

# Session 4

DSP Hardware Lab

Optimized Convolution : FIR Filters in  
Assembly

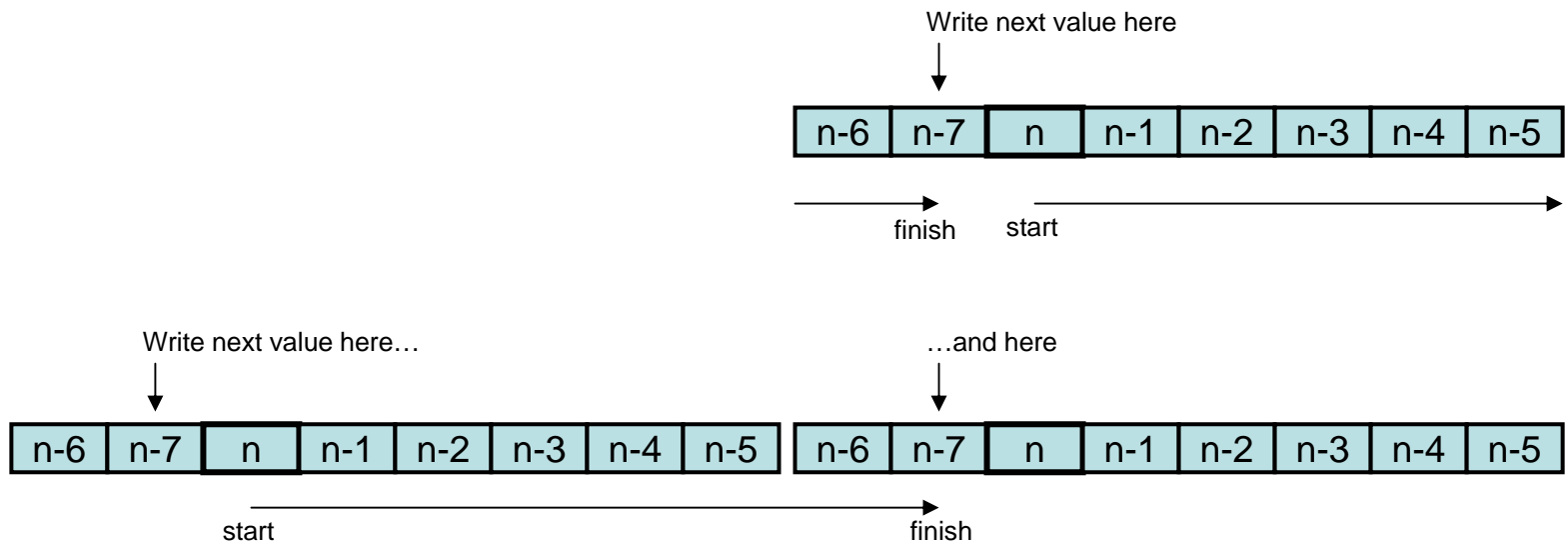
# Review : Computing Convolution

---

- Hold Last N samples  $x[n]$ ,  $x[n-1]$ ...
- For each new input sample  $x[n]$ , compute a  $y[n]$ 
  - $h[0] * x[n] + h[1]*x[n-1] + \dots + h[N-1]*x[n-N+1]$
- Shifting Data
- Circular Buffer

# Circular Buffers in C

- To implement the circular buffers in C, we will need to perform the address arithmetic manually, including:
  - Keeping track of the next index to write to
  - Manually “looping” from the end of the circular buffer to the beginning
- This technique can be sped up a number of ways:
  - Only allowing the number of taps to be a power of 2 means that address calculations can avoid the complexity of a modulo operation or a branch
  - Double-buffering the input with two circular buffers means that through “unrolling” the processing can access the memory in a linear fashion



# Hardware Circular Buffers

---

- Hardware support allows the convolution sum without moving data, and without the modulo address calculation
  - Simply MAC (multiply – accumulate), increment pointers, MAC, ...etc.
- Requirements for use
  - Buffer length must be of size  $2^{**}N$
  - Buffer must be aligned in memory on a  $2^{**}N$  boundary

# Hardware Circular Buffers

- Length of filter =  $2^{**}N$ 
  - Pad coefficients with 0's
  - make a better filter
- Aligning buffer on  $2^{**}N$  boundary
  - `#pragma DATA_ALIGN(x, BUF_LEN)`
    - x is the variable
    - BUF\_LEN is size of buffer in **bytes** (*i.e. for 32 tap filter, 128*)

128 = binary : 10000000, hex  
Ex : address = 0x32180

# Circular Addressing

---

- Tell processor to (for a particular register) not allow carries or borrows across a specified bit boundary.
  - So upper bits (above that boundary) don't change as the address is incremented or decremented
    - This is why you align, (we can't need upper bits to move)

# Circular Addressing

8 Tap filter

Buffer size = 8 words = 32 bytes

Align on 32 byte boundary.

(ex. 0x32120)

Tell processor to not allow carries/borrows to bit 5, i.e. lower 5 bits remain 0

0x32120

X[N-3]

0x32124

X[N-2]

0x32128

X[N-1]

0x3212C

X[N]

0x32130

X[N-7]

0x32134

X[N-6]

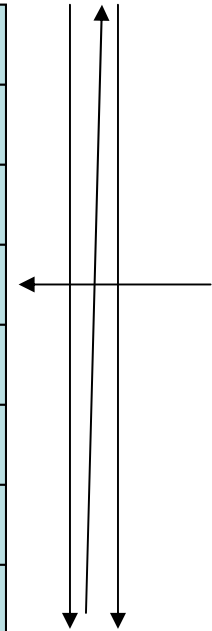
0x32138

X[N-5]

0x3213C

X[N-4]

Fill



For computation, address calc can be ignored, just subtract 4 each time and it goes 0x3212C, 0x32128, 0x32124, 0x32120, 0x3213C...etc. Same thing works for addition

# Implementation Details

---

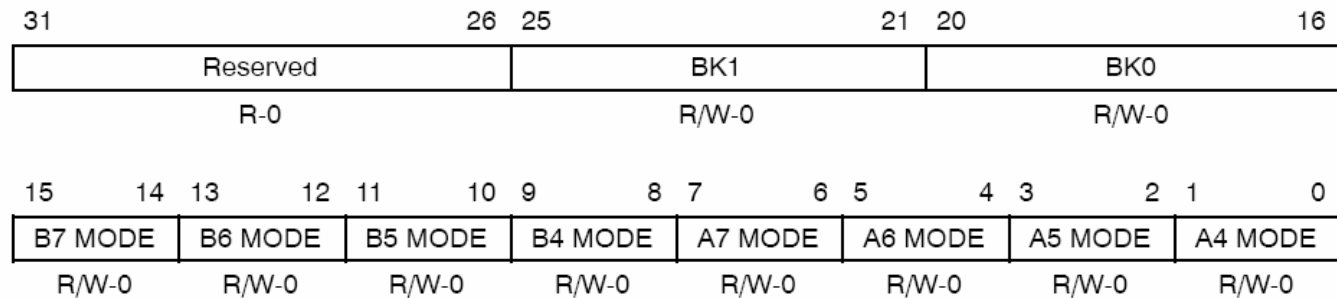
- Can't be done in C
  - Requires specific registers to be used, with specific addressing modes
- Registers A4,A5,A6,A7 and B4,B5,B6,B7 can be used for **indirect addressing** (either linear or circular)
  - Ex : LDW \*A4++, A8
    - Get the data who's address is in A4, and put that data in A8. At the same time increment A4 by 4

# Implementation Details

---

- Indirect Addressing is means for circular buffering.
  - One of the indirect addressing registers holds the address of the data buffer (or the current sample location in that buffer)
  - Address Mode Register is set to tell the CPU that this register is circular, and what size
  - AMR is not memory mapped, rather it is a special control register in the CPU core itself. You can write to it via the special instruction MVC

Figure 2–3. Addressing Mode Register (AMR)



**Legend:** R = Readable by the **MVC** instruction; W = Writeable by the **MVC** instruction; -n = value after reset

Table 2–5. Addressing Mode Register (AMR) Field Descriptions

Bit	Field	Value	Description
31–26	Reserved	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
25–21	BK1	0–1Fh	Block size field 1. A 5-bit value used in calculating block sizes for circular addressing. <a href="#">Table 2–6</a> shows block size calculations for all 32 possibilities. <i>Block size (in bytes) = <math>2^{(N+1)}</math>, where N is the 5-bit value in BK1</i>
20–16	BK0	0–1Fh	Block size field 0. A 5-bit value used in calculating block sizes for circular addressing. <a href="#">Table 2–6</a> shows block size calculations for all 32 possibilities. <i>Block size (in bytes) = <math>2^{(N+1)}</math>, where N is the 5-bit value in BK0</i>
15–14	B7 MODE	0–3h	Address mode selection for register file B7.
		0	Linear modification (default at reset)
		1h	Circular addressing using the BK0 field
		2h	Circular addressing using the BK1 field
		3h	Reserved

# Programming In Assembly

---

- Benefit : we can use some of the hardware features that we have no way of expressing in C
  - AMR for circular addressing
  - Other control bits for special features (ex. Saturating arithmetic)
- Benefit : we can also be generally more efficient than the code we write in C
- Drawback : [spru733.pdf](#)
- Compromise : Write parts that need to be fast in ASM, rest in C for simplicity.
  - We need to know “the rules” of register usage..etc. so that we can co-exist with the code generated by the C compiler.
  - These rules are well documented.

# Proposed Design Flow

---

- Write Program Entirely in C
  - Understandable code
  - Fast development
  - Portable code
- Use profiling tools / common sense to decide where to optimize by hand
  - Hand optimized assembly
  - Linear Assembly (use optimizing assembler)
  - Prefab library routines

# Register Usage (from spru187)

Where do arguments come to your asm, and how it returns values back to C

You are responsible for Preserving their value

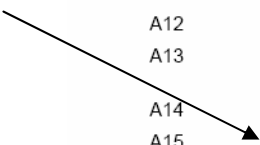


Table 8-2. Register Usage

Register	Function Preserved By	Special Uses	Register	Function Preserved By	Special Uses
A0	Parent	–	B0	Parent	–
A1	Parent	–	B1	Parent	–
A2	Parent	–	B2	Parent	–
A3	Parent	Structure register (pointer to a returned structure)	B3	Parent	Return register (address to return to)
A4	Parent	Argument 1 or return value	B4	Parent	Argument 2
A5	Parent	Argument 1 or return value with A4 for doubles, longs and long longs	B5	Parent	Argument 2 with B4 for doubles, longs and long longs
A6	Parent	Argument 3	B6	Parent	Argument 4
A7	Parent	Argument 3 with A6 for doubles, longs, and long longs	B7	Parent	Argument 4 with B6 for doubles, longs, and long longs
A8	Parent	Argument 5	B8	Parent	Argument 6
A9	Parent	Argument 5 with A8 for doubles, longs, and long longs	B9	Parent	Argument 6 with B8 for doubles, longs, and long longs
A10	Child	Argument 7	B10	Child	Argument 8
A11	Child	Argument 7 with A10 for doubles, longs, and long longs	B11	Child	Argument 8 with B10 for doubles, longs, and long longs
A12	Child	Argument 9	B12	Child	Argument 10
A13	Child	Argument 9 with A12 for doubles, longs, and long longs	B13	Child	Argument 10 with B12 for doubles, longs, and long longs
A14	Child	–	B14	Child	Data page pointer (DP)
A15	Child	Frame pointer (FP)	B15	Child	Stack pointer (SP)
A16-A31	Parent	C6400 only	B16-B31	Parent	C6400 only
ILC	Child	C6400+ only, loop buffer counter	NRP	Parent	
IRP	Parent		RILC	Child	C6400+ only, loop buffer counter

All other control registers are not saved or restored by the compiler.

The compiler assumes that control registers not listed in [Table 8-2](#) that can have an effect on compiled code have default values. For example, the compiler assumes all circular addressing-enabled registers are set for linear addressing (the AMR is used to enable circular addressing). Enabling circular addressing and then calling a C/C++ function without restoring the AMR to a default setting violates the calling convention. You must be certain that control registers which affect compiler-generated code have a default value when calling a C/C++ function from assembly.

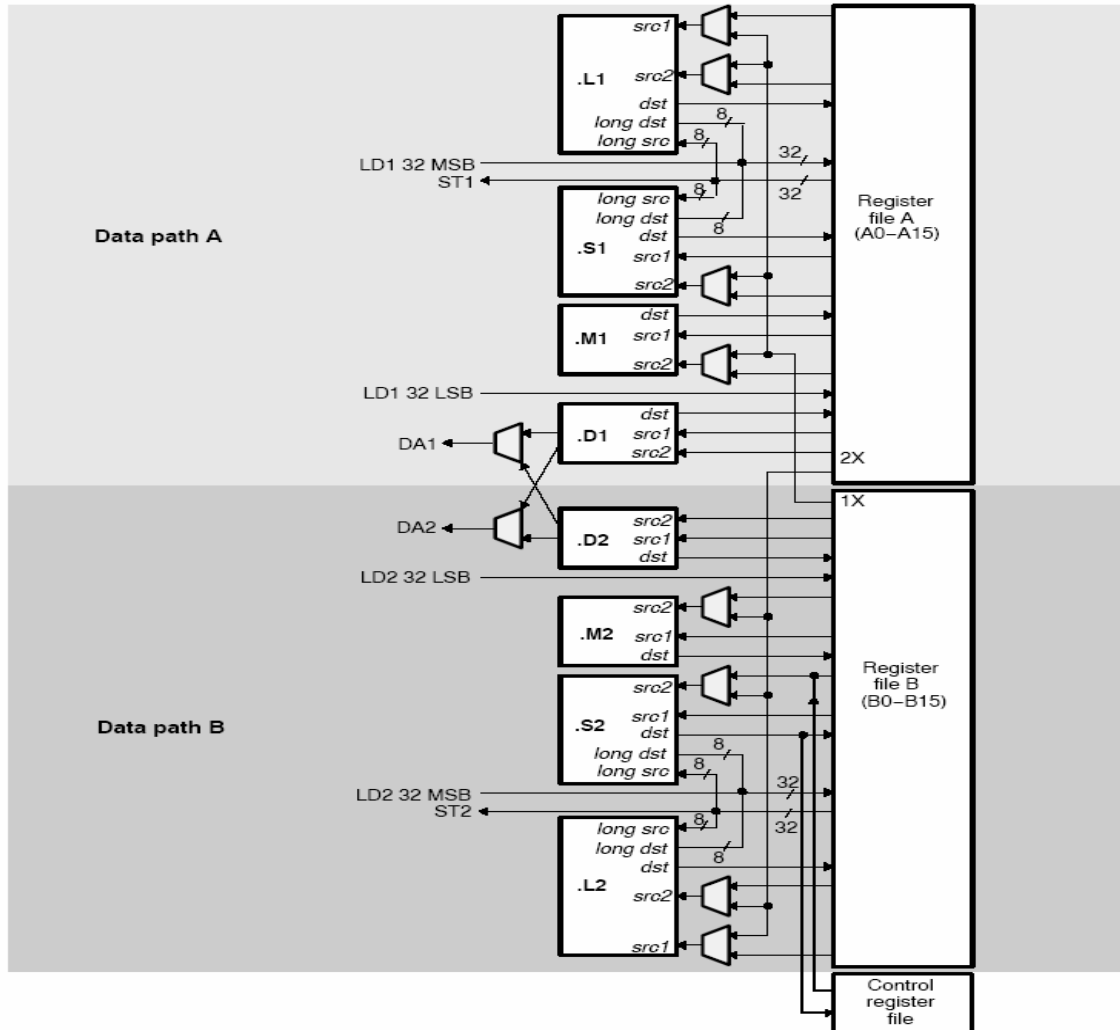
# Excerpt from spru187

---

You are responsible for handling the AMR control register and the SAT bit in the CSR correctly inside an interrupt. By default, the compiler does not do anything extra to save/restore the AMR and the SAT bit. Macros for handling the SAT bit and the AMR register are included in the `c6x.h` header file. For example, you are using circular addressing in some hand assembly code (that is, the AMR does not equal 0). This hand assembly code can be interrupted into a C code interrupt service routine. The C code interrupt service routine assumes that the AMR is set to 0. You need to define a local unsigned int temporary variable and call the `SAVE_AMR` and `RESTORE_AMR` macros at the beginning and end of your C interrupt service routine to correctly save/restore the AMR inside the C interrupt service routine.

# Datapaths / Functional Units

Figure 2-1. TMS320C67x CPU Data Paths



# Datapaths / Functional Units

Functional Unit	Fixed-Point Operations	Floating-Point Operations
.L unit (.L1, .L2)	32/40-bit arithmetic and compare operations 32-bit logical operations Leftmost 1 or 0 counting for 32 bits Normalization count for 32 and 40 bits	Arithmetic operations DP → SP, INT → DP, INT → SP conversion operations
.S unit (.S1, .S2)	32-bit arithmetic operations 32/40-bit shifts and 32-bit bit-field operations 32-bit logical operations Branches Constant generation Register transfers to/from control register file (.S2 only)	Compare Reciprocal and reciprocal square-root operations Absolute value operations SP → DP conversion operations SP and DP adds and subtracts SP and DP reverse subtracts (src2 – src1)
.M unit (.M1, .M2)	16 × 16-bit multiply operations 32 × 32-bit multiply operations	Floating-point multiply operations Mixed-precision multiply operations
.D unit (.D1, .D2)	32-bit add, subtract, linear and circular address calculation Loads and stores with 5-bit constant offset Loads and stores with 15-bit constant offset (.D2 only)	Load doubleword with 5-bit constant offset

# Hand Optimized Assembly

---

- Knows exactly what each unit is doing and when
  - Seeks to optimize parallelization
    - 8 parallel operations at once can be launched from the pipeline scheduler
- Understands the pipeline
  - Multiple clocks required to execute the full part of each instruction
  - Seeks to minimize waiting
- Uses the most efficient instructions for the desired operations

# Example Instruction

## MPYSP

*Multiply Two Single-Precision Floating-Point Values*

### Syntax

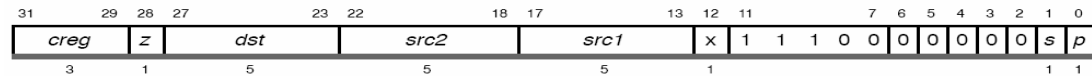
**MPYSP** (.unit) *src1*, *src2*, *dst*

.unit = .M1 or .M2

### Compatibility

C67x and C67x+ CPU

### Opcode



Opcode map field used...	For operand type...	Unit
<i>src1</i>	sp	.M1, .M2
<i>src2</i>	xsp	
<i>dst</i>	sp	

### Description

The *src1* operand is multiplied by the *src2* operand. The result is placed in *dst*.

### Pipeline

Pipeline Stage	E1	E2	E3	E4
Read	<i>src1</i> <i>src2</i>			
Written				<i>dst</i>
Unit in use	.M			

If *dst* is used as the source for the **ADDDP**, **CMPEQDP**, **CMPLTDP**, **CMPGTDP**, **MPYDP**, or **SUBDP** instruction, the number of delay slots can be reduced by one, because these instructions read the lower word of the DP source one cycle before the upper word of the DP source.

Instruction Type 4-cycle

Delay Slots 3

Functional Unit Latency 1

# Pipeline Stages

1. The *program fetch stage* is composed of four phases:
  - (a) *PG*: program address generate (in the CPU) to fetch an address
  - (b) *PS*: program address send (to memory) to send the address
  - (c) *PW*: program address ready wait (memory read) to wait for data
  - (d) *PR*: program fetch packet receive (at the CPU) to read opcode from memory
2. The *decode stage* is composed of two phases:
  - (a) *DP*: to dispatch all the instructions within an FP to the appropriate functional units
  - (b) *DC*: instruction decode
3. The *execute stage* is composed of 6 phases (with fixed point) to 10 phases (with floating point) due to delays (latencies) associated with the following instructions:
  - (a) Multiply instruction, which consists of two phases due to one delay
  - (b) Load instruction, which consists of five phases due to four delays
  - (c) Branch instruction, which consists of six phases due to five delays

Table 3.2 shows the pipeline phases, and Table 3.3 shows the pipelining effects. The first row in Table 3.3 represents cycle 1, 2, . . . , 12. Each subsequent row represents an FP. The rows represented PG, PS, . . . illustrate the phases associated with each FP. The program generate (PG) of the first FP starts in cycle 1, and the PG of the second FP starts in cycle 2, and so on. Each FP takes four phases for program fetch and two phases for decoding. However, the execution phase can take from 1

TABLE 3.2 Pipeline Phases

Program Fetch				Decode		Execute
PG	PS	PW	PR	DP	DC	E1-E6 (E1-E10 for double precision)

TABLE 3.3 Pipelining Effects

Clock Cycle											
1	2	3	4	5	6	7	8	9	10	11	12
PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	E6
	PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5
		PG	PS	PW	PR	DP	DC	E1	E2	E3	E4
			PG	PS	PW	PR	DP	DC	E1	E2	E3
				PG	PS	PW	PR	DP	DC	E1	E2
					PG	PS	PW	PR	DP	DC	E1
						PG	PS	PW	PR	DP	DC

During the execution phase, sources are read. If data is available on the next cycle, then the delay slot is 0. If data is available, 2 clocks later, the delay slot is 1, etc.

# Delay Slot

## 3.4 Delay Slots

The execution of fixed-point instructions can be defined in terms of delay slots. The number of delay slots is equivalent to the number of cycles required after the source operands are read for the result to be available for reading. For a single-cycle type instruction (such as **ADD**), source operands read in cycle  $i$  produce a result that can be read in cycle  $i + 1$ . For a multiply instruction (**MPY**), source operands read in cycle  $i$  produce a result that can be read in cycle  $i + 2$ . Table 3–5 shows the number of delay slots associated with each type of instruction.

Delay slots are equivalent to an execution or result latency. All of the instructions that are common to the C62x, C64x, and C67x have a functional unit latency of 1. This means that a new instruction can be started on the functional unit each cycle. Single-cycle throughput is another term for single-cycle functional unit latency.

Table 3–5. Delay Slot and Functional Unit Latency Summary

Instruction Type	Delay Slots	Functional Unit Latency	Read Cycles†	Write Cycles†	Branch Taken†
NOP (no operation)	0	1			
Store	0	1	$i$	$i$	
Single cycle	0	1	$i$	$i$	
Multiply ( $16 \times 16$ )	1	1	$i$	$i + 1$	
Load	4	1	$i$	$i, i + 4$ §	
Branch	5	1	$i$ ‡		$i + 5$

† Cycle  $i$  is in the E1 pipeline phase.

‡ The branch to label, branch to IRP, and branch to NRP instructions instruction does not read any registers.

§ The write on cycle  $i + 4$  uses a separate write port from other .D unit instructions.

# More Assembly Instructions

## 2. Load/Store

### (a) The instruction

```
LDH .D2 *B2++, B7 ;load (B2) → B7, increment B2
|| LDH .D1 *A2++, A7 ;load (A2) → A7, increment A2
```

loads into B7 the half-word (16 bits) whose address in memory is specified/pointed to by B2. Then register B2 is incremented (postincremented) to point at the next higher memory address. In parallel is another indirect addressing mode instruction to load into A7 the content in memory whose address is specified by A2. Then A2 is incremented to point at the next higher memory address.

The instruction LDW loads a 32-bit word. Two paths using .D1 and .D2 allow for the loading of data from memory to registers A and B using the instruction LDW. The double-word load floating-point instruction LDDW on the C6713 can simultaneously load two 32-bit registers into side A and two 32-bit registers into side B.

### (b) The instruction

```
STW .D2 A1, *+A4 [20] ;store A1→(A4) offset by 20
```

stores the 32-bit word A1 in memory whose address is specified by A4 offset by 20 words (32 bits) or 80 bytes. The address register A4 is pre-incremented with offset, but it is not modified (two plus signs are used if A4 is to be modified).

## 3. Branch/Move. The following code segment illustrates branching and data transfer:

```
Loop MVKL .S1 x, A4 ;move 16LSBs of x address →A4
      MVKH .S1 x, A4 ;move 16MSBs of x address →A4
      .
      .
      .
      SUB .S1 A1, 1, A1 ;decrement A1
[A1] B .S2 Loop ;branch to Loop if A1 # 0
      NOP 5 ;five no-operation instructions
      STW .D1 A3, *A7 ;store A3 into (A7)
```

The first instruction moves the lower 16 bits (LSBs) of address  $x$  into register A4. The second instruction moves the higher 16 bits (MSBs) of address  $x$  into A4, which now contains the full 32-bit address of  $x$ . One must use the instructions MVKL/MVKH in order to get a 32-bit constant into a register.

Register A1 is used as a loop counter. After it is decremented with the SUB instruction, it is tested for a conditional branch. Execution branches to the label or address Loop if A1 is not zero. If A1 = 0, execution continues and data in register A3 are stored in memory whose address is specified (pointed) by A7.

# Example: Calling an assembly function

```
//Factorial.c Finds factorial of n. Calls function factfunc

#include <stdio.h>                                //for print statement

void main()
{
    short n = 7;                                  //set value
    short result;                                 //result from asm function
    result = factfunc(n);                         //call ASM function factfunc
    printf("factorial = %d", result);             //print result from asm function
}
```

**FIGURE 3.10.** C program that calls an ASM function to find the factorial of a number (factorial.c).

```
;Factfunc.asm Assembly function called from C to find factorial

                .def    _factfunc                ;ASM function called from C
_factfunc:     MV      A4,A1                    ;setup loop count in A1
                SUB    A1,1,A1                  ;decrement loop count
LOOP:          MPY    A4,A1,A4                  ;accumulate in A4
                NOP    ;for 1 delay slot with MPY
                SUB    A1,1,A1                  ;decrement for next multiply
[A1]          B      LOOP                      ;branch to LOOP if A1 # 0
                NOP    5                        ;five NOPs for delay slots
                B      B3                      ;return to calling routine
                NOP    5                        ;five NOPs for delay slots
                .end
```

**FIGURE 3.11.** ASM function called from C that finds the factorial of a number (factfunc.asm).

## Example 3.3: Factorial of a Number Using C Calling an Assembly Function

# Indirect Addressing

## 3.7.1 Indirect Addressing

Indirect addressing can be used with or without displacement. Register  $R$  represents one of the 32 registers A0 through A15 and B0 through B15 that can specify or point to memory addresses. As such, these registers are pointers. Indirect addressing mode uses a “\*” in conjunction with one of the 32 registers. To illustrate, consider  $R$  as an address register.

1.  $*R$ . Register  $R$  contains the address of a memory location where a data value is stored.
2.  $*R++(d)$ . Register  $R$  contains the memory address (location). After the memory address is used,  $R$  is postincremented (modified) such that the new address is the current address offset by the displacement value  $d$ . If  $d = 1$  (by default), the new address is  $R + 1$ , or  $R$  is incremented to the next higher address in memory. A double minus ( $--$ ) instead of a double plus would update or postdecrement the address to  $R - d$ .
3.  $*++R(d)$ . The address is preincremented or offset by  $d$ , such that the current address is  $R + d$ . A double minus would predecrement the memory address so that the current address is  $R - d$ .
4.  $*+R(d)$ . The address is preincremented by  $d$ , such that the current address is  $R + d$  (as with the preceding case). However, in this case,  $R$  preincrements without modification. Unlike the previous case,  $R$  is not updated or modified.

# DSPLIB

The TI C67x DSPLIB is an optimized DSP Function Library for C programmers using TMS320C67x devices. It includes C-callable, assembly-optimized general-purpose signal-processing routines. These routines are typically used in computationally intensive real-time applications where optimal execution speed is critical. By using these routines, you can achieve execution speeds considerably faster than equivalent code written in standard ANSI C language. In addition, by providing ready-to-use DSP functions, TI DSPLIB can significantly shorten your DSP application development time.

The TI DSPLIB includes commonly used DSP routines. Source code is provided that allows you to modify functions to match your specific needs.

The routines contained in the library are first classified in to single- and double-precision functions and then they are organized into seven different functional categories.

Table 3-5. Filtering and Convolution

Functions	Description	Page
void DSPF_sp_fir_cplx (float *x, float *h, float *r, int nh, int nr)	Complex FIR filter (radix 2)	4-38
void DSPF_sp_fir_gen (float *x, float *h, float *r, int nh, int nr)	FIR filter (general purpose)	4-40
void DSPF_sp_fir_r2 (float *x, float *h, float *r, int nh, int nr)	FIR filter (radix 2)	4-41
void DSPF_sp_fircirc (float x[], float h[], float r[], int index, int osize, int nh, int nr)	FIR filter with circularly addressed input	4-43
void DSPF_sp_biquad (float x[], float b[], float a[], float delay[], float r[], int nx)	Biquad filter (IIR of second order)	4-45
void DSPF_sp_iir (float *r1, float *x, float *r2, float *h2, float *h1, int nr)	IIR filter (used in VSELP vocoder)	4-46
void DSPF_sp_iirlat (float *x, int nx, float *k, int nk, float *b, float *r)	All-pole IIR lattice filter	4-48
void DSPF_sp_convovl (float *x, float *h, float *r, int nh, int nr)	Convolution	4-50

Table 3-6. Math

Functions	Description	Page
float DSPF_sp_dotp_sqr (float G, float *x, float *y, float *r, int nx)	Vector dot product and square	4-51
float DSPF_sp_dotprod (float *x, float *y, int nx)	Vector dot product	4-52
void DSPF_sp_dotp_cplx (float *x, float *y, int n, float *re, float *im)	Complex vector dot product	4-54
float DSPF_sp_maxval (float *x, int nx)	Maximum value of a vector	4-55
int DSPF_sp_maxidx (float *x, int nx)	Index of the maximum element of a vector	4-57
float DSPF_sp_minval (float *x, int nx)	Minimum value of a vector	4-58
void DSPF_sp_vecrecip (float *x, float *r, int n)	Vector reciprocal	4-59
float DSPF_sp_vecsum_sq (float *x, int n)	Sum of squares	4-60
void DSPF_sp_w_vec (float *x, float *y, float m, float *r, int nr)	Weighted vector sum	4-61
void DSPF_sp_vecmul (float *x, float *y, float *r, int n)	Vector multiplication	4-62

Table 3-7. Matrix

Functions	Description	Page
void DSPF_sp_mat_mul (float *x, int r1, int c1, float *y, int c2, float *r)	Matrix multiplication	4-64
void DSPF_sp_mat_trans (float *x, int rows, int cols, float *r)	Matrix transpose	4-65
void DSPF_sp_mat_mul_cplx (float *x, int r1, int c1, float *y, int c2, float *r)	Complex matrix multiplication	4-66

# DSPLIB

## 2.2.2 Calling a DSPLIB Function From C

In addition to correctly installing the DSPLIB software, you must follow these steps to include a DSPLIB function in your code:

- Include the function header file corresponding to the DSPLIB function
- Link your code with `dsp67x.lib`
- Use a correct linker command file for the platform you use. Remember most functions in `dsp67x.lib` are written assuming little-endian mode of operation.

For example, if you want to call the single precision Autocorrelation DSPLIB function, you would add:

```
#include <dspf_sp_autocor.h>
```

# Linear Assembly

---

- Examples of regular assembly shown in Chassaing : fig 4.31, 4.32.
  - Note that units don't need to be specified, we hope that assembler will do the best it can
- Linear Assembly (.sa) extension is a nice alternative which allows you to write in a linear fashion
  - Don't even assign registers, units...etc.
- Example from alternative text distributed in class

# Linear Assembly Flow

---

- Linear assembly code .sa extension written
- Assembler produces normal assembly out.
- Can explicitly “keep” this code
- You can now modify this code to use your registers for circular buffering...etc.

# End of Lecture

---