR AR2, #6                 ; Loop counter
                    LAR AR0, #array+7           ; Point to last input
                    LAR AR1, #constants+7       ; Point to last constant
                    MAR *, AR0                  ; Make AR0 the current
                                                ; auxiliary register.

                    LT *-, AR1                  ; Load buffer[7]
                    MPY *-, AR0                 ; Multiply by constant
                    PAC                         ; Copy P to accumulator

loop:               LT *-, AR1                  ; Load the T register
                                                ; with buffer
                    MPY *-, AR2                 ; Multiply by constant
                    APAC                        ; Add product to the
                                                ; accumulator.
                    BANZ loop, *-, AR0          ; Test AR2 for zero, and
                                                ; if so, branch to the
                                                ; label loop. If not
                                                ; zero, execute the next
```

		; instruction. Make AR0 ; the current auxiliary ; register.
	B *start*	; Go round again.

When the BANZ loop terminates, the accumulator will contain:

8*8 + 7*7 + 6*6 + 5*5 + 4*4 + 3*3 + 2*2 + 1*1 = 204 decimal (CCh).

For clarity in Examples 10-7 and 10-8, the data move for each memory location was not shown. We shall now re-write a fragment of Example 10-8 to show how the instruction DMOV would be used

Example 10-9.

	LT *-, AR1	; Load buffer[7]
	MPY *-, AR0	; Multiply by constant
		; No needed DMOV here.
loop:	APAC	; Add previous product to ; the accumulator.
	LT *	; Load the T register ; with buffer content.
	DMOV *-, AR1	; Copy to next.
	MPY *-, AR2	; Multiply by constant.
	BANZ *loop*, *-, AR0	; Test AR2 for zero, and ; if so, branch to the ; label *loop*. If not ; zero, execute the next ; instruction. Decrement ; AR2. Make AR0 ; the current auxiliary ; register.
	APAC	; Accumulate final value.

When the T Register has been loaded, the contents of the buffer[] location is moved to the next highest element.

In this case we have also altered the order in which the instructions occur. Now the accumulations in the loop are done before the multiplications. The product of the first multiplication is accumulated near the label loop, rather than by using the instruction PAC, which has been done away with.

Combining the Instruction DMOV with Other Instructions

In practice, the instruction DMOV would not be used often on its own. This is because it can be combined with other instructions such as LTD i.e. Load T Register, Accumulate previous product and Move Data.

In fact, the following three instructions can be condensed into a single instruction:

Example 10-10.

```
        APAC            ; Add contents of P register to
                        ; accumulator.
        LT *            ; Load the T register with the
                        ; contents of a data memory
                        ; address
        DMOV *-         ; Copy contents of dma to the
                        ; next higher data memory
                        ; address. Decrement the pointer
```

In Example 10-10, the three instructions are the equivalent to the instruction `LTD`. The following instruction would be an instruction such as Multiply (`MPY`).

When combined with a loop, the entire contents of the straight buffer would be shifted along one place.

Example 10-11.

```
                LT *-, AR1      ; Load buffer[7]
                MPY *-, AR0     ; Multiply by constant
                                ; No DMOV here.
loop:           LTD *-, AR1     ; Add previous product to
                                ; the accumulator, load T
                                ; register and copy data to
                                ; the next higher memory
                                ; location.
                MPY *-, AR2     ; Multiply by constant.
                BANZ loop, *-, AR0
                                ; Test AR2 for zero, and
                                ; if so, branch to the
                                ; label loop. If not
                                ; zero, execute the next
                                ; instruction. Make AR0
                                ; the current auxiliary
                                ; register.
                APAC            ; Accumulate final value.
```

The function of Example 10-9 and 10-11 is identical, but the latter requires less instructions and less clock cycles.

Multiplication and Accumulation Using Signed Numbers

Consider the case where we are multiplying together two signed numbers. The two largest numbers we could multiply are 7FFFh by 7FFFh (32767 by 32767). The most positive product produced is 3FF0001. There are effectively two sign bits where only one is required.

The TMS320F24x provides a mechanism to allow us to get rid of this extra sign bit.

It can be configured to do so by shifting the product in the P register one place to the left before adding to the accumulator. This uses the instruction Set P Mode (SPM)

The code fragment in Example 10-12 shows the effect of the instruction SPM when using with differing operands.

Example 10-12.

```
                .setsect        ".text", 8800h

start:          LDP #6          ; Data Page 6. Access 300h to 3FFh.
                SPLK #1, 0h     ; Data memory address 300h contains 1h.
                LT 0h           ; T register contains 1h.
                MPY #0FFFh      ; P register contains 0FFFh.
                LACC #0         ; Clear accumulator to zero.
                SPM 0           ; PM = 00. No shift of P register when
                                ; accumulating.
                APAC            ; Add P register to accumulator. No
                                ; shift of P register involved.
                                ; Accumulator contains 00000FFFh.

                LACC #0         ; Clear accumulator.
                SPM 1           ; PM = 01. Shift P one place left
                                ; before adding to the accumulator.
                APAC            ; Add the contents of P register
                                ; shifted one place to the left to the
                                ; accumulator. Accumulator contains
                                ; 00001FFEh.
                B start         ; Go round again.
```

Implementing a Finite Impulse Response (FIR) Filter

When using the contents of a buffer for digital filtering, it is necessary to use each value in the buffer. Normally, a multiplication would be done and the product added to a running total. After all the calculations have been done, the values in the buffer are shuffled along one and the new input value read.

Converting the ADC Measurement to Signed Values

The input from the ADC produces positive values in the range 0 to FFC0h. A Finite Impulse Response (FIR) Filter uses positive and negative values about a central zero.

The maximum ADC input is FFC0h, which is treated as an unsigned number. To convert this to a signed number, we can insert a zero in bit 15 (the left-most bit.) This gives us a maximum value of 7FE0h. If we now subtract the value of the half-way point, that is 7FE0h/2 = 3FF0h, we now have a maximum value of 3FF0h (+16368 decimal).

The minimum value will now be 0 – 3FF0h = - 3FF0h = C010h (-16368 decimal).

Converting the FIR Filter Output to an Unsigned Value

The output of the FIR filter will be a signed value. This we need to convert to a signed value for the Digital to Analog Conversion (DAC).

In Example 10-13, the maximum value of the FIR is 4AF93D00h and the minimum value is B506C300h. If we were to discard the low (right-most) 16 bits, we would have:

4AF9h = + 19193 decimal.
B506h = - 19193 decimal.

To convert the signed values to unsigned, we add 19193 to the FIR filter output. This would give us a maximum of 19193 + 19193 = 38386 decimal. The minimum value would be – 19193 + 19193 = 0 (zero decimal).

Finally, in order to convert the unsigned FIR output in the range 0 to 38386 decimal (0 to 95F2h) to a value usable by the PWM (pulse width modulation), we would apply the appropriate scaling factor.

If we were to use a PWM period of 1023 (3FFh), then the scaling factor would be calculated as follows:

95F2h *x* scaling factor = 3FF0000h
scaling factor = 6D2h

A Complete FIR Filter

In Example 10-13, the complete code is shown to implement a low-pass filter using the TMS320F243 DSK. It assumes that the input value from the ADC has its zero at the mid point of the range, that is at 200h.

The constants are taken for a low-pass filter are taken from the Chassaing book given in the Reference Section.

Example 10-13.

```
DP_ADC      .set 244        ; Data Page for ADC
ADCTRL1     .set 32h        ; ADC Control Register 1
ADCTRL2     .set 34h        ; ADC Control Register 2
ADCFIFO1    .set 36h        ; Result of ADC
                            ; conversion.
DP_GP       .set 6h         ; General purpose.
            
N           .set 11-1       ; Index for array size
```

```
                    .setsect ".data",       300h
                    .data
array:              .word 0,0,0,0,0         ; Fill array with known
                    .word 0,0,0,0,0         ; values.
                    .word 0                 ; Dummy value for data
                                            ; move.

constants:          .word 0                 ; FIR constants
                    .word 1534
                    .word 3306
                    .word 4961
                    .word 6134
                    .word 6554
                    .word 6134
                    .word 4961
                    .word 3306
                    .word 1534
                    .word 0h

factor:             .word 06D2h             ; PWM scaling factor.
result:             .word 0h                ; Value to send to PWM.

                    .setsect ".text",       8800h
                    .text

start:              LDP #DP_ADC             ; Data page for ADC.
                    SPLK #0C01h, ADCTRL1    ; Continuous conversion.
                    SPLK #0007h, ADCTRL2    ; Conversion time.

loop:               LAR AR4, #array         ; AR4 points to start of
                                            ; array.
                    MAR *, AR4              ; Make AR4 the current
                                            ; auxiliary register.
                    CLRC SXM                ; ADC reading is
                                            ; unsigned.
                    LACC ADCFIFO1           : Load ADC measurement
                                            ; into the accumulator
                    SFR                     ; Make room for sign
                                            ; bit.
                    SUB #0FFC0h/4           ; Convert unsigned ADC
                                            ; measurement to signed.
                    SACL *                  ; Saved signed value.

                    LAR AR0, #array + N     ; Point to last element
                                            ; of input buffer
                    LAR AR1, #constants+N   ; Point to last
                                            ; constant.
                    LAR AR3, #N             ; Number of
                                            ; multiplications to be
```

```
                        ; done

                        MAR *, AR0      ; Make AR0 the current
                                        ; auxiliary register.
                        LACC #0         ; Clear the accumulator.
                        MPY #0          ; Clear the P register.
                        SPM 1           ; Shift product right
                                        ; before accumulating.
                        SETC SXM        ; Work with signed
                                        ; values.

loop1:                  LTD *-, AR1     ; Load the T register
                                        ; with an element from
                                        ; the input array.
                                        ; Decrement the pointer
                                        ; to next lower
                                        ; location. Copy data to
                                        ; next element. Make AR1
                                        ; the current auxiliary
                                        ; register.
                        MPY *-, AR3     ; Multiply input value
                                        ; by corresponding
                                        ; constant. Decrement
                                        ; the pointer to the
                                        ; next constant. Make
                                        ; AR3 the current
                                        ; auxiliary register.
                        BANZ loop1, *-, AR0
                                        ; Test AR3 then
                                        ; decrement it. If the
                                        ; value is non-zero,
                                        ; branch to the label
                                        ; loop1. Make AR0 the
                                        ; current auxiliary
                                        ; register.

                        APAC            ; Accumulate the final
                                        ; product

                        CLRC SXM        ; We need 0 in msb of
                                        ; accumulator. Turn off
                                        ; sign-extension mode.
                        ADD #4AF9h*2, 15
                                        ; Convert to unsigned.
                        ADD #3D00h      ; Adjust low part.

                        LDP #DP_GP      ; General purpose use.
                        SACH result     ; Save high word result
                        LT result       ; Copy to T register.
                        SPM 0           ; PM = 0. No shift.
                        MPYU factor     ; Result x factor.

                        PAC             ; Copy to accumulator
                        ADD #8000h      ; Round up. Add
```

		; equivalent of 0.5 ; decimal.
	SACH result	; Range = 0 to 3FFh.
	LDP #DP_ADC	; For next measurement.
	B *loop*	; Go round again
.end		

Note that to add the value 4AF93D00h to the accumulator is not that easy. The maximum shift available with the instruction ADD is 15. This means we have to add an extra shift ourselves. We do this by multiplying 4AF9h by two.

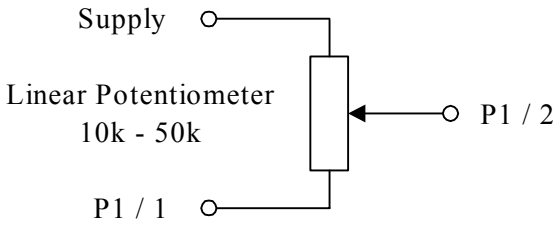
The most significant bit of the accumulator (left-most bit) is set using the instructions SETC SXM or CLRC SXM. In this case, we want the most significant bit to be zero, so we use the instruction CLRC SXM. On the other hand, had we wished the most significant bit to be zero, we would use the instruction SETC SXM.

TMS320F243 DSK Experiments

Equipment Required

TMS320F243 DSK
Linear Potentiometer connected as per Figure 10-2.

Figure 10-2.



For the TMS320F243 DSK, the supply can be up to 5V. However, for 3.3V devices such as the TMS3420LF2407, the maximum supply is 3.3V.

Experiment 10-1.

Objective: To show how input values are stored in a buffer

Enter the code given in Example 10-6 into a text editor and save it. Assemble the program and load into the debugger. Setup a memory window containing 8 or more values starting at data memory address 300h. SingleStep through the code. After the value has been read from ADCFIFO1, turn the potentiometer a little.

Repeat until all the values measured by the Analog-to-digital converter have been entered into the buffer.

Experiment 10-2.

Objective: To show the effect of multiplication with accumulation.

Enter the code from Example 10-8 into a text editor, save it, assemble it and load into the debugger. SingleStep through the code, noting the value in the P (product register). The contents of the P register should change as follows:

64
64 + 49
64 + 49 + 36
64 + 49 + 36 + 25
64 + 49 + 36 + 25 + 16

64 + 49 + 36 + 25 + 16 + 9
64 + 49 + 36 + 25 + 16 + 9 + 4
64 + 49 + 36 + 25 + 16 + 9 + 4 + 1

Experiment 10-3.

Objective: To show accumulation with data shift.

Enter the code in Example 10-9 into a text editor and save it. Assemble it and load into the debugger. Setup a memory window for data starting at address 300h, containing 8 locations. SingleStep through the code and monitor both the buffer at data memory address 300h and the P register. Accumulation should occur and the data in the buffer should be shifted.

Modify Example 10-9, as shown in Example 10-11 to replace three instructions with the single instruction `LTD`. Save the program and assemble it. Load into the debugger and use SingleStep to go through the program. The behavior should be exactly the same as when the three instructions were used.

Experiment 10-4.

Objective: To show the effect of PM

Enter Example 10-12 into a text editor and save it. Assemble it and load the program into the debugger. SingleStep through the code. When PM = 00, then there is no shift when accumulating. When PM = 01, the value in the P register is shifted one place to the left to lose the second sign bit.

Experiment 10-4.

Objective: To show how a Buffer would be used for a FIR Filter

Enter the code in Example 10-13 into a text editor and save it. Assemble it and load into the debugger. Setup a memory window for data starting at data memory address 300h and which contains 12 elements. Monitor the P register and the accumulator.

SingleStep through the program to observe how the inputs from the ADC are loaded into a buffer, multiplied by each constant and then accumulated.

Set a breakpoint at the instruction `B loop` and continually run the program to the breakpoint, moving the potentiometer each time.

Design Problem 10-1.

Write code to load the accumulator with a known value then add 12345678h to it. Then add 80000001h to the accumulator. To carry out the additions, the instruction `ADD` will be required with shift, as well as to set the correct value of `SXM`.

Self-Test Questions..... *Click Here To View* Answers!

1.	What is meant by the term *buffer*?
2.	How do we define a buffer in C?
3.	When using the instruction `DMOV` with a repeat, why do we start from the high end and work downwards?
4.	Why is a straight buffer useful for DSP?
5.	Why is the instruction `DMOV` not often used on its own?

6.	Combine the following instructions into a single instruction:
	`APAC`
	`LT *`
	`DMOV *`

7.	Why do we not use the instruction `DMOV` for the highest value in the buffer?
8.	When using a buffer, why would we use indirect addressing rather than direct addressing?
9.	Why might we place a dummy value at the end of an array?
10.	Which one of the following is correct? a) `BANZ label, *-, AR3` b) `BANZ label, AR3, *-` c) `BANZ *-, label, AR3` d) `BANZ *-, AR3, label` e) `BANZ AR3, label, *-` f) `BANZ AR3, *-, label`
11.	The instruction `BANZ` means which of the following: a) Branch on auxiliary register non-zero b) Branch and negate zero c) Byte and with zero d) Bit arithmetical negate zero e) Branch and zero
12.	Why should we want to shift the product of a multiplication of two signed variables to the left before adding it to the accumulator?
13.	The instruction `SPM` means which one of the following: a) Set P Mode b) Shift Product Module c) Subtract Product and Move d) Shift Parity Mode e) Store Part Move
14.	In order to add a value of greater than FFFFh to the accumulator, why must we set the appropriate value of sign-extension mode (SXM)?

References

TMS320F/C24x DSP Controllers. CPU and Instruction Set. Reference number SPRU160

TMS320F/C240 DSP Controllers. Peripheral Library and Specific Devices. Reference Number SPRU161.

TMS320F243, TMS320F241 DSP Controllers. Reference Number SPRS064.

Rulph Chassaing. Digital Signal Processing with C and the TMS320C30. Chapter 4. Finite Response Filters.

CLICK TO VIEW
Tutorials
1 2 3 4 5 6 7 8 9 10

Click here to view..... Route Map