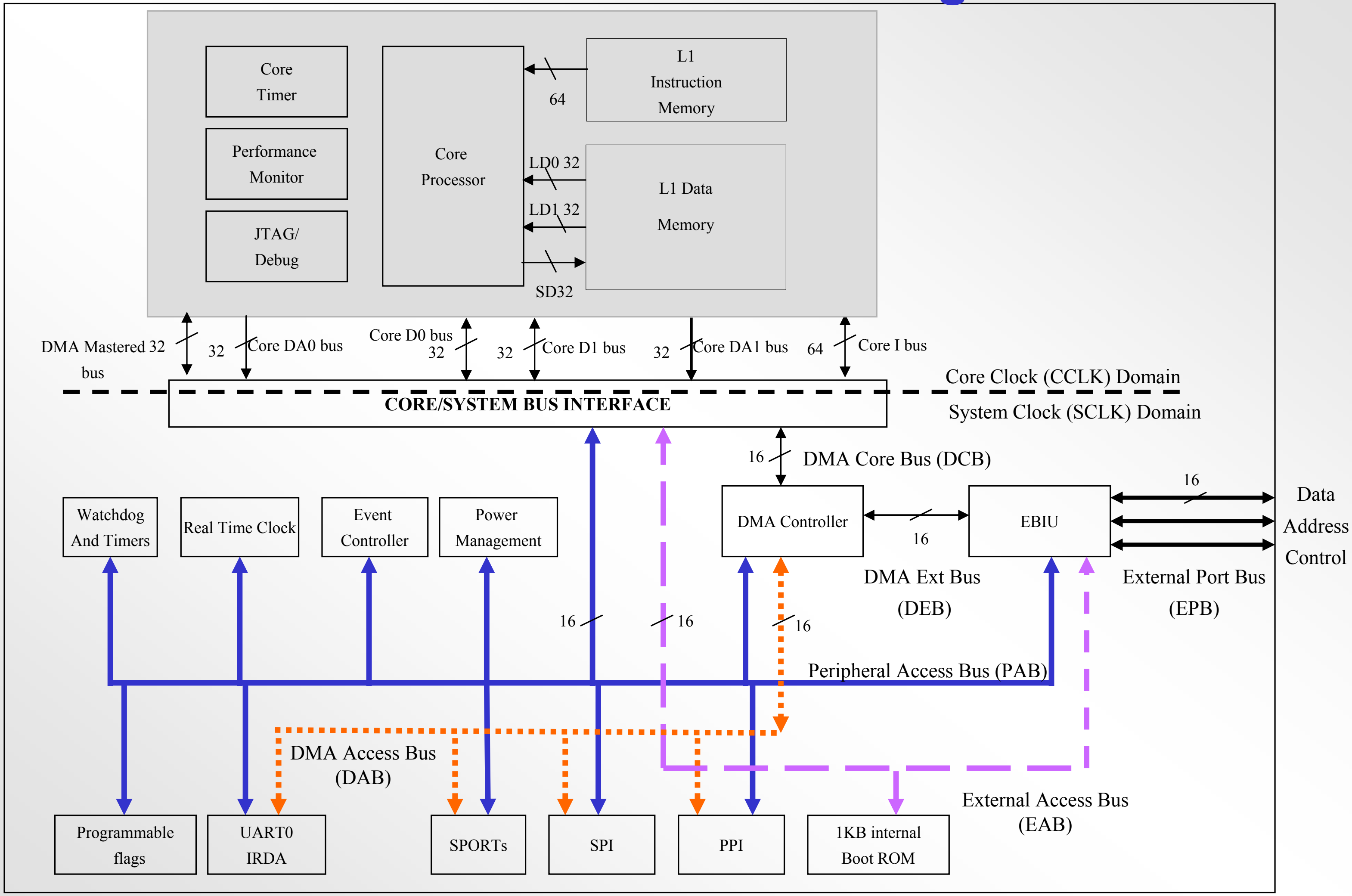
Section 5

Addressing Modes

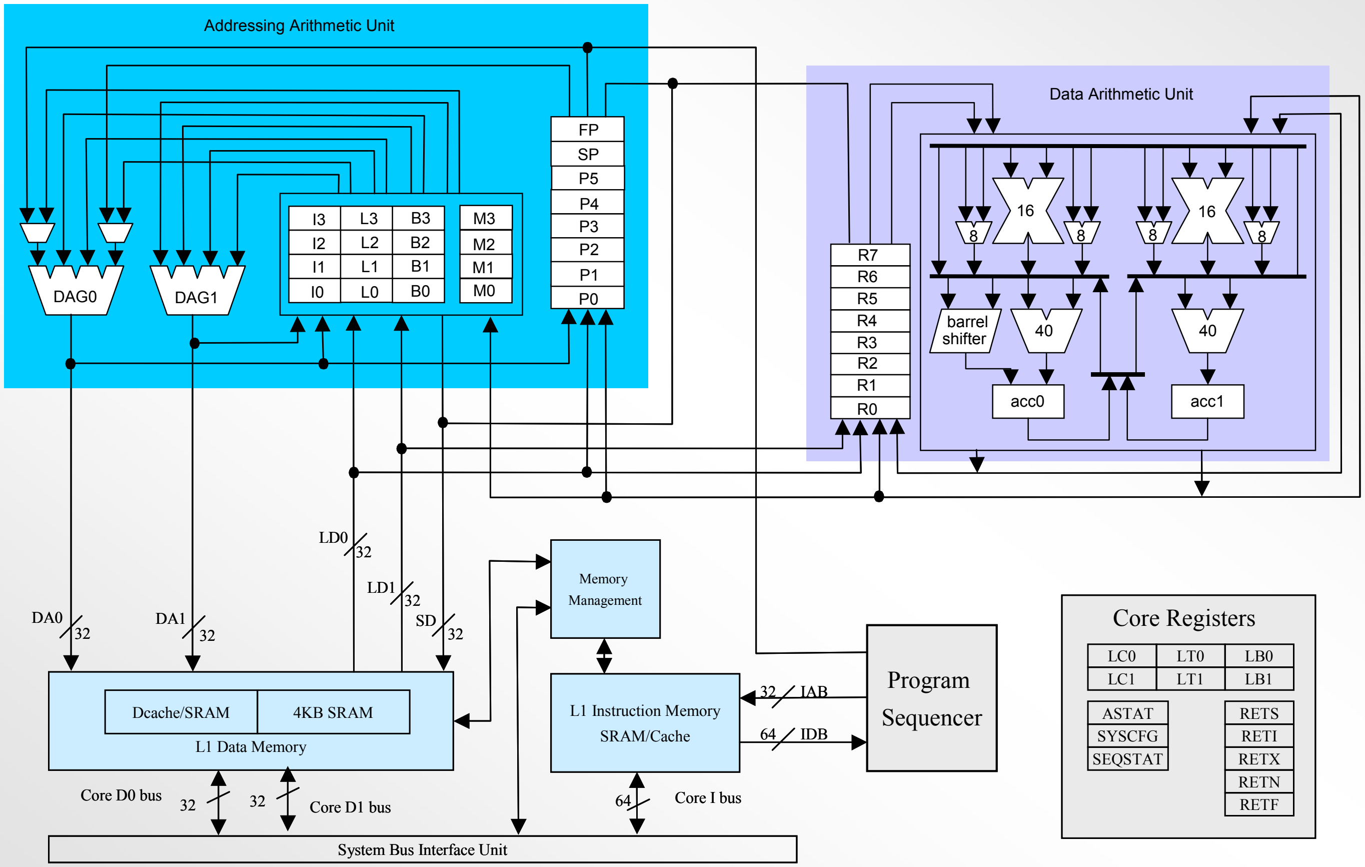
ADSP-BF533 Block Diagram



KAZTEK ENGINEERING

ANALOG DEVICES

ADSP-BF533 Core



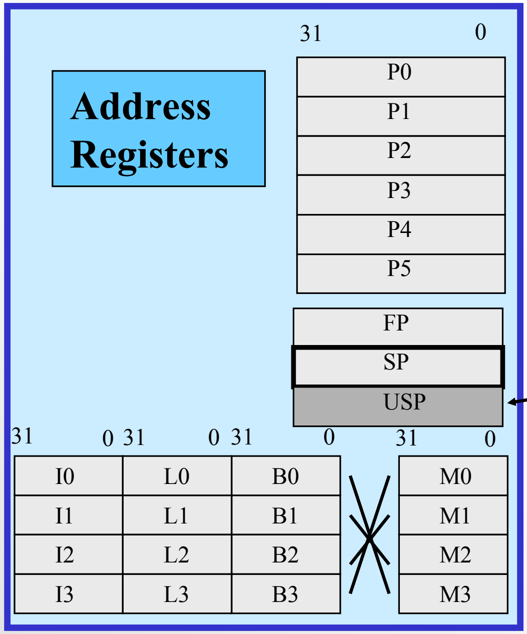
Core Registers

LC0	LT0	LB0
LC1	LT1	LB1

ASTAT	RETS
SYSCFG	RETI
SEQSTAT	RETX
	RETN
	RETF

Address Registers

- **One set of 32 bit general purpose Pointer registers**
  - P0-P5, SP and FP
- **One set of 32 bit DSP addressing Index registers**
  - I0-I3, B0-B3, L0-L3, M0-M3
- **All addresses are byte addresses into a 4 GB address space**



SP points to supervisor stack in Supervisor mode and user stack in User mode

USP is accessible in supervisor mode only – Allows access to user stack location while in Supervisor mode

Addressing Methods

- Register Indirect Addressing
  - Index Registers (32-bit and 16-bit accesses)
  - Pointer Registers P0 – P5 (32-bit, 16-bit, and 8-bit accesses)
  - Stack and Frame Pointer Registers (32-bit accesses)

- Types of address pointer modify
  - Modify/Post-Modify
    - Linear addressing
    - Circular buffering/modulo addressing
      - Enables automatic maintenance of pointers to stay within bounds of a circular buffer
    - Bit Reversal (Modify only)
  - Pre-Modify with update (using Stack Pointer)
  - Pre-Modify without update

Indirect Memory Access

- **Indirect Addressing**
  - **Square brackets ‘[‘ and ‘]’ denote the use of Index, Pointer and Stack/Frame Pointer Registers as address pointers in data fetches**

    - **Loads are of the general form:**

      dreg = [preg] ; // Where the preg points to some location in memory

      dreg = [ireg] ; // Where the ireg points to some location in memory

    - **Stores are of the general form:**

      [preg]=dreg ; // Where the preg points to some location in memory

      [ireg] =dreg ; // Where the ireg points to some location in memory

Indirect Addressing

- **Pointer Registers (P0-P5) support additional 16-bit (W) and 8-bit (B) options of the form:**
  - **Dreg_lo_hi = W[preg];** //loads 16-bit value pointed to by preg and loads into hi or lo half of dreg
  - **Dreg = B[preg] (z);** //loads 8-bit value pointed to by preg and loads into dreg

  Analogous store instructions also exist

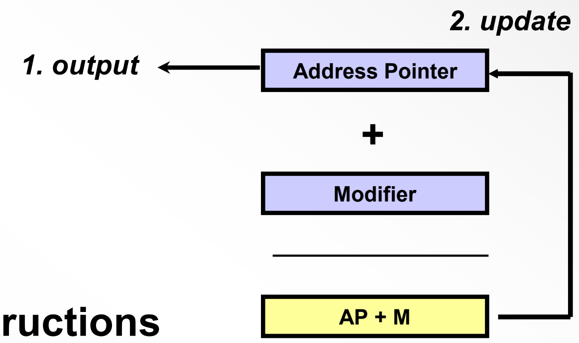
- **Index Registers (I0-I3) support an additional 16-bit (W) option of the form:**
  - **W[ireg]= Dreg_lo_hi;** //stores 16-bit value in dreg to location pointed to by ireg

  Analogous load instructions also exist

- **When an 8 or 16-bit value is transferred to a 32-bit register, an extension option must be used to specify sign (X) or zero (Z) extension**

  For example:
  - **R0=W[P0] (Z); // Loads 16 bit value into 32-bit register and zero //extends result**

Post-Modify Operations



- **Post-Modify Instructions**
  - **32-bit accesses**

```
R0 = [P0++];        /* Increments the value of P0 by 4 after the read */
R0 = [P1 ++ P2];    /* Increments P1 by P2 after reading 32-bit word from P1 only */
```

  - **16-bit accesses**

```
R0.l = W[I0--];     /* Decrements the value of I0 by 2 after the read */
R0.h = W[I2++M2];   /* Increments the value of I2 by M2 after reading 16-bit
                    word from I2 only */
```

  - **8-bit accesses**

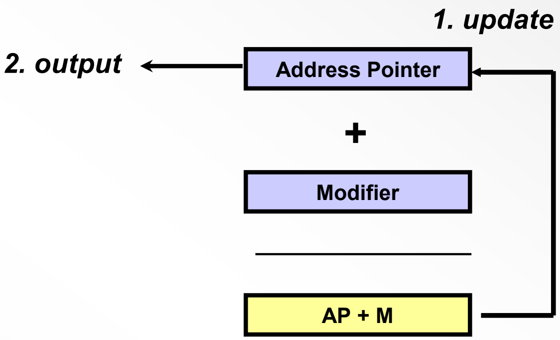
```
R0 = B[P0++](z);    /* Increments the value of P0 by 1 after the read */
R2 = B[P4 ++P5](x);/* Increments P4 by P5 after reading 8-bit word from P4 only */
```

- **Analogous store instructions exist**

KAZTEK ENGINEERING

ANALOG DEVICES

Pre-Modify Operations



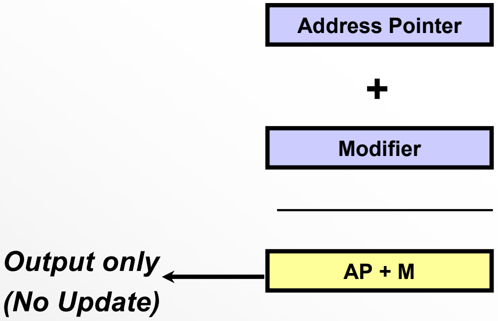
- The only pre-modify instruction with update supported uses the Stack Pointer
  - [-- SP] = R0; /* Decrements current value in SP by 4, and then writes the value in R0 to the updated value in SP */

Indexed Addressing with Immediate Offset

- Pointer Registers may be modified by an immediate value.

- These operations provide a pre-modify without update operation

```
P0 = [P1 + 0x10];       /* loads P0 with value that P1 + 0x10 points to */
[P0 + 0x20] = R0;       /* stores r0 in to the location that P0 + 0x20
                        points to */
```



Modifying DAG and Pointer Registers

- Direct modification of Index and Pointer Registers
  - Pointer registers use a P-register as modifier

    P0 += P1; /* P0 is modified by P1. */

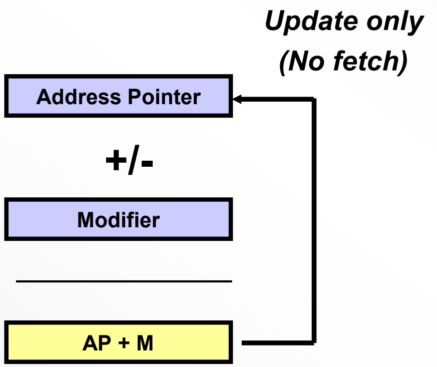
  - Index registers use an M-register as modifier

    I0 += M1; /* I0 is modified by M1. */

- Modify-Decrement supported as well as Modify-Increment

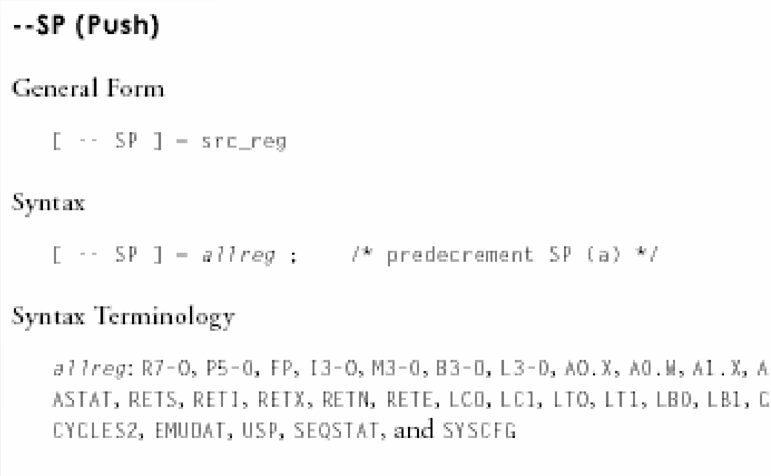
Example

- P0 -= P4;



Stack Instructions

- **Push Instruction: [--SP] = src_reg;**
  - **The push instruction stores the contents of a specified register or registers in the stack**
  - **The instruction pre-decrements the stack pointer to the next available location in the stack first**
  - **Push multiple instruction allows multiple registers to be placed on the stack with single instruction**



--SP (Push)

General Form

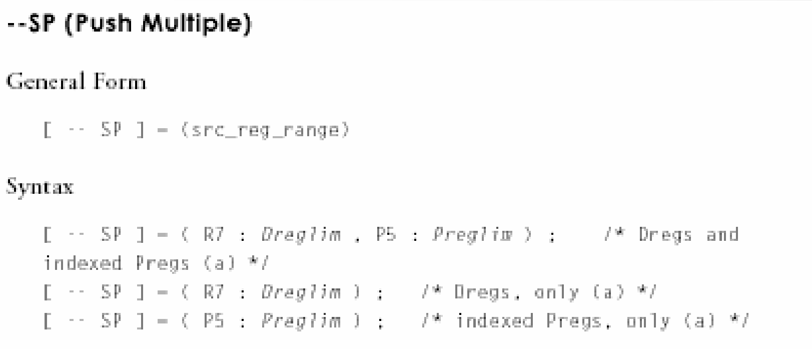
```
[ -- SP ] = src_reg
```

Syntax

```
[ -- SP ] = allreg ;     /* predecrement SP (a) */
```

Syntax Terminology

allreg: R7–0, P5–0, FP, I3–0, M3–0, B3–0, L3–0, A0.X, A0.W, A1.X, A1.W, ASTAT, RETS, RETI, RETX, RETN, RETE, LC0, LC1, LT0, LT1, LB0, LB1, CYCLES, CYCLES2, EMUDAT, USP, SEQSTAT, and SYSCFG



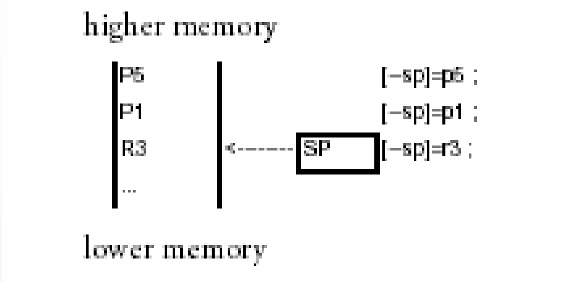
--SP (Push Multiple)

General Form

```
[ -- SP ] = (src_reg_range)
```

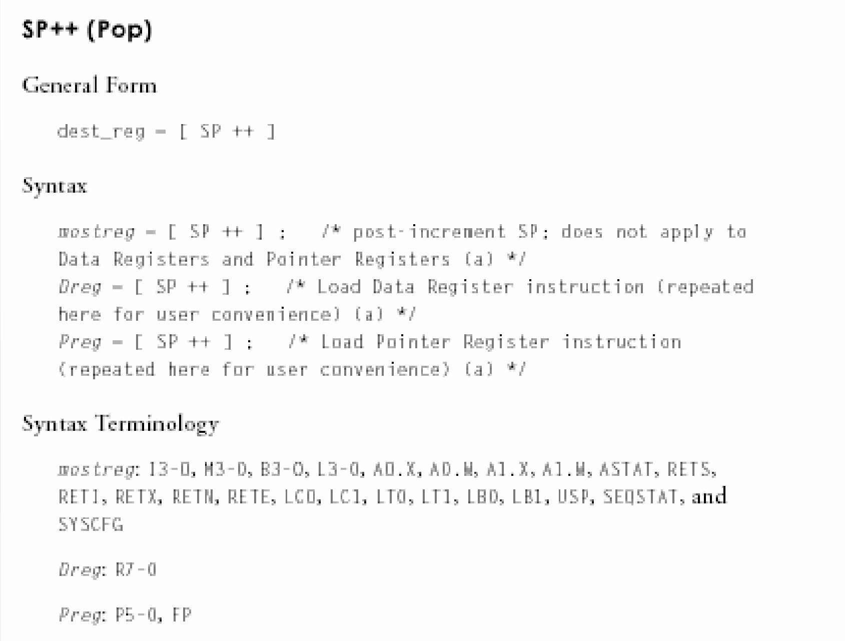
Syntax

```
[ -- SP ] = ( R7 : Dreglim , P5 : Preglim ) ;    /* Dregs and
indexed Pregs (a) */
[ -- SP ] = ( R7 : Dreglim ) ;   /* Dregs, only (a) */
[ -- SP ] = ( P5 : Preglim ) ;   /* indexed Pregs, only (a) */
```



Stack Instructions (Continued)

- **Pop Instruction: dest_reg= [SP++];**
  - The pop instruction loads the contents of the stack indexed by the current stack pointer into a specified register
  - The instruction post-increments the stack pointer to the next occupied location in the stack before concluding
  - Pop multiple instruction allows multiple registers to be popped from the stack with single instruction



SP++ (Pop)

General Form

```
dest_reg = [ SP ++ ]
```

Syntax

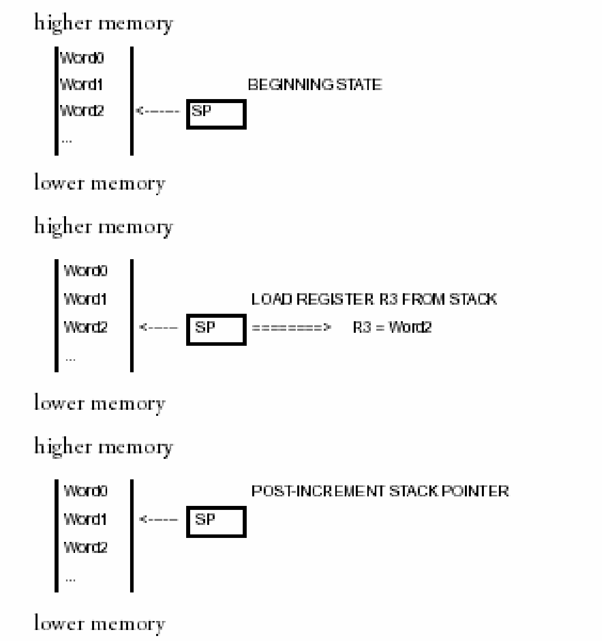
```
mostreg = [ SP ++ ] ;   /* post-increment SP; does not apply to
Data Registers and Pointer Registers (a) */
Dreg = [ SP ++ ] ;   /* Load Data Register instruction (repeated
here for user convenience) (a) */
Preg = [ SP ++ ] ;   /* Load Pointer Register instruction
(repeated here for user convenience) (a) */
```

Syntax Terminology

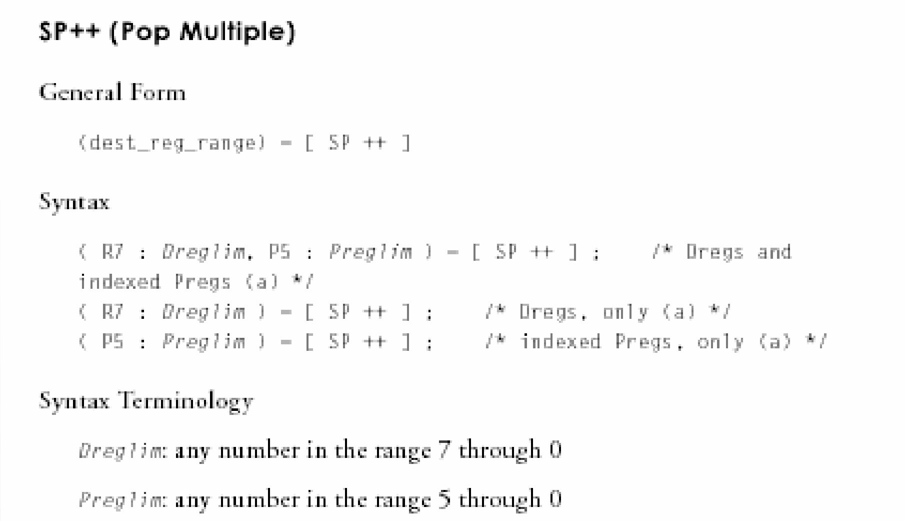
mostreg: I3-0, M3-0, B3-0, L3-0, A0.X, A0.W, A1.X, A1.W, ASTAT, RETS, RETI, RETX, RETN, RETE, LC0, LC1, LT0, LT1, LB0, LB1, USP, SEQSTAT, and SYSCFG

Dreg: R7-0

Preg: P5-0, FP



Stack Instructions (Continued)



SP++ (Pop Multiple)

General Form

```
(dest_reg_range) = [ SP ++ ]
```

Syntax

```
( R7 : Dreglim, P5 : Preglim ) = [ SP ++ ] ;    /* Dregs and
indexed Pregs (a) */
( R7 : Dreglim ) = [ SP ++ ] ;    /* Dregs, only (a) */
( P5 : Preglim ) = [ SP ++ ] ;    /* indexed Pregs, only (a) */
```

Syntax Terminology

`Dreglim`: any number in the range 7 through 0

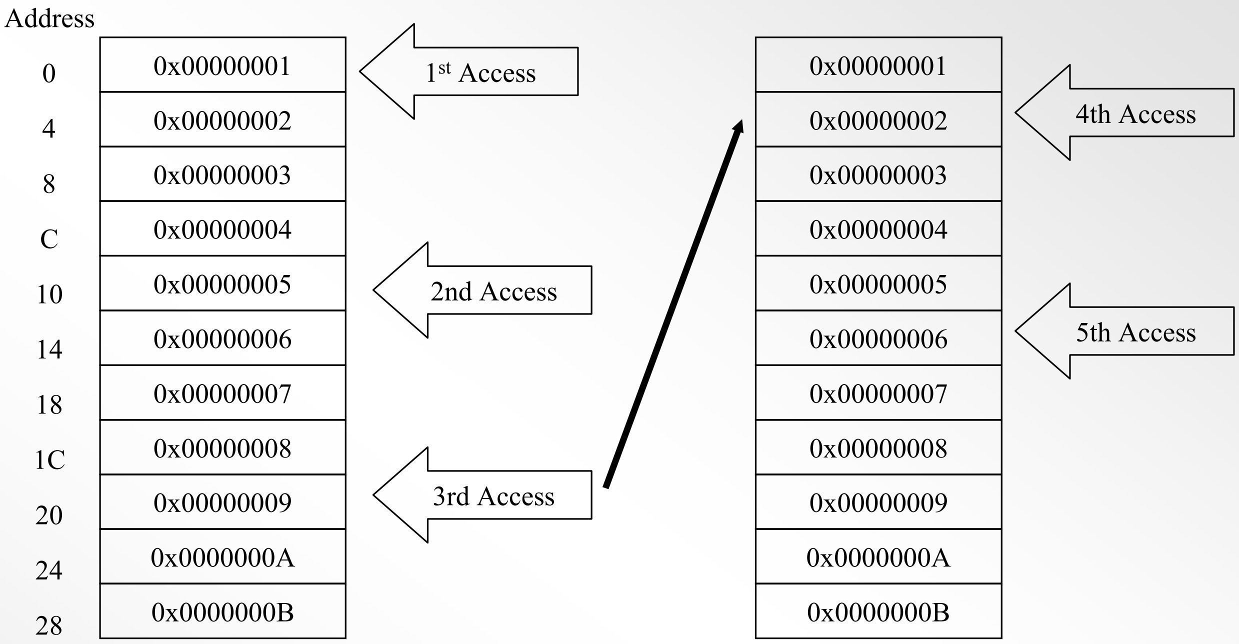
`Preglim`: any number in the range 5 through 0

Circular Buffering

- Only used with Index Registers
  - Index (I) registers holds the address sent out on the address bus.
  - Base (B) registers contain the starting address of the circular buffer.
  - Length (L) registers specify the length of the buffer.
  - Modify (M) registers contain the value (positive or negative) that will be added to the I registers at the end of each memory access.
    - The size of the modify value must be less than or equal to the length of the circular buffer.
- On a Post-Modify access, the address pointed to by the Index register automatically wraps to the circular buffer defined by the corresponding Base address and Length registers.

- Note: L registers must be initialized to 0 when not using circular buffering

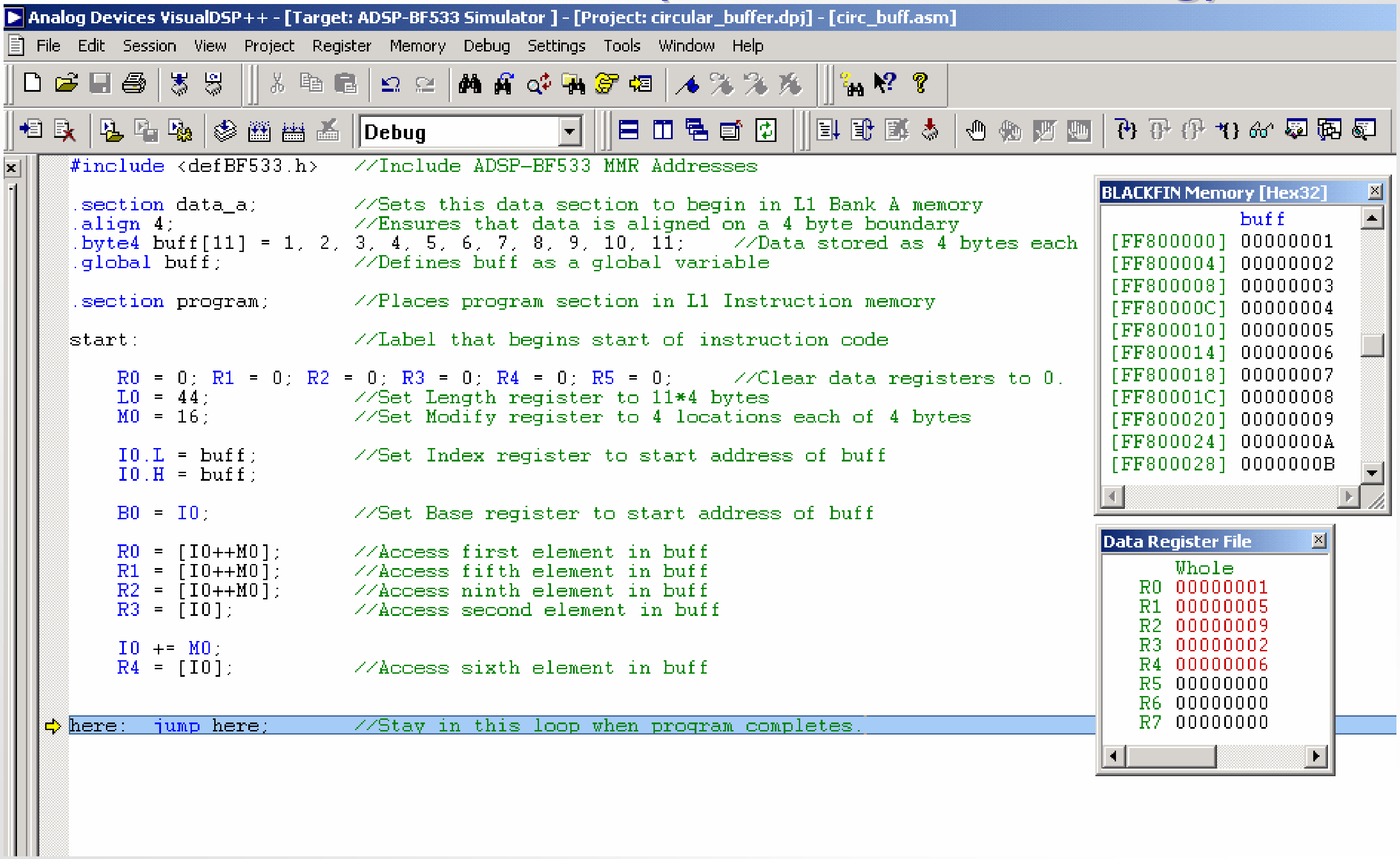
Circular Buffer Example



Address		
0	0x00000001	0x00000001
4	0x00000002	0x00000002
8	0x00000003	0x00000003
C	0x00000004	0x00000004
10	0x00000005	0x00000005
14	0x00000006	0x00000006
18	0x00000007	0x00000007
1C	0x00000008	0x00000008
20	0x00000009	0x00000009
24	0x0000000A	0x0000000A
28	0x0000000B	0x0000000B

- **Base address and Starting Index Address = 0**
- **Buffer length L = 44**
  - **There are 11 data elements and each data element is 4 bytes**
- **Modify value M = 16 (4 elements * 4 bytes/element)**

Circular Buffers (Modulo Addressing)


Analog Devices VisualDSP++ - [Target: ADSP-BF533 Simulator] - [Project: circular_buffer.dpj] - [circ_buff.asm]
File Edit Session View Project Register Memory Debug Settings Tools Window Help
Debug
BLACKFIN Memory [Hex32]
Data Register File

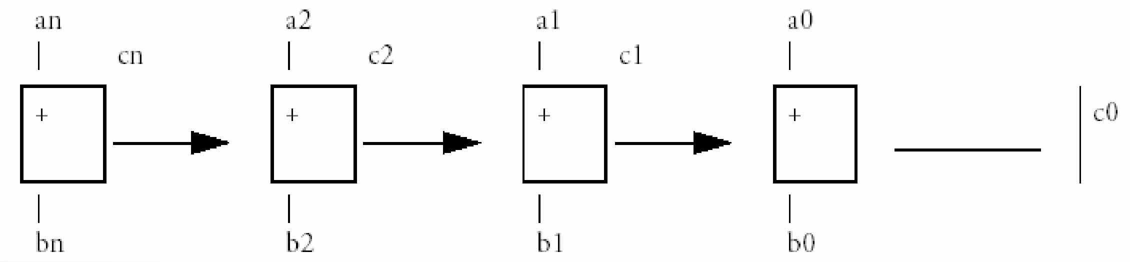
```
#include <defBF533.h>   //Include ADSP-BF533 MMR Addresses

.section data_a;        //Sets this data section to begin in L1 Bank A memory
.align 4;               //Ensures that data is aligned on a 4 byte boundary
.byte4 buff[11] = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11;    //Data stored as 4 bytes each
.global buff;           //Defines buff as a global variable

.section program;       //Places program section in L1 Instruction memory

start:                  //Label that begins start of instruction code

    R0 = 0; R1 = 0; R2 = 0; R3 = 0; R4 = 0; R5 = 0;     //Clear data registers to 0.
    L0 = 44;            //Set Length register to 11*4 bytes
    M0 = 16;            //Set Modify register to 4 locations each of 4 bytes

    I0.L = buff;        //Set Index register to start address of buff
    I0.H = buff;

    B0 = I0;            //Set Base register to start address of buff

    R0 = [I0++M0];      //Access first element in buff
    R1 = [I0++M0];      //Access fifth element in buff
    R2 = [I0++M0];      //Access ninth element in buff
    R3 = [I0];          //Access second element in buff

    I0 += M0;
    R4 = [I0];          //Access sixth element in buff

here:  jump here;       //Stay in this loop when program completes.
```

```
            buff
[FF800000] 00000001
[FF800004] 00000002
[FF800008] 00000003
[FF80000C] 00000004
[FF800010] 00000005
[FF800014] 00000006
[FF800018] 00000007
[FF80001C] 00000008
[FF800020] 00000009
[FF800024] 0000000A
[FF800028] 0000000B
```

```
      Whole
  R0 00000001
  R1 00000005
  R2 00000009
  R3 00000002
  R4 00000006
  R5 00000000
  R6 00000000
  R7 00000000
```

Bit Reverse Addressing

- Bit Reverse Carry Adder (BREV) :
  - When this option is specified, the carry bit propagates from left to right.
  - Used to support operand addressing for FFT, DCT, and DFT algorithms



- Only supported with Modify-Increment instruction
  - Preg += Preg (BREV);
  - Ireg += Mreg (BREV);

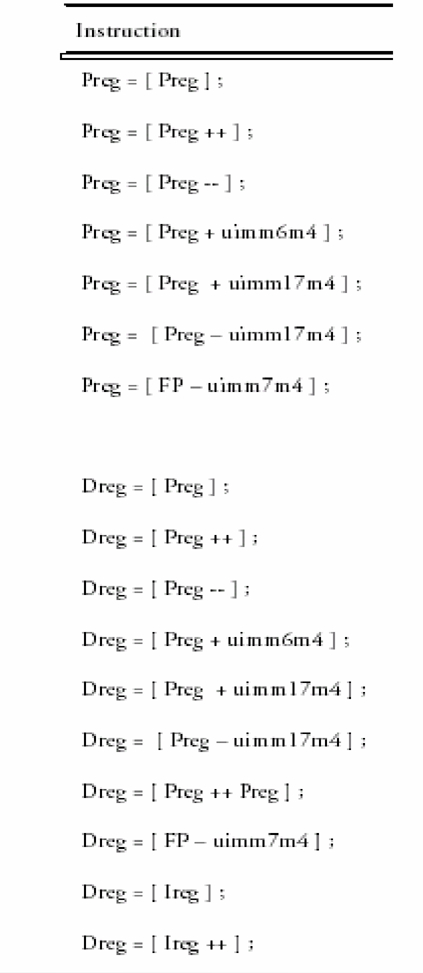
- With the Index Register version of this instruction, circular buffering is disabled

  I0 += M0 (brev);

  Pointer register example

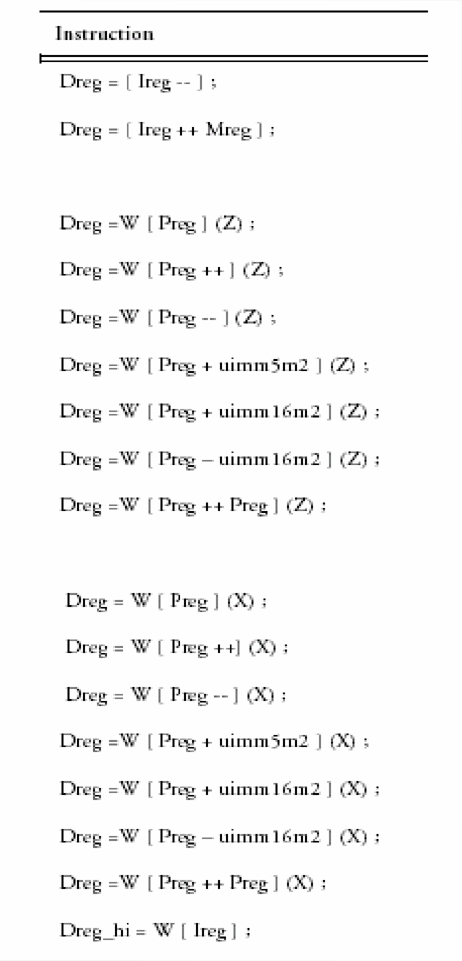
  P3 += P0 (brev);

DAG Instructions 1 of 6



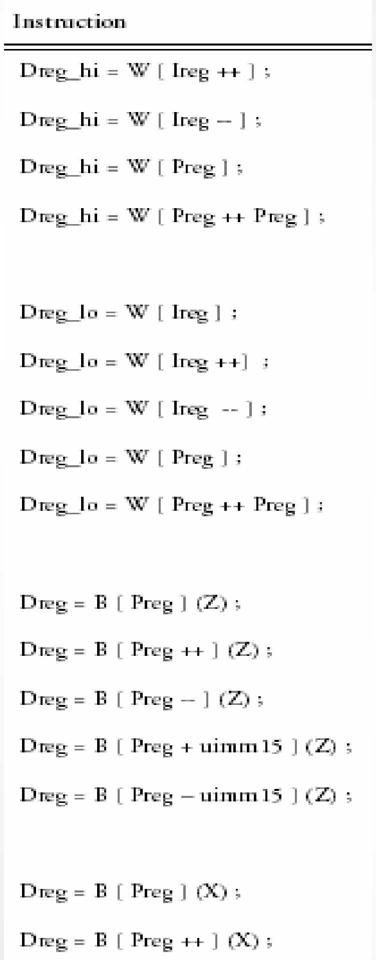
Instruction
Preg = [Preg] ;
Preg = [Preg ++] ;
Preg = [Preg --] ;
Preg = [Preg + uimm6m4] ;
Preg = [Preg + uimm17m4] ;
Preg = [Preg – uimm17m4] ;
Preg = [FP – uimm7m4] ;
Dreg = [Preg] ;
Dreg = [Preg ++] ;
Dreg = [Preg --] ;
Dreg = [Preg + uimm6m4] ;
Dreg = [Preg + uimm17m4] ;
Dreg = [Preg – uimm17m4] ;
Dreg = [Preg ++ Preg] ;
Dreg = [FP – uimm7m4] ;
Dreg = [Ireg] ;
Dreg = [Ireg ++] ;

DAG Instructions 2 of 6



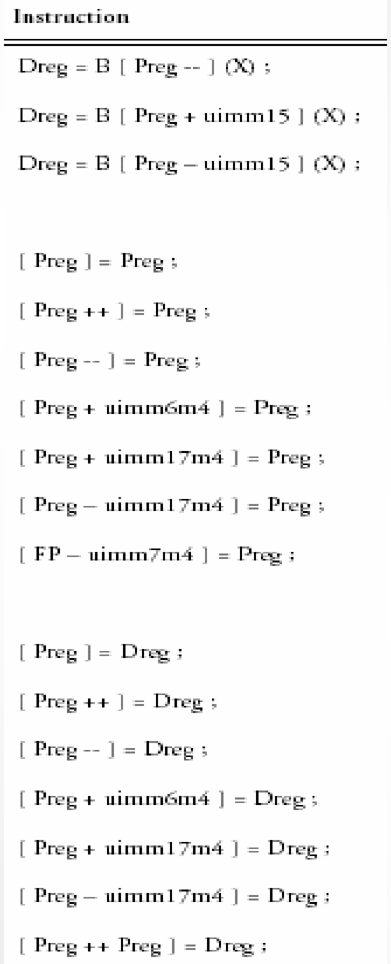
Instruction
Dreg = [Ireg --] ;
Dreg = [Ireg ++ Mreg] ;
Dreg =W [Preg] (Z) ;
Dreg =W [Preg ++] (Z) ;
Dreg =W [Preg --] (Z) ;
Dreg =W [Preg + uimm5m2] (Z) ;
Dreg =W [Preg + uimm16m2] (Z) ;
Dreg =W [Preg – uimm16m2] (Z) ;
Dreg =W [Preg ++ Preg] (Z) ;
Dreg = W [Preg] (X) ;
Dreg = W [Preg ++] (X) ;
Dreg = W [Preg --] (X) ;
Dreg =W [Preg + uimm5m2] (X) ;
Dreg =W [Preg + uimm16m2] (X) ;
Dreg =W [Preg – uimm16m2] (X) ;
Dreg =W [Preg ++ Preg] (X) ;
Dreg_hi = W [Ireg] ;

DAG Instructions 3 of 6



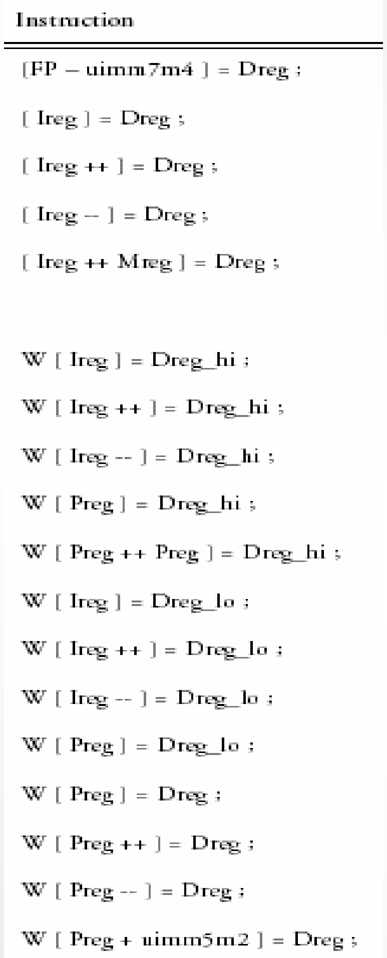
Instruction
Dreg_hi = W [Ireg ++] ;
Dreg_hi = W [Ireg --] ;
Dreg_hi = W [Preg] ;
Dreg_hi = W [Preg ++ Preg] ;
Dreg_lo = W [Ireg] ;
Dreg_lo = W [Ireg ++] ;
Dreg_lo = W [Ireg --] ;
Dreg_lo = W [Preg] ;
Dreg_lo = W [Preg ++ Preg] ;
Dreg = B [Preg] (Z) ;
Dreg = B [Preg ++] (Z) ;
Dreg = B [Preg --] (Z) ;
Dreg = B [Preg + uimm15] (Z) ;
Dreg = B [Preg – uimm15] (Z) ;
Dreg = B [Preg] (X) ;
Dreg = B [Preg ++] (X) ;

DAG Instructions 4 of 6



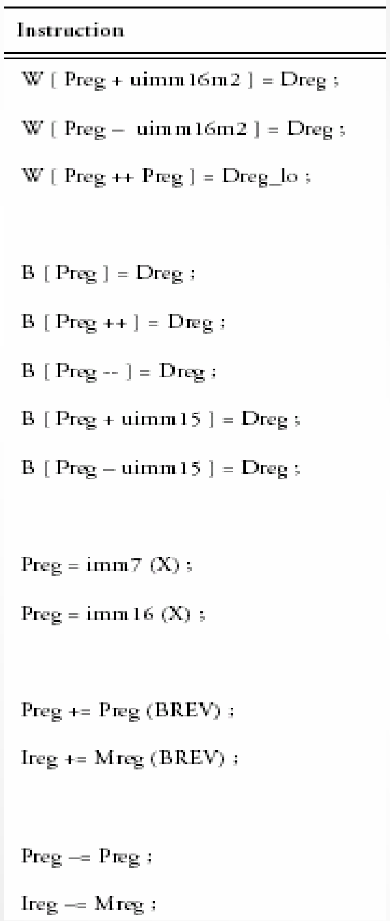
Instruction
Dreg = B [Preg --] (X) ;
Dreg = B [Preg + uimm15] (X) ;
Dreg = B [Preg – uimm15] (X) ;
[Preg] = Preg ;
[Preg ++] = Preg ;
[Preg --] = Preg ;
[Preg + uimm6m4] = Preg ;
[Preg + uimm17m4] = Preg ;
[Preg – uimm17m4] = Preg ;
[FP – uimm7m4] = Preg ;
[Preg] = Dreg ;
[Preg ++] = Dreg ;
[Preg --] = Dreg ;
[Preg + uimm6m4] = Dreg ;
[Preg + uimm17m4] = Dreg ;
[Preg – uimm17m4] = Dreg ;
[Preg ++ Preg] = Dreg ;

DAG Instructions 5 of 6



Instruction
[FP – uimm7m4] = Dreg ;
[Ireg] = Dreg ;
[Ireg ++] = Dreg ;
[Ireg --] = Dreg ;
[Ireg ++ Mreg] = Dreg ;
W [Ireg] = Dreg_hi ;
W [Ireg ++] = Dreg_hi ;
W [Ireg --] = Dreg_hi ;
W [Preg] = Dreg_hi ;
W [Preg ++ Preg] = Dreg_hi ;
W [Ireg] = Dreg_lo ;
W [Ireg ++] = Dreg_lo ;
W [Ireg --] = Dreg_lo ;
W [Preg] = Dreg_lo ;
W [Preg] = Dreg ;
W [Preg ++] = Dreg ;
W [Preg --] = Dreg ;
W [Preg + uimm5m2] = Dreg ;

DAG Instructions 6 of 6



Instruction
W [Preg + uimm16m2] = Dreg ;
W [Preg – uimm16m2] = Dreg ;
W [Preg ++ Preg] = Dreg_lo ;
B [Preg] = Dreg ;
B [Preg ++] = Dreg ;
B [Preg --] = Dreg ;
B [Preg + uimm15] = Dreg ;
B [Preg – uimm15] = Dreg ;
Preg = imm7 (X) ;
Preg = imm16 (X) ;
Preg += Preg (BREV) ;
Ireg += Mreg (BREV) ;
Preg –= Preg ;
Ireg –= Mreg ;

Data Address Generator Exercise

Lab 6

Reference Material

Addressing

Memory Addressing

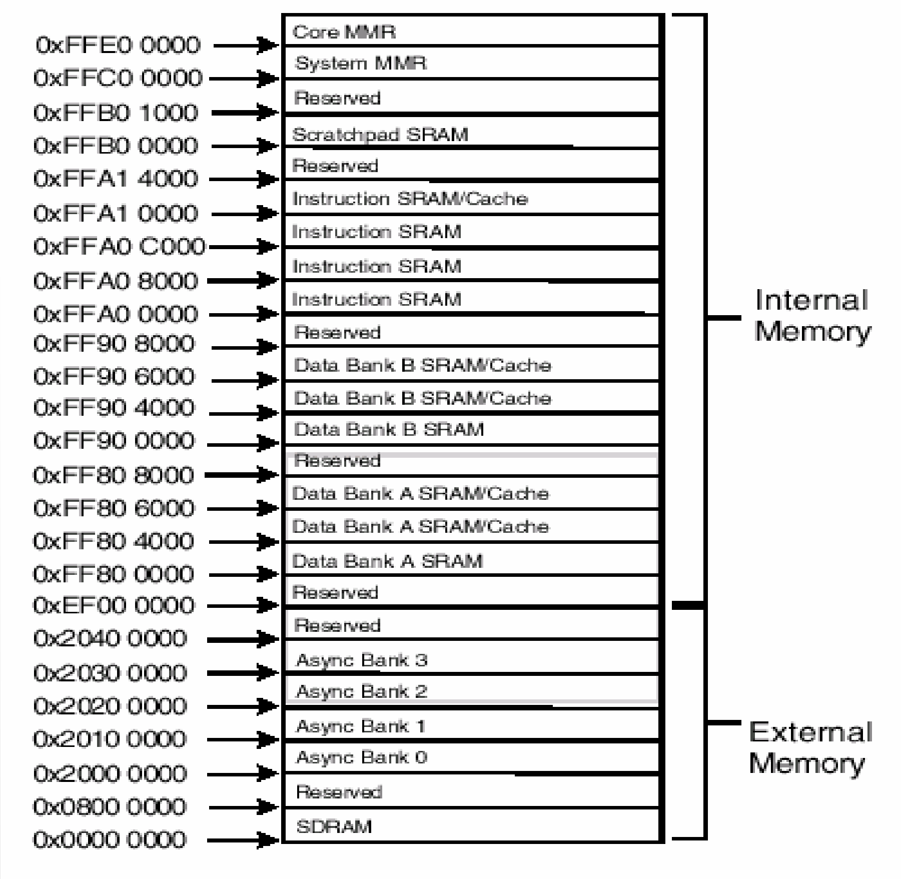
There are 4 units that generate addresses for memory accesses

- **Two Data Address Generators (DAG0 and DAG1)**
  - **Generates addresses for data fetches**
  - **User programs addresses for data fetches**
    - **Extremely flexible data addressing**
    - **DAG0 and DAG1 can be used in the same instruction (to perform a dual data fetch**
- **The Program Sequencer**
  - **Generates addresses for instruction fetches**
  - **Automatically controlled from program**
- **The DMA controller**
  - **Generates addresses for DMA transfers**
    - **User defines addresses for source and destination of transfer**
    - **DMA controller automatically generates addresses to complete DMA transfer**

BF533 Memory Addressing

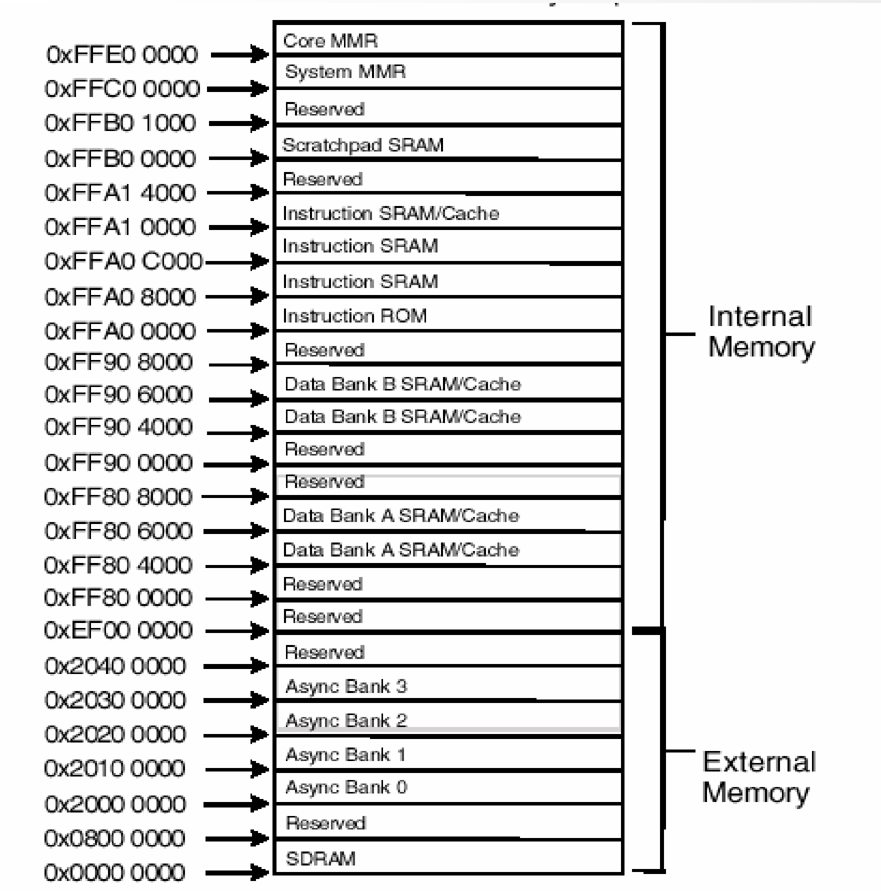
- All memory on the BF533 resides in a 32 bit linear address range
- Different types of memory are mapped into address spaces within the linear address range (i.e. the memory map)
  - The BF533 provides decoded bank selects for the address spaces
    - For Example, to access Async Bank 0, you simply fetch from an address in the range of 0x2000 0000 and 0x200F FFFF. The BF533 will automatically assert the corresponding bank Select pin and drive the EPB address lines.
- Each of the BF533 family members has it's own memory map
  - You use the memory map to define (in the LDF) all the available memory in your system

ADSP-BF533 Memory Map



Address	Region
0xFFE0 0000	Core MMR
0xFFC0 0000	System MMR
0xFFB0 1000	Reserved
0xFFB0 0000	Scratchpad SRAM
0xFFA1 4000	Reserved
0xFFA1 0000	Instruction SRAM/Cache
0xFFA0 C000	Instruction SRAM
0xFFA0 8000	Instruction SRAM
0xFFA0 0000	Instruction SRAM
0xFF90 8000	Reserved
0xFF90 6000	Data Bank B SRAM/Cache
0xFF90 4000	Data Bank B SRAM/Cache
0xFF90 0000	Data Bank B SRAM
0xFF80 8000	Reserved
0xFF80 6000	Data Bank A SRAM/Cache
0xFF80 4000	Data Bank A SRAM/Cache
0xFF80 0000	Data Bank A SRAM
0xEF00 0000	Reserved
0x2040 0000	Reserved
0x2030 0000	Async Bank 3
0x2020 0000	Async Bank 2
0x2010 0000	Async Bank 1
0x2000 0000	Async Bank 0
0x0800 0000	Reserved
0x0000 0000	SDRAM

ADSP-BF532 Memory Map



0xFFE0 0000
0xFFC0 0000
0xFFB0 1000
0xFFB0 0000
0xFFA1 4000
0xFFA1 0000
0xFFA0 C000
0xFFA0 8000
0xFFA0 0000
0xFF90 8000
0xFF90 6000
0xFF90 4000
0xFF90 0000
0xFF80 8000
0xFF80 6000
0xFF80 4000
0xFF80 0000
0xEF00 0000
0x2040 0000
0x2030 0000
0x2020 0000
0x2010 0000
0x2000 0000
0x0800 0000
0x0000 0000

Core MMR
System MMR
Reserved
Scratchpad SRAM
Reserved
Instruction SRAM/Cache
Instruction SRAM
Instruction SRAM
Instruction ROM
Reserved
Data Bank B SRAM/Cache
Data Bank B SRAM/Cache
Reserved
Reserved
Data Bank A SRAM/Cache
Data Bank A SRAM/Cache
Reserved
Reserved
Reserved
Async Bank 3
Async Bank 2
Async Bank 1
Async Bank 0
Reserved
SDRAM

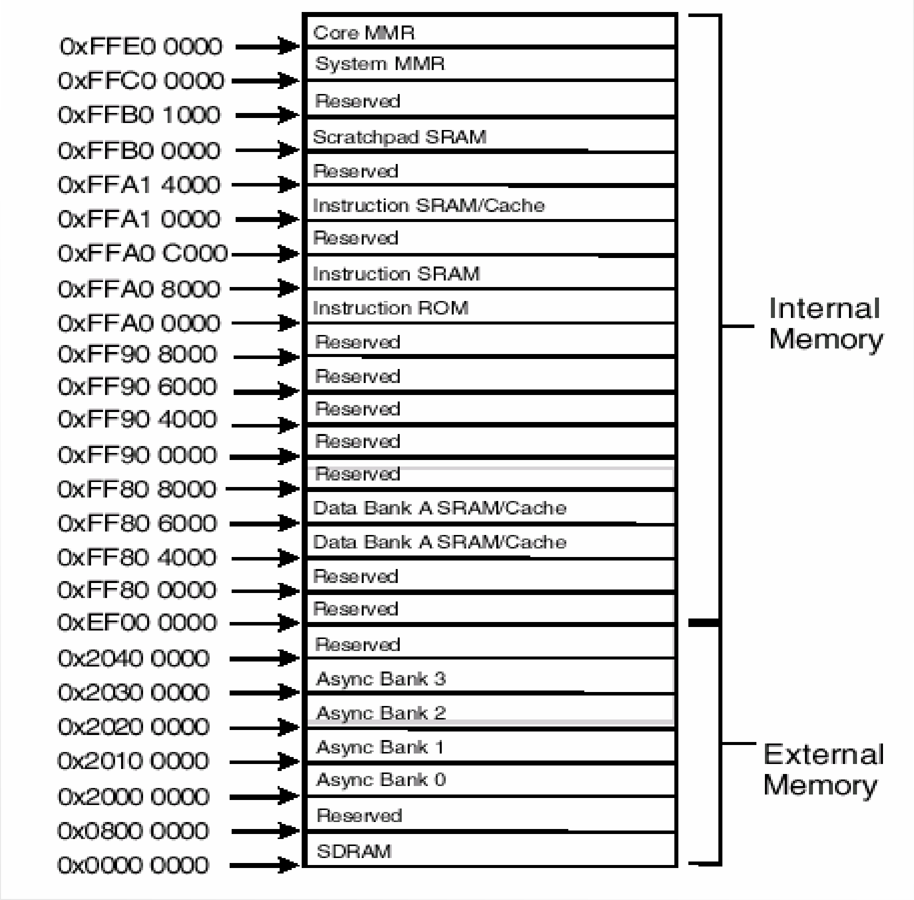
Internal Memory

External Memory

KAZTEK ENGINEERING

ANALOG DEVICES

ADSP-BF531 Memory Map



0xFFE0 0000	Core MMR
0xFFC0 0000	System MMR
0xFFB0 1000	Reserved
0xFFB0 0000	Scratchpad SRAM
0xFFA1 4000	Reserved
0xFFA1 0000	Instruction SRAM/Cache
0xFFA0 C000	Reserved
0xFFA0 8000	Instruction SRAM
0xFFA0 0000	Instruction ROM
0xFF90 8000	Reserved
0xFF90 6000	Reserved
0xFF90 4000	Reserved
0xFF90 0000	Reserved
0xFF80 8000	Reserved
0xFF80 6000	Data Bank A SRAM/Cache
0xFF80 4000	Data Bank A SRAM/Cache
0xFF80 0000	Reserved
0xEF00 0000	Reserved
0x2040 0000	Reserved
0x2030 0000	Async Bank 3
0x2020 0000	Async Bank 2
0x2010 0000	Async Bank 1
0x2000 0000	Async Bank 0
0x0800 0000	Reserved
0x0000 0000	SDRAM

Memory Addressing

- All memory for the BF533 family is addressed in bytes
  - A single byte requires one memory location
  - A 16 bit word requires 2 memory locations
  - A 32 bit word requires 4 memory locations
  - A 64 bit word requires 8 memory locations
- The Program Sequencer fetches Instructions and will automatically increment the address correctly
- The programmer is responsible for incrementing the address for data fetches
  - This is handled through instruction syntax

Indirect Address Examples

- **DSP Addressing Modes – Index Registers**
  - Indirect, auto-increment, auto-decrement for 16/32-bit loads/stores

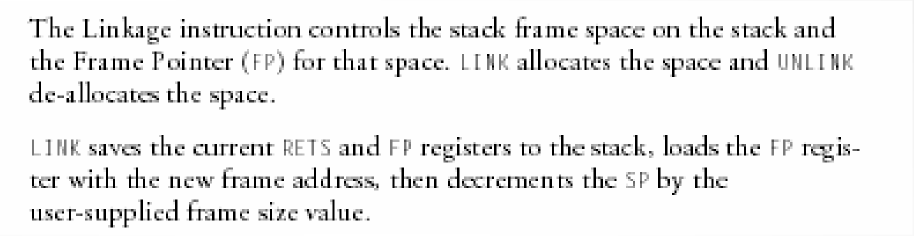
```
R0 = [I2];              // Loads R0 with 32-bit value that address I2 points to
R0.H = W[I0++];         // Loads R0.H with 16 bit value that address I0 points to
                        // W implies a 2 byte, post modify increment
[I2--] = R0;            // Stores R0 to address that I2 points to
                        // Address of I2 is decremented by 4 bytes after store
R1 = [I2 ++ M1];        // Post-modify with non-unity stride for 32-bit loads/stores
```

- **General Addressing Modes – Pointer Registers**
  - Indirect, auto-increment, auto-decrement, indexed with immediate offset for 8/16/32-bit loads/stores

```
R3 = [P0];              // Loads R0 with 32-bit value that address P0 points to
R7 = W[P1++] (z);       // Loads R7.L with 16 bit value that address P1 points to
                        // W implies a 2 byte, post modify increment
R2 = B[P2--] (Z);       // B implies a 1 byte decrement
R0.H = W[P1++P2];       //Post-modify with non-unity stride for 16/32-bit loads/stores
```

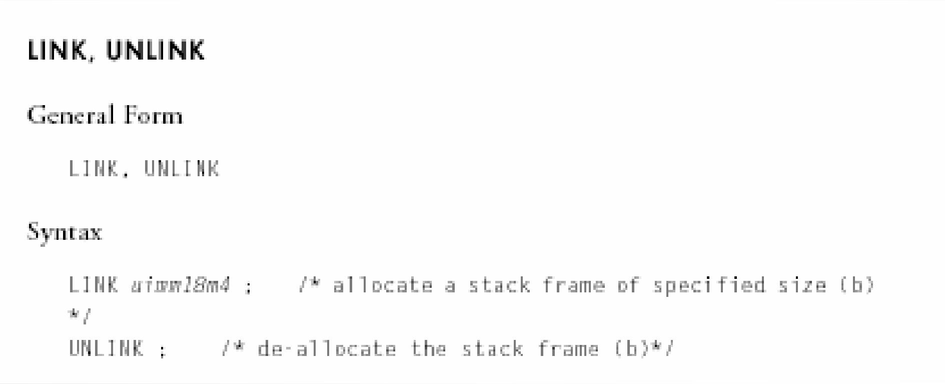
Additional Instructions

- **Link/Unlink:**



The Linkage instruction controls the stack frame space on the stack and the Frame Pointer (FP) for that space. LINK allocates the space and UNLINK de-allocates the space.

LINK saves the current RETS and FP registers to the stack, loads the FP register with the new frame address, then decrements the SP by the user-supplied frame size value.



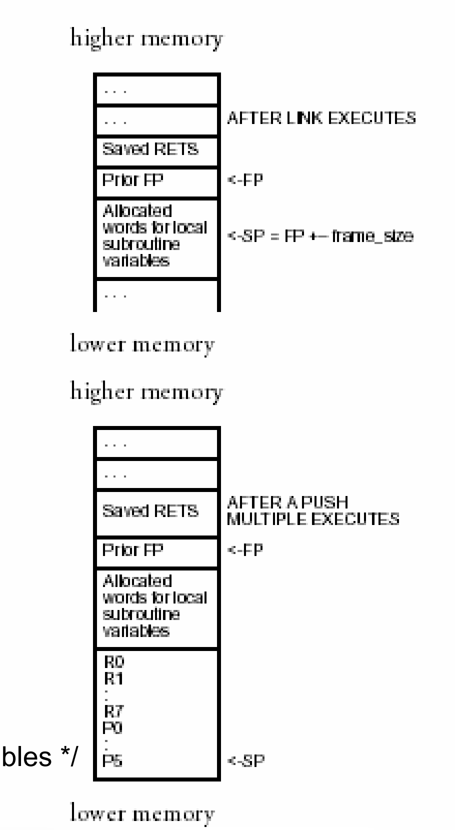
LINK, UNLINK

General Form

```
   LINK, UNLINK
```

Syntax

```
   LINK uimm18m4 ;    /* allocate a stack frame of specified size (b)
   */
   UNLINK ;    /* de-allocate the stack frame (b)*/
```



Example:

```
link 8;           /* establish frame with 2 words (8 bytes) allocated for local variables */
[--sp] = (r7:0, p5:0);          /* Save D- and P-registers */
(r7:0, p5:0) = [sp++];          /* restore D- and P- registers */
unlink;                         /* close the frame */
```