The Blackfin Instruction Set Architecture

Mark McDermott

With help from our friends at Analog Devices, Inc.

Lecture Outline

- Architecture overview
- Pipeline
- Register Set
- Addressing modes
- Instruction Set Encoding
- Instruction Set Operations
  - Program Flow Operation
  - Load-Store Operations
  - Move Operations
  - Stack Operation
  - Control Code Operations
  - Logical Operations
  - Arithmetic Operations
  - Bit, Shift and Rotate Operation
  - External Event Management
  - Video Pixel Operations
  - Vector Operations

Blackfin Architecture Overview

- Two 16-bit Multipliers
  - Performs dual MACs(multiply-accumulates) when used with ALUs
- Two 32/40-bit ALUs
- Four 8-bit Video ALUs
- Barrel Shifter
  - Performs shifts, rotates, bit operations
- There are 8x 32-bit registers in the data register file.
  - Used to hold 32-bit vales or packed 16-bit
- There are 2x 40-bit accumulators.
  - Typically used for MAC operations
- The addressing unit generates addresses for data fetches.
  - Two DAG (Data Address Generator) arithmetic units enable generation of independent 32-bit wide addresses that can reach anywhere within the Blackfin memory space.
  - Up to two fetches can occur at the same time.
- Variable Length Instructions
  - 16-bit Instructions
  - 32-bit Instructions
  - Multi-Issue, 64-bit Instructions
- 10 Stage Pipeline (each stage takes 1 cycle unless there are dependencies)

Blackfin Pipeline Stages

The Blackfin has 10 pipeline stages

Pipeline Stage	Description
Instruction Fetch 1 (IF1)	Issue instruction address to IAB bus, start compare tag of instruction cache
Instruction Fetch 2 (IF2)	Wait for instruction data
Instruction Fetch 3 (IF3)	Read from IDB bus and align instruction
Instruction Decode (DEC)	Decode instructions
Address Calculation (AC)	Calculation of data addresses and branch target address
Data Fetch 1 (DF1)	Issue data address to DA0 and DA1 bus, start compare tag of data cache
Data Fetch 2 (DF2)	Read Register files
Execute 1 (EX1)	Read data from LD0 and LD1 bus, start multiply and video instructions
Execute 2 (EX2)	Execute/Complete instructions (shift, add, logic, etc.)
Write Back (WB)	Writes back to register files, SD bus, and pointer updates (also referred to as the “commit” stage)

Blackfin Pipeline Stages (IF1, IF2, IF3, DEC, AC, DF1, DF2, EX1, EX2, WB)

AC
DF1

Memory Access

ADDRESS ARITHMETIC UNIT

SP
FP
P5
P4
P3
P2
P1
P0

I3	L3	B3	M3
I2	L2	B2	M2
I1	L1	B1	M1
I0	L0	B0	M0

DAG0
DAG1

IF1,IF2,IF3,DEC

SEQUENCER
ALIGN
DECODE
LOOP BUFFER
CONTROL UNIT

Instruction Sequencer

Register Access

R7
R6
R5
R4
R3
R2
R1
R0

DF2
WB

EX1

8
16
8
8
16
8

BARREL SHIFTER
40
40
EX2
ACC 0
ACC 1

Arithmetic & Logical

DATA ARITHMETIC UNIT

Blackfin Core Registers

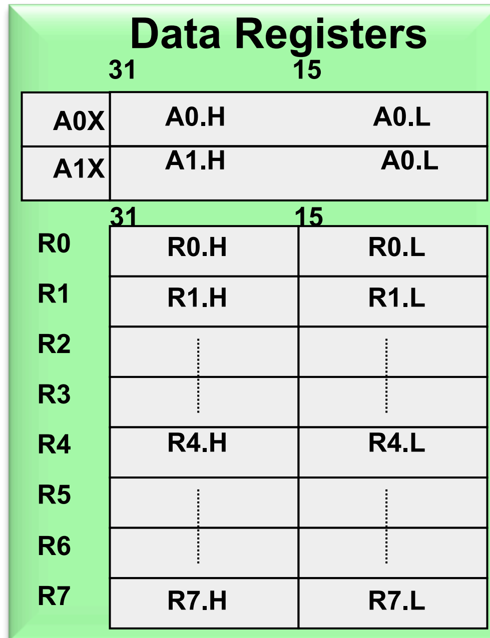
Register Name	Mnemonic
Data Registers	R0-R7
Accumulator Registers	A0, A1
Pointer Registers	P0-P5, FP, SP,USP
DAG Registers	I0-I3, M0-M3, B0-B3, L0-L3
Cycle Counters	Cycles, cycles2
Program Sequencer	SEQSTAT
System Configuration Register	SYSCFG
Loop Registers	LT[1:0], LB[1:0], LC[1:0]
Interrupt Return Registers	RETI, RETX, RETN, RETE

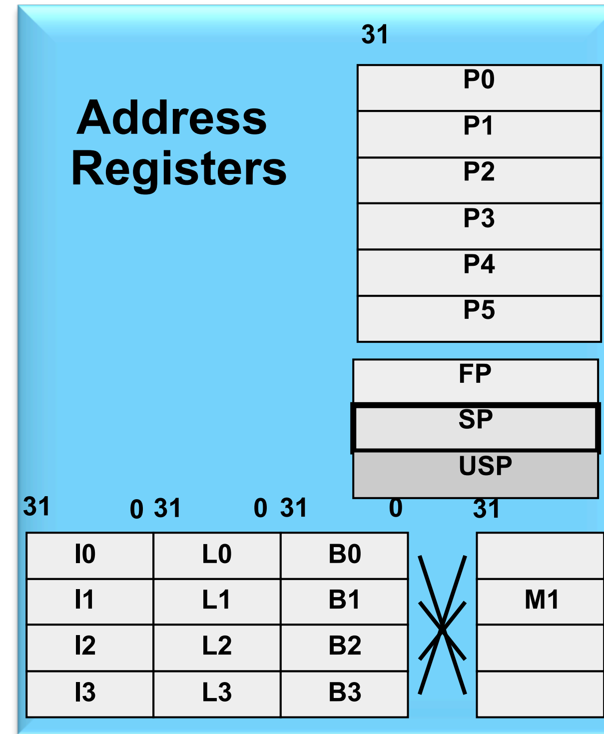
Example:
R0 = SYSCFG; // Load data register with contents of SYSCFG register

Blackfin Core Registers (cont)

Data Registers:
R0-R7 are referred to as "dreg"

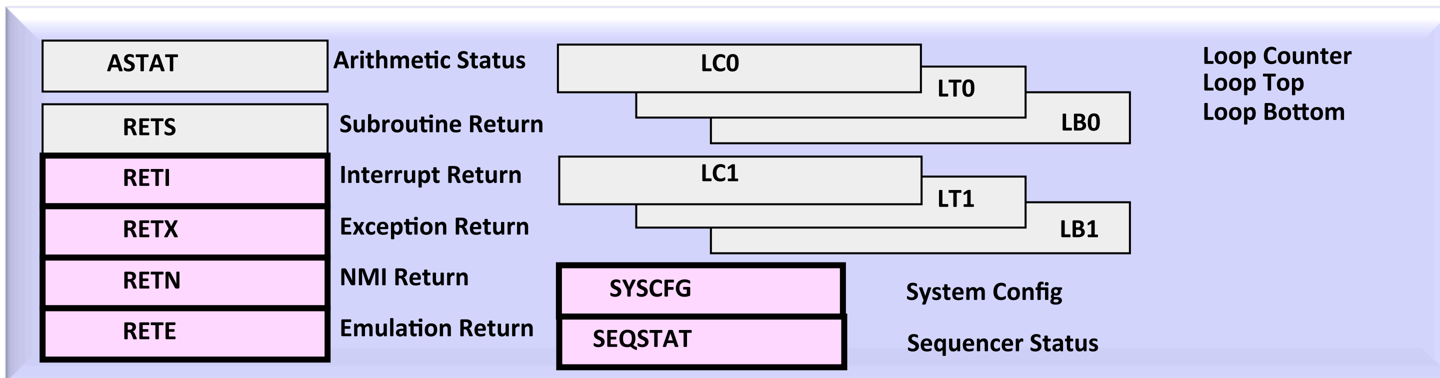
_lo refers to .L
and
_hi refers to .H





Pointer Registers:
P0-P5 are referred to as "preg"

Index Registers:
I0-I3 are referred to as "ireg"



Shaded registers only accessible in Supervisor mode

Data and Accumulator Registers

- The Data Register File consists of eight registers.
  - Each 32 bits wide.
  - Each register may be viewed as a pair of independent 16-bit registers.
  - Each is denoted as the low half or high half. Thus the 32-bit register R0 may be regarded as two independent register halves, R0.L and R0.H.

- The processor has two dedicated, 40-bit accumulator registers.
  - Each can be referred to as its 16-bit low half (An.L) or high half (An.H) plus its 8-bit extension (An.X).
  - Each can also be referred to as a 32-bit register (An.W) consisting of the lower 32 bits, or as a complete 40-bit result register (An).

Data Registers

	31	15
A0X	A0.H	A0.L
A1X	A1.H	A0.L

	31	15
R0	R0.H	R0.L
R1	R1.H	R1.L
R2		
R3		
R4	R4.H	R4.L
R5		
R6		
R7	R7.H	R7.L

BLACKFIN

Data Address Generator (DAG) Registers

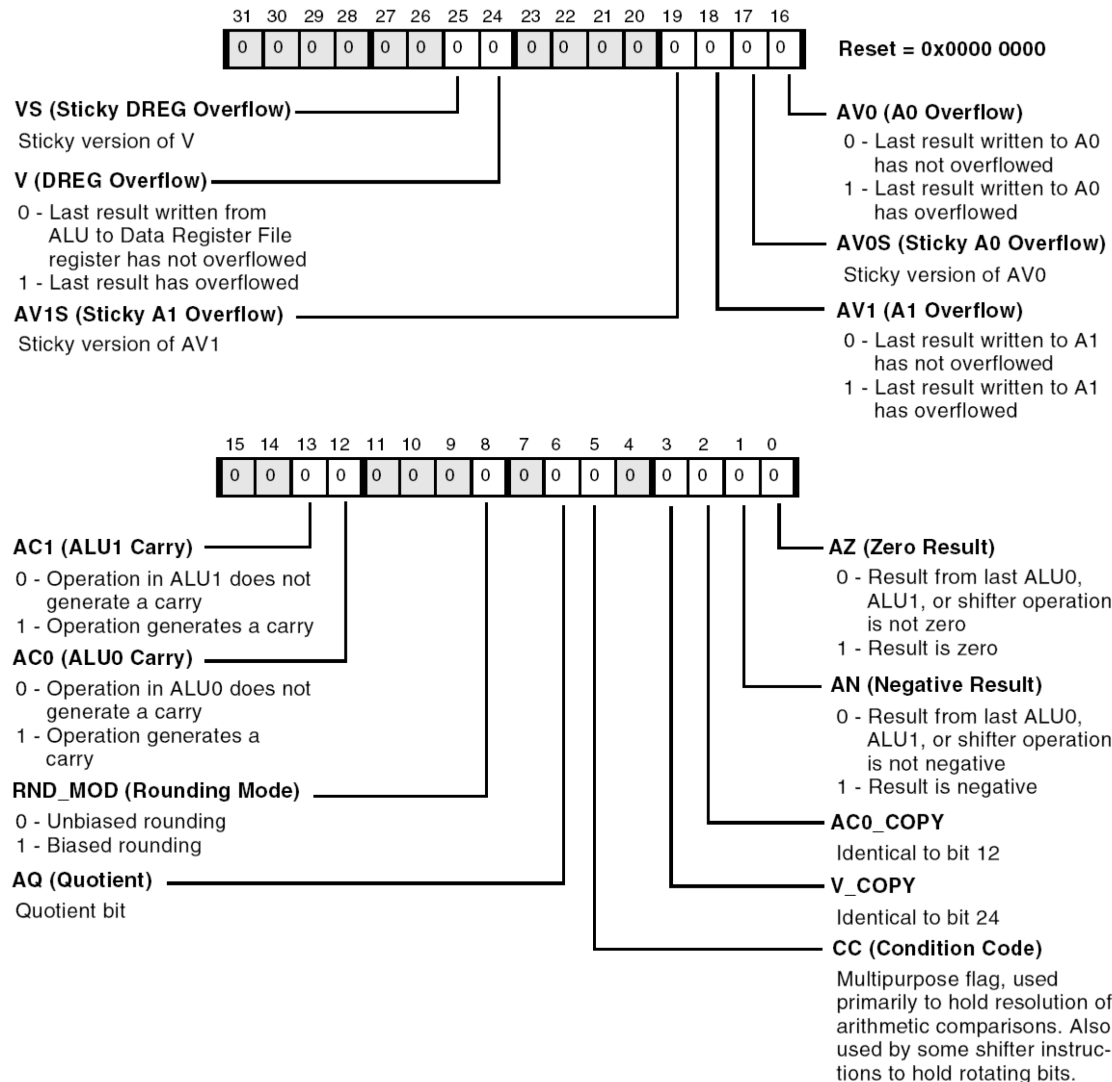
- **Blackfin instructions primarily use the Data Address Generator (DAG) register set for addressing. The DAG register set consists of these registers:**
  - I[3:0] contain index addresses
  - M[3:0] contain modify values
  - B[3:0] contain base addresses
  - L[3:0] contain length values

- **All DAG registers are 32 bits wide.**
  - The I (Index) registers and B (Base) registers always contain addresses of 8-bit bytes in memory. The Index registers contain an effective address.
  - The M (Modify) registers contain an offset value that is added to one of the Index registers or subtracted from it.
  - The B (Base) and L (Length) registers define circular buffers. The B register contains the starting address of a buffer, and the L register contains the length in bytes. Each L and B register pair is associated with the corresponding I register.
    - For example, L0 and B0 are always associated with I0. However, any M register may be associated with any I register. For example, I0 may be modified by M3.

Pointer Register

- The general-purpose Address Pointer registers, called P-registers, are organized as:
  - 6-entry, P-register files P[5:0]
  - Frame Pointers (FP) used to point to the current procedure's activation record
  - Stack Pointer registers (SP) used to point to the last used location on the runtime stack
- P-registers are 32 bits wide. P-registers are primarily used for address calculations, but they may also be used for general integer arithmetic with a limited set of arithmetic operations
  - e.g. to maintain counters.
  - NOTE: P-register arithmetic does not affect the Arithmetic Status (ASTAT) register status flags.

ASTAT Register

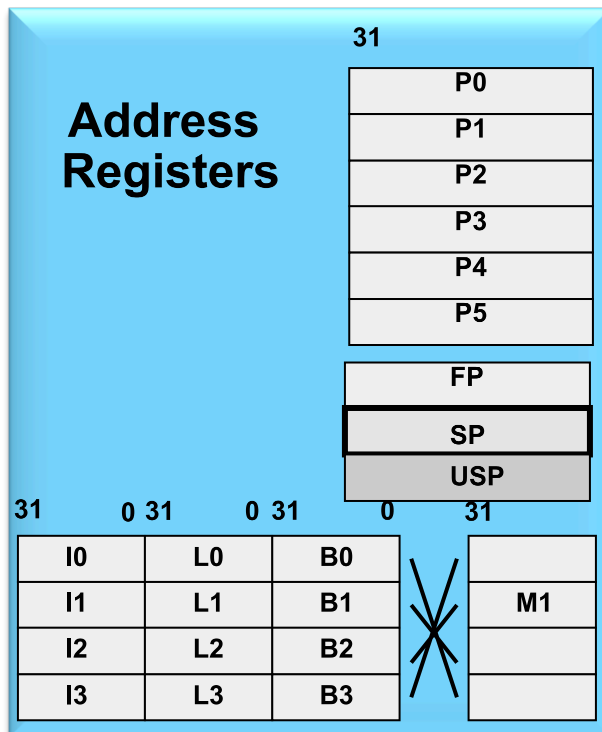


The processor updates the status bits in ASTAT, indicating the status of the most recent ALU, multiplier, or shifter operation.

Addressing Modes

Address Registers

- **One set of general purpose pointer registers**
  - P0-P5, SP and FP
- **One set of DSP addressing index registers**
  - I0-I3, B0-B3, L0-L3, M0-M3
- **All addresses are byte addresses**



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

Memory Addressing

- All BF family members memory 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

Addressing Modes

- **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

Memory Addressing Modes

Mode	Example	Meaning	Use
Displacement	R0 = [P0 + 100];	$4 <- M[$P0 + 100]	Arrays w/constant offset
Register Indirect	R0 = [P0];	$4 <- M[$P0]	You have pointer
Autoincrement	R0 = [P0++];	$4 <- M[P0]; P0++;	Stepping through array in loop
Autodecrement	R0 = [P0--];	$4 <- M[P0]; P0--;	Stepping through array in loop
Post-increment index	R0=[P1++P2];	$4 <- M[I0]; I0 += M0;	Indexing arrays of large elements

Various combinations of pointer regs or index regs are allowed.

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_lo_hi = B[preg];   //loads 8-bit value pointed to by preg and loads into
                        //  hi or lo half of 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:

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

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.L = W[P1++];         // 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
```

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 */
  ```

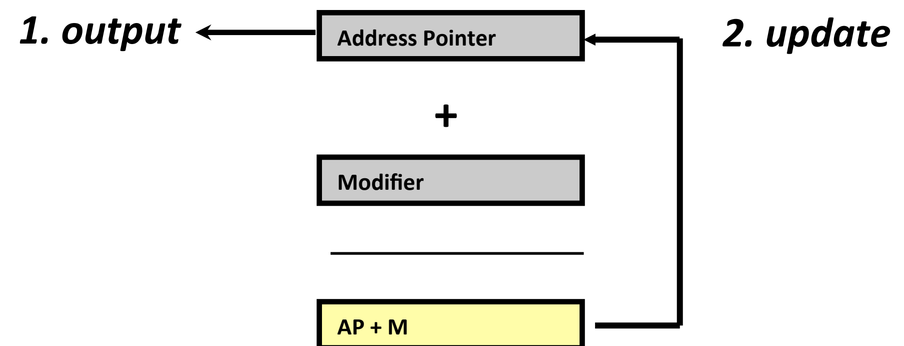
  16-bit accesses

  ```
  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 */
  ```

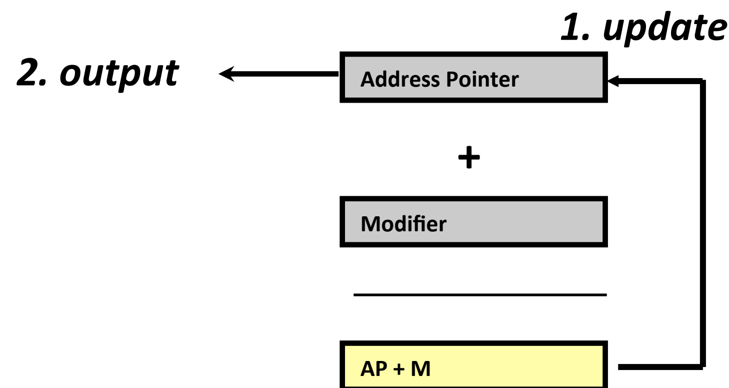
- **Analogous store instructions exist**



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 */
```

Address Pointer

+

Modifier

AP + M

Output only
(No Update)

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. */
  ```

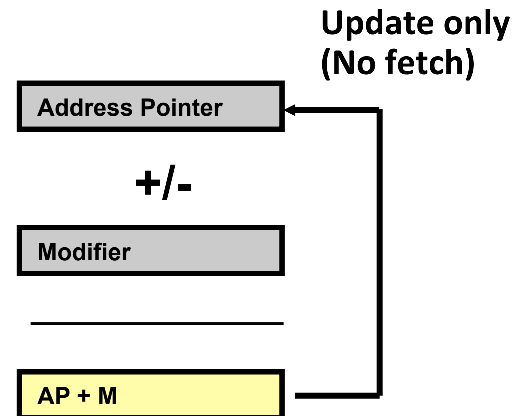
  Index registers use an M-register as modifier

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

- **Modify-Decrement supported as well as Modify-Increment**

  Example

  ```
  P0 -= P4;
  ```

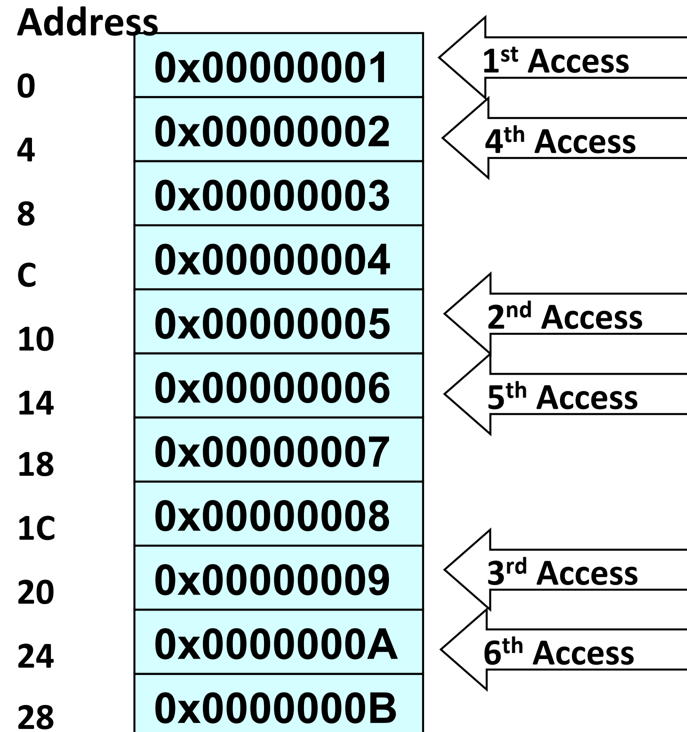


Circular Buffering

- **Only used with Index Registers**
  - Index (I) registers point to the address sent out on the address bus.
  - Base (B) registers contain the starting address of the buffer. The circular buffer wraps around to the address contained in the B register.
  - Length (L) registers specify the length of the buffer.
  - Modify (M) registers contain the post-modify value in which to modify I registers at the end of each memory access.
    - The size of the modify value must be less than or equal to the L register of the circular buffer.

Note: L registers must be initialized to 0 when circular buffering is not used!!!

Circular Buffer Example

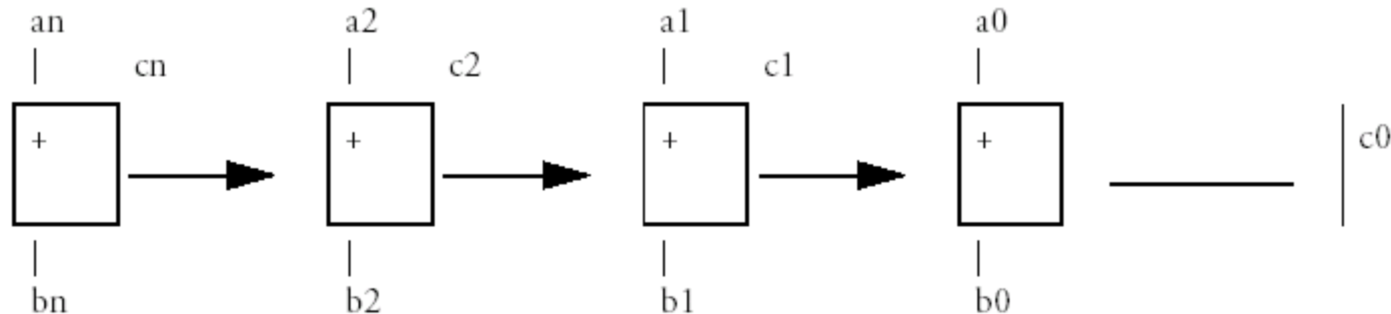


- **Base address and Starting Index Address = 0**
- **Buffer length L = 44 (bytes)**
  - **There are 11 data elements and each data element is 4 bytes**
- **Modify value M = 16 (4 elements * 4 bytes/element)**

Bit Reverse Addressing

- **Bit Reverse Carry Adder (BREV) : When this option is specified, the carry bit propagates from left to right.**

```
Preg += Preg (BREV);
Ireg += Mreg (BREV);
```



  - With the Index Register version of this instruction, circular buffering is disabled to support operand addressing for FFT, DCT, and DFT algorithms.

    ```
    I0 += M0 (brev);
    ```

  - Pointer register example

    ```
    P3 += P0 (brev);
    ```

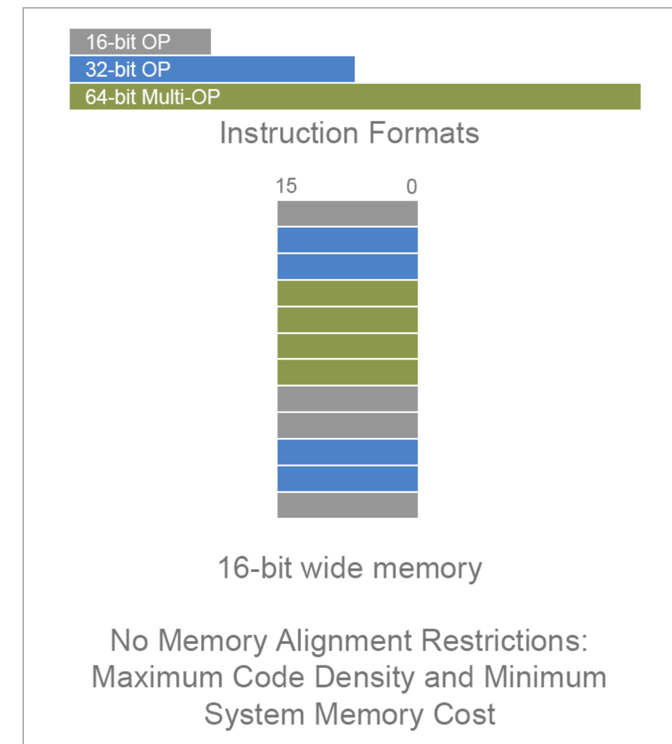
Instruction Set Encoding

Instruction Opcode Length

- The Blackfin architecture uses three instruction opcode lengths to obtain the best code density while maintaining high performance.
  - 16-bit instructions
    - Most control-type instructions and data fetches are 16-bits long to improve code density.
  - 32-bit instructions
    - Most control-type instructions with an immediate value in the expression and most arithmetic instructions are 32-bits in length.
  - Multi-issue 64-bit instructions
    - Certain 32-bit instructions can be executed in parallel with a pair of specific 16-bit instructions and specified with one 64-bit instruction.
      - Typically a 32-bit ALU/MAC instruction and one or two data fetch instructions
  - The delimiter symbol to separate instructions in a multi-issue instruction is a double pipe character "||."

Instruction Opcode Packing

- **When code is compiled and linked, the instructions are packed into memory as densely as possible.**
  - **i.e., no wasted memory space**
- **No memory alignment restrictions for code:**
  - **Instructions can be placed anywhere in memory.**
  - **The sequencer fetches 64-bits of instruction at a time (from 8-byte boundaries) and performs an alignment operation to:**
    - **Isolate shorter opcodes**
    - **Realign larger opcodes that straddle 4-/8-byte boundaries**
  - **This realignment hardware is transparent to the user.**



Blackfin ISA Encoding

Instruction and Version	Opcode Range	Bin															
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Instruction Name	Single Hex Value	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit
Syntax without variable arguments (16-bit Instruction)																	
Instruction Name	Min. Value—Max. Value	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit
Syntax with variable arguments (16-bit Instruction)																	
Instruction Name	Single Hex Value	bit	bit	bit	bit	bit	Most significant bits						bit	bit	bit	bit	bit
		bit	bit	bit	bit	bit	Least significant bits						bit	bit	bit	bit	bit
Syntax without variable arguments (32-bit Instruction)																	
Instruction Name	Min. Value—Max. Value	bit	bit	bit	bit	bit	Most significant bits						bit	bit	bit	bit	bit
		bit	bit	bit	bit	bit	Least significant bits						bit	bit	bit	bit	bit
Syntax with variable arguments (32-bit Instruction)																	

Blackfin ISA Encoding (cont)

Instruction and Version	Opcode Range	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Jump	0x0050—0x0057	0	0	0	0	0	0	0	0	0	1	0	1	Preg #			
JUMP (Preg)																	

Instruction and Version	Opcode Range	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Move Register	0xC00B 3800—0xC00B 39C0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	1	1
		0	0	1	1	1	0	0	Dreg_even #		0	0	0	0	0	0
Dreg_even = A0																	

Instruction and Version	Opcode Range	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Arithmetic Shift	0xC680 2180—0xC680 2FFF	1	1	0	0	0	1	1	0	1	x	x	0	0	0	0	0
		0	0	1	0	Dest. Dreg #			2's comp. of uimm4						Source Dreg #		
Dreg_hi = Dreg_lo >>> uimm4																	

Issuing Parallel Instructions

DSP64:
2 computes
2 load/stores

32-bit ALU/MAC instruction	16-bit instruction	16-bit instruction

- **64-bit instruction with 3 slots**
  - Slot 1 : One 32-bit DSP instruction or MNOP
  - Slot 2 : One 16-bit load or store instruction
    - Can be any of DSP Load/Store, Preg load/store, NOP
  - Slot 3 : One 16-bit load or store instruction
    - Can be a DSP Load
    - Can be a DSP Store if Slot 2 is NOT a DSP store
    - Can be a NOP
- **The Blackfin DSP Instruction Set Reference contains a table that lists which instructions can be placed in a particular multi-issue slot**

Parallel Issue Examples

- Two parallel memory access instructions

```
R3.H=(A1+=R0.L*R1.H), R3.L=(A0+=R0.L*R1.L) || r0 = [i0++] || r1 = [i1++];
mnop || r1 = [i0++] || r3 = [i1++];
```

- One Ireg and one memory access in parallel

```
R2=R2+|+R4, R4=R2-|-R4 (ASR) || I0+=M0 (BREV) || R1=[I0]
r7.h = r7.l=sign(r2.h)*r3.h + sign(r2.l)*r3.l || i1 += m3 || r0 = [i0];
```

- One Ireg Instruction in parallel

```
R6=(A0+=R3.H*R2.H) (FU) || I2-=M0;
```

Code Density Versus Speed

- There is more than one way to execute multiple instructions
- Execute as two consecutive instructions

```
r6=(a0+=r3.h*r2.h) (fu);      // 32-bit instruction
i2-=m0;                       // 16-bit instruction
```

  - These two instructions take two cycles to execute out of L1 memory
  - The total code memory required to store these two instructions is 6 bytes
- Execute in a multi-issue instruction

```
r6=(a0+=r3.h*r2.h) (fu) || i2-=m0;
```

  - Note that the assembler realizes a multi-issue instruction (based on the || syntax), and implicitly inserts a 16-bit NOP in the second multi-issue slot
  - A fully equivalent form of this multi-issue instruction is

```
r6=(a0+=r3.h*r2.h) (fu) || i2-=m0 || nop;
```

  - The multi-issue instruction takes one cycle to execute out of L1 memory
  - The total code memory required to store this multi-issue instruction is 8 bytes

Instruction Set Operations

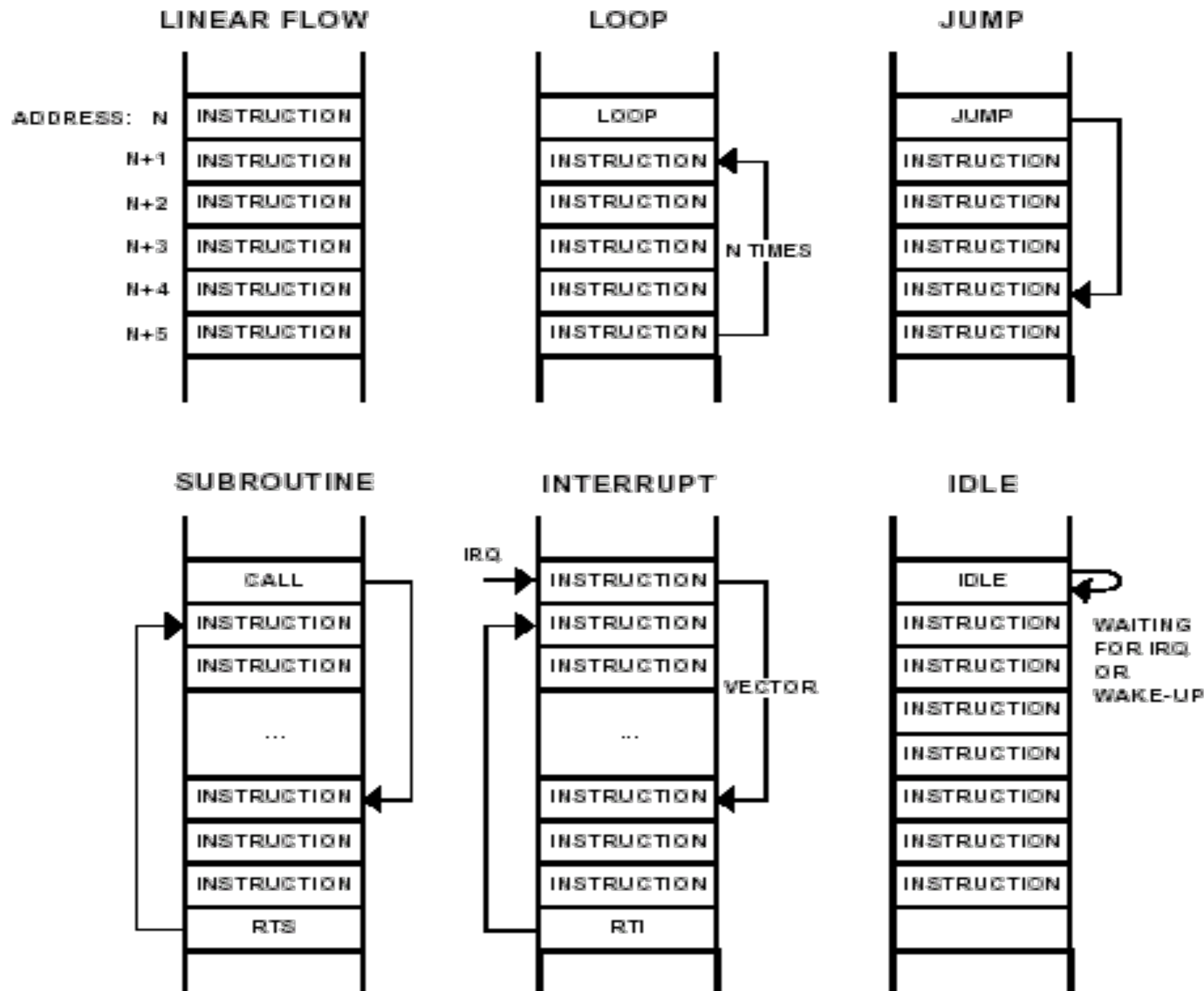
- Program Flow Operation
- Load-Store Operations
- Move Operations
- Stack Operation
- Control Code Operations
- Arithmetic Operations
- Logical Operations
- Bit, Shift and Rotate Operation
- External Event Management
- Video Pixel Operations
- Vector Operations

Program Flow Operations

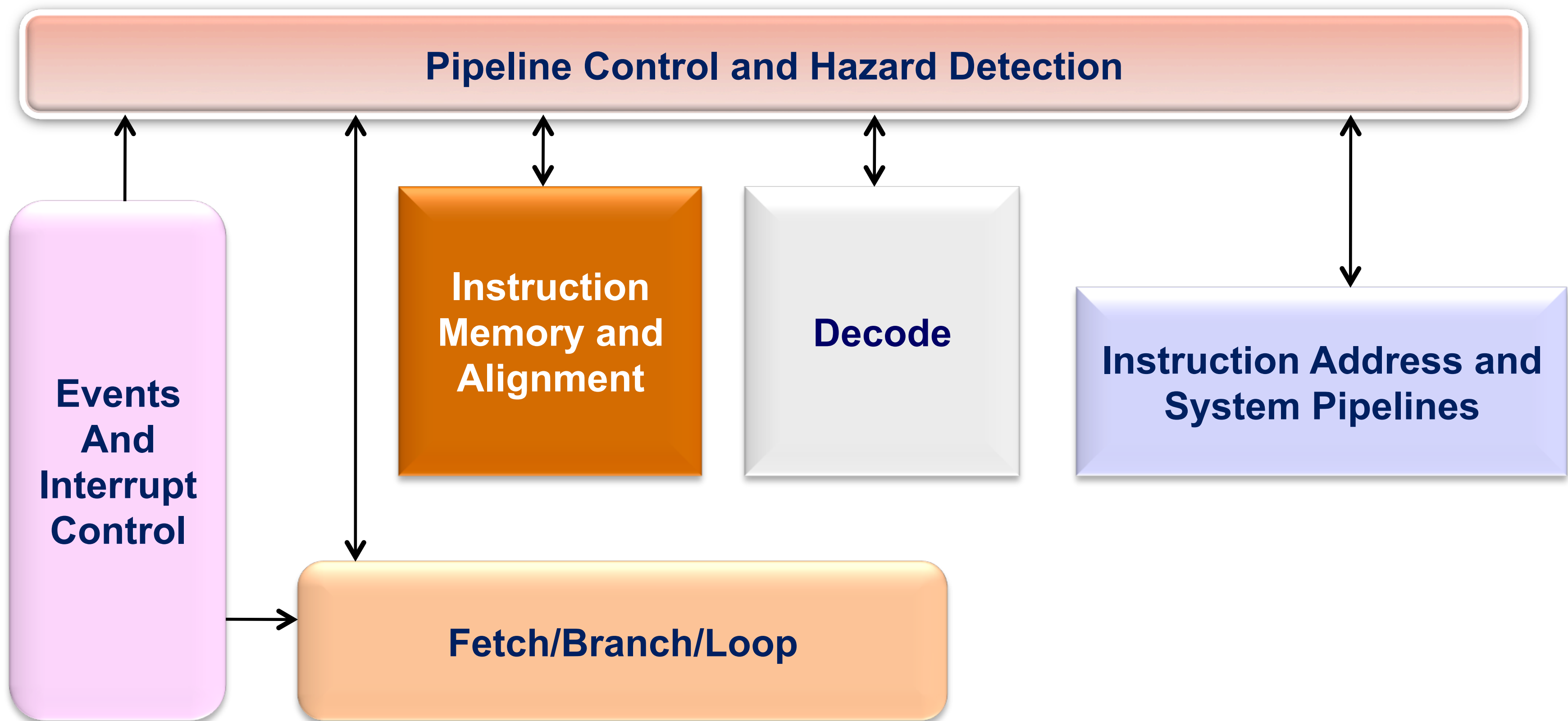
Program Sequencer Features

- **The Program Sequencer controls all program flow. It maintains Loops, Subroutines, Jumps, Idle, Interrupts and Exceptions**
  - The Jump instruction forces a new value into the Program Counter (PC) to change program flow. In the Indirect and Indexed versions of the instruction, the value in Preg must be an even number (bit0=0) to maintain 16-bit address alignment. Otherwise, an odd offset in Preg causes the processor to invoke an alignment exception.
  - The Conditional JUMP instruction forces a new value into the Program Counter (PC) to change the program flow, based on the value of the CC bit. The range of valid offset values is –1024 through 1022.
  - The CALL instruction calls a subroutine from an address that a P-register points to or by using a PC-relative offset. After the CALL instruction executes, the RETS register contains the address of the next instruction. The value in the Preg must be an even value to maintain 16-bit alignment.

Variations in Program Flow



Program Sequencer State Machine



Program Sequencer Related Registers

Register Name	Description
SEQSTAT	Sequencer Status register (HWERR/EXCAUSE)
 RETX RETN RETI RETE RETS	**Return Address registers:** Exception Return NMI Return Interrupt Return Emulation Return Subroutine Return
 LC0, LC1 LT0, LT1 LB0, LB1	**Zero-Overhead Loop registers:** Loop Counters Loop Tops Loop Bottoms
FP, SP	**Frame Pointer and Stack Pointer**
SYSCFG	System Configuration Register (CCEN, Sup. Single Step)
CYCLES, CYCLES2	Cycle Counters:
PC	Program Counter

Program Flow Control Instructions

- Jump
  - JUMP (destination_indirect)
  - JUMP (PC + offset)
  - JUMP offset
  - JUMP.S offset
  - JUMP.L offset
- IF CC JUMP
  - IF CC JUMP destination
  - IF !CC JUMP destination
- Call
  - CALL (destination_indirect)
  - CALL (PC + offset)
  - CALL offset
- LSETUP, LOOP
  - There are two forms of this instruction. The first is:
    - LOOP loop_name loop_counter
    - LOOP_BEGIN loop_name
    - LOOP_END loop_name
  - The second form is:
    - LSETUP (Begin_Loop, end_Loop)Loop_Counter
- RTS, RTI, RTX, RTN, RTE (Return)
  - RTS ; // Return from Subroutine (a)
  - RTI ; // Return from Interrupt (a)
  - RTX ; // Return from Exception (a)
  - RTN ; // Return from NMI (a)
  - RTE ; // Return from Emulation (a)

Load Store Operations

LOAD Operations

- The Load Immediate instruction loads immediate values, or explicit constants, into registers. The only values that can be immediately loaded into 40-bit Accumulator registers are zeros.
- The Load Pointer Register instruction loads a 32-bit P-register with a 32-bit word from an address specified by a P-register.
- The Load Data Register instruction loads a 32-bit word into a 32-bit D-register from a memory location. The Source Pointer register can be a P-register, I-register, or the Frame Pointer.
- The Load Half-Word – Sign-Extended instruction loads 16 bits sign-extended from a memory location into a 32-bit data register. The Pointer register is a P-register. The MSB of the number loaded is replicated in the whole upper-half word of the destination D-register.
- The Load Byte – Zero-Extended instruction loads an 8-bit byte, zero-extended to 32 bits indicated by an I-register or a P-register, from a memory location into a 32-bit data register. Fill the D-register bits 31–8 with zeros.

STORE Operations

- The Store Pointer Register instruction stores the contents of a 32-bit P-register to a 32-bit memory location. The Pointer register is a P-register.
- The Store Data Register instruction stores the contents of a 32-bit D-register to a 32-bit memory location. The destination Pointer register can be a P-register, I-register, or the Frame Pointer.
- The Store High Data Register Half instruction stores the most significant 16 bits of a 32-bit data register to a 16-bit memory location. The Pointer register is either an I-register or a P-register.
- The Store Low Data Register Half instruction stores the least significant 16 bits of a 32-bit data register to a 16-bit memory location. The Pointer register is either an I-register or a P-register.
- The Store Byte instruction stores the least significant 8-bit byte of a data register to an 8-bit memory location. The Pointer register is a P-register. The indirect address and offset have no restrictions for memory address alignment.

LOAD/STORE Instructions

- **Load Immediate**
  - register = constant
  - A1 = A0 = 0
- **Load Pointer Register**
  - P-register = [indirect_address]
- **Load Data Register**
  - D-register = [indirect_address]
- **Load Half-Word –Zero-Extended**
  - D-register = W [indirect_address] (Z)
- **Load Half-Word –Sign-Extended**
  - D-register = W [indirect_address] (X)
- **Load High Data Register Half**
  - Dreg_hi = W [indirect_address]
- **Load Low Data Register Half**
  - Dreg_lo = W [indirect_address]
- **Load Low Data Register Half**
  - Dreg_lo = W [indirect_address]
- **Load Byte –Sign-Extended**
  - D-register = B [indirect_address] (X)
- **Store Pointer Register**
  - [indirect_address] = P-register
- **Store Data Register**
  - [indirect_address] = D-register
- **Store High Data Register Half**
  - W [indirect_address] = Dreg_hi
- **Store Low Data Register Half**
  - W [indirect_address] = Dreg_lo
  - W [indirect_address] = D-register
- **Store Byte**
  - B [indirect_address] = D-register

Move Operations

Move Operations

- The Move Register instruction copies the contents of the source register into the destination register. The operation does not affect the source register contents.
  - All moves from smaller to larger registers are sign extended.
- The Move Conditional instruction moves source register contents into a destination register, depending on the value of CC.

  IF CC DPreg = DPreg, the move occurs only if CC = 1.

  IF ! CC DPreg = DPreg, the move occurs only if CC = 0.
- The source and destination registers are any D-register or P-register.

MOVE Instructions

Move Register

dest_reg = src_reg

Move Conditional

IF CC dest_reg = src_reg

IF ! CC dest_reg = src_reg

Move Half to Full Word –Zero-Extended

dest_reg = src_reg (Z)

Move Half to Full Word –Sign-Extended

dest_reg = src_reg (X)

Move Register Half

dest_reg_half = src_reg_half

dest_reg_half = accumulator (opt_mode)

Move Byte –Zero-Extended

dest_reg = src_reg_byte (Z)

Move Byte –Sign-Extended

dest_reg = src_reg_byte (X)

Stack Operations

Stack Operations

- The Push instruction stores the contents of a specified register in the stack. The instruction pre-decrements the Stack Pointer to the next available location in the stack first.
- Push and Push Multiple are the only instructions that perform pre-modify functions. The stack grows down from high memory to low memory. Consequently, the decrement operation is used for pushing, and the increment operation is used for popping values.
- The Stack Pointer always points to the last used location. Therefore, the effective address of the push is SP–4. The following illustration shows what the stack would look like when a series of pushes occur.
- The Stack Pointer must already be 32-bit aligned to use this instruction. If an unaligned memory access occurs, an exception is generated and the instruction aborts.

Stack Addressing 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 Addressing 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

General Form
dest_reg = [SP ++]

Syntax
***mostreg* = [SP ++] ; /* post-increment SP; does not apply to Data Registers and Pointer Registers (a) */**

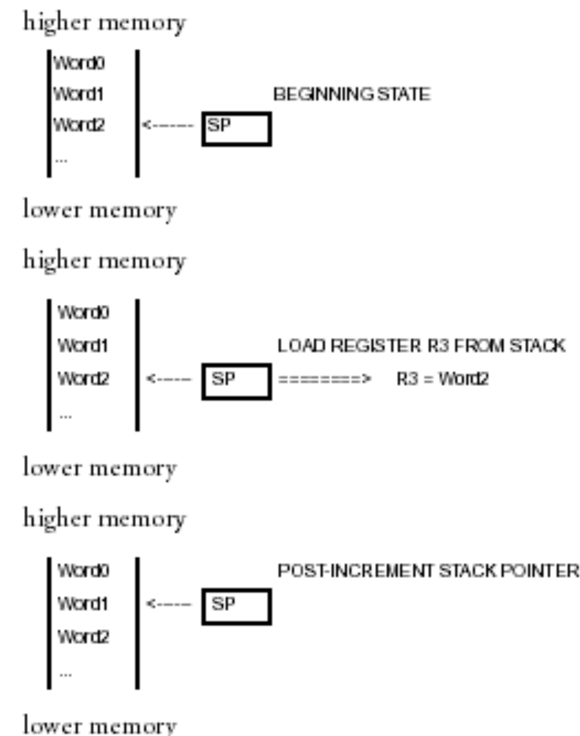
***Dreg* = [SP ++] ; /* Load Data Register instruction (repeated here for user convenience) (a) */**

***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 Addressing 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

Control Code Operations

CC Operations

- The Compare Data Register instruction sets the Control Code (CC) bit based on a comparison of two values. The input operands are D-registers. The compare operations are nondestructive on the input operands and affect only the CC bit and the flags. The value of the CC bit determines all subsequent conditional branching.
- The various forms of the Compare Data Register instruction perform 32-bit signed compare operations on the input operands or an unsigned compare operation, if the (IU) optional mode is appended. The compare operations perform a subtraction and discard the result of the subtraction without affecting user registers. The compare operation that you specify determines the value of the CC bit.
- Similar operations can be performed on the Pointer Registers.

Blackfin Instruction Set: Condition Code Bit Management

Compare Data Register

- CC = operand_1 == operand_2
- CC = operand_1 < operand_2
- CC = operand_1 <= operand_2
- CC = operand_1 < operand_2 (IU)
- CC = operand_1 <= operand_2 (IU)

Compare Pointer

- CC = operand_1 == operand_2
- CC = operand_1 < operand_2
- CC = operand_1 <= operand_2
- CC = operand_1 < operand_2 (IU)
- CC = operand_1 <= operand_2 (IU)

Move CC

- dest = CC
- dest |= CC
- dest &= CC
- dest ^= CC
- CC = source
- CC |= source
- CC &= source
- CC ^= source

Negate CC

- CC = ! CC

Compare Accumulator

- CC = A0 == A1
- CC = A0 < A1
- CC = A0 <= A1

CC Bit Instructions

Data Register-related instructions		
	CC = Dreg == Dreg ;	/* equal, register, signed (a) */
	CC = Dreg == imm3 ;	/* equal, immediate, signed (a) */
	CC = Dreg < Dreg ;	/* less than, register, signed (a) */
	CC = Dreg < imm3 ;	/* less than, immediate, signed (a) */
	CC = Dreg <= Dreg ;	/* less than or equal, register, signed (a) */
	CC = Dreg <= imm3 ;	/* less than or equal, immediate, signed (a) */
	CC = Dreg < Dreg (IU) ;	/* less than, register, unsigned (a) */
	CC = Dreg < uimm3 (IU) ;	/* less than, immediate, unsigned (a) */
	CC = Dreg <= Dreg (IU) ;	/* less than or equal, register, unsigned (a) */
	CC = Dreg <= uimm3 (IU) ;	/* less than or equal, immediate unsigned (a) */

Pointer Register-related instructions		
	CC = Preg == Preg ;	/* equal, register, signed (a) */
	CC = Preg == imm3 ;	/* equal, immediate, signed (a) */
	CC = Preg < Preg ;	/* less than, register, signed (a) */
	CC = Preg < imm3 ;	/* less than, immediate, signed (a) */
	CC = Preg <= Preg ;	/* less than or equal, register, signed (a) */
	CC = Preg <= imm3 ;	/* less than or equal, immediate, signed (a) */
	CC = Preg < Preg (IU) ;	/* less than, register, unsigned (a) */
	CC = Preg < uimm3 (IU) ;	/* less than, immediate, unsigned (a) */
	CC = Preg <= Preg (IU) ;	/* less than or equal, register, unsigned (a) */
	CC = Preg <= uimm3 (IU) ;	/* less than or equal, immediate unsigned (a) */

Accumulator-related instructions		
	CC = A0 == A1 ;	/* equal, signed (a) */
	CC = A0 < A1 ;	/* less than, Accumulator, signed (a) */
	CC = A0 <= A1 ;	/* less than or equal, Accumulator, signed (a) */

Conditional Code (CC) Bit in ASTAT

- **CC bit is used in several instructions**
  - Action taken in the instruction depends on the value of CC

```
If cc jump here;      //if cc = 1, jump to label "here"
If cc r3 = r0;        // perform move if cc=1
```

- **CC bit value is based on a comparison of two registers, pointers or accumulators**
- **CC bit can be moved to and from a data register or ASTAT bit**
- **CC bit can be negated:**

```
If !cc jump here;
```

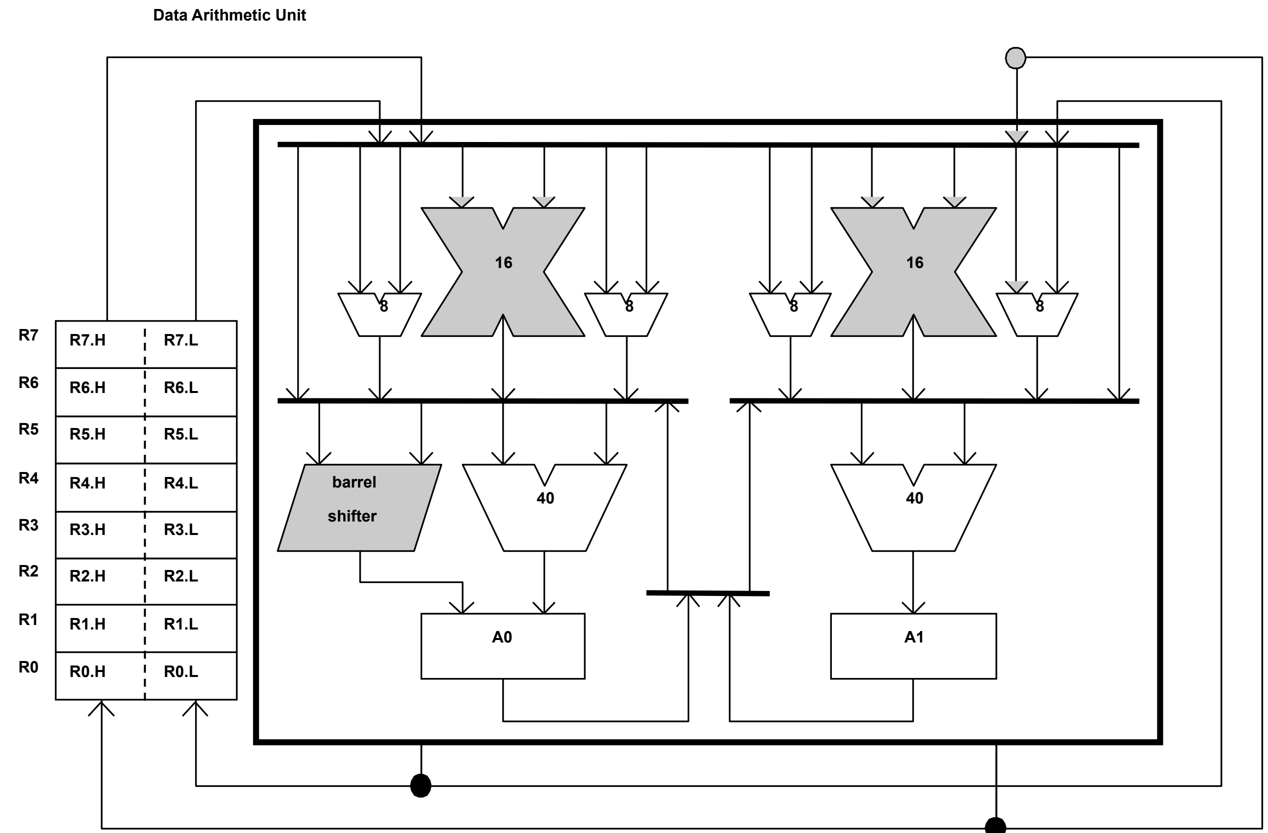
Arithmetic Operations

Arithmetic Logic Unit (ALU)

- Two ALUs operating on 16-bit, 32-bit, and 40-bit input operands and output 16-bit, 32-bit, and 40-bit results.
- Functions
  - Fixed-point addition and subtraction
  - Addition and subtraction of immediate values
  - Accumulator and subtraction of multiplier results
  - Logical AND, OR, NOT, XOR, bitwise XOR, Negate
  - Functions: ABS, MAX, MIN, Round, division primitives
- Features
  - Supports conditional instructions
  - 8-bit video ALU operations

ALU Operations

- Single 16-Bit Ops
- Dual 16-Bit Ops
- Quad 16-Bit Ops
- Single 32-Bit Ops
- Dual 32-Bit Ops



ALU Operations: Single 16-Bit Operations

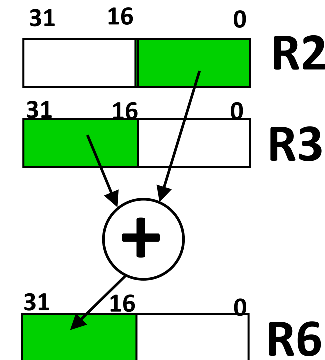
- **Single 16-bit Addition, Subtraction Operations**
  - Any two 16-bit register halves may be used as inputs.
  - One 16-bit result is deposited in designated 16-bit register half.

- **General Form:**

    Dreg_lo_hi = Dreg_lo_hi + Dreg_lo_hi;

- **Example:**

    R6.H = R2.L + R3.H; //no saturation



Single
16-bit addition

ALU Operations: Dual 16-Bit Operations

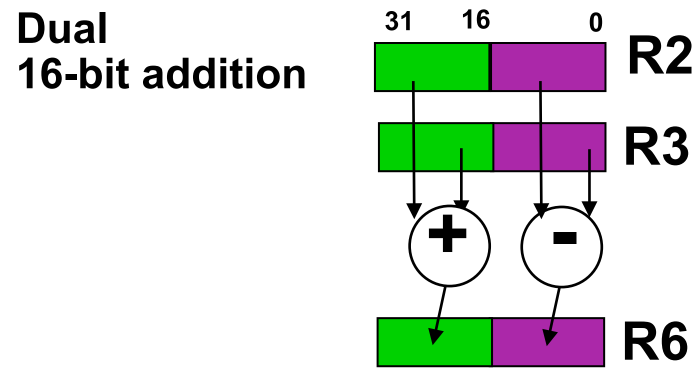
- **Dual 16-bit Addition, Subtraction Operations**
  - Any two 32-bit registers may be used as inputs.
  - Two 16-bit results are deposited in designated 32-bit register.
- **General Form:**

  Dreg = Dreg +|+ Dreg; Dreg = Dreg -|- Dreg;

  Dreg = Dreg +|- Dreg; Dreg = Dreg -|+ Dreg;

- **Example:**

  R6 = R2 + | - R3; // R6.H=R2.H + R3.H, R6.L=R2.L - R3.L



ALU Operations: Quad 16-Bit Operations

- **Quad 16-bit Addition, Subtraction Operations**
  - Any two 32-bit registers may be used as inputs.
  - Four 16-bit results are deposited in two designated 32-bit registers.
- **General Form:**

  Dreg = Dreg +|+ Dreg, Dreg = Dreg -|- Dreg;

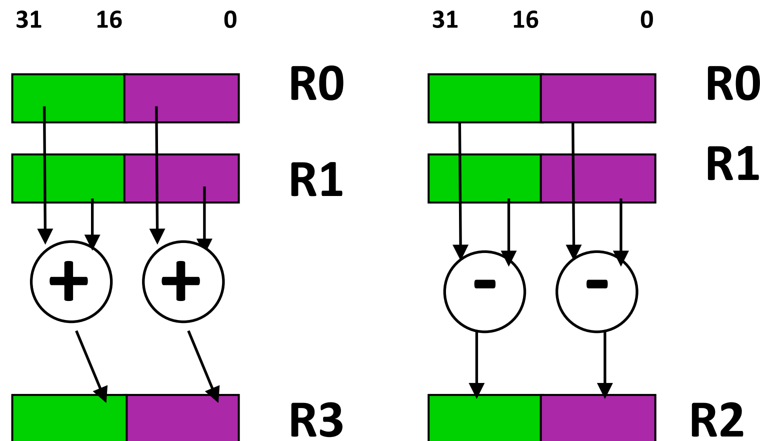
  Dreg = Dreg +|- Dreg, Dreg = Dreg -|+ Dreg;
- **Example:**

  R3 = R0 + | + R1, R2 = R0 - | - R1; // R3.H=R0.H + R1.H, R3.L=R0.L + R1.L

  // R2.H=R0.H - R1.H, R2.L=R0.L – R1.L

Quad 16-bit addition

Note:
1. BF535 saturates and then scales the result.
2. BF533/2/1 scales and then saturates the result.



ALU Operations: Single 32-Bit Operations

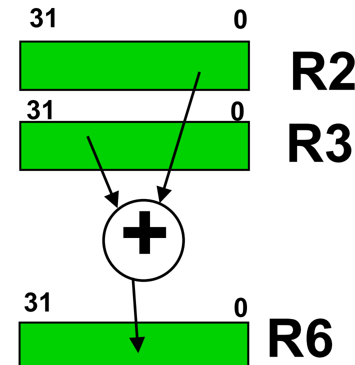
- **Single 32-bit Addition, Subtraction Operations**
  - Any two 32-bit registers may be used as inputs.
  - One 32-bit result is deposited in designated 32-bit register.
- **General Form:**

  Dreg = Dreg + Dreg;

  Dreg = Dreg – Dreg;
- **Example:**

  R6 = R2 + R3;

32-bit addition

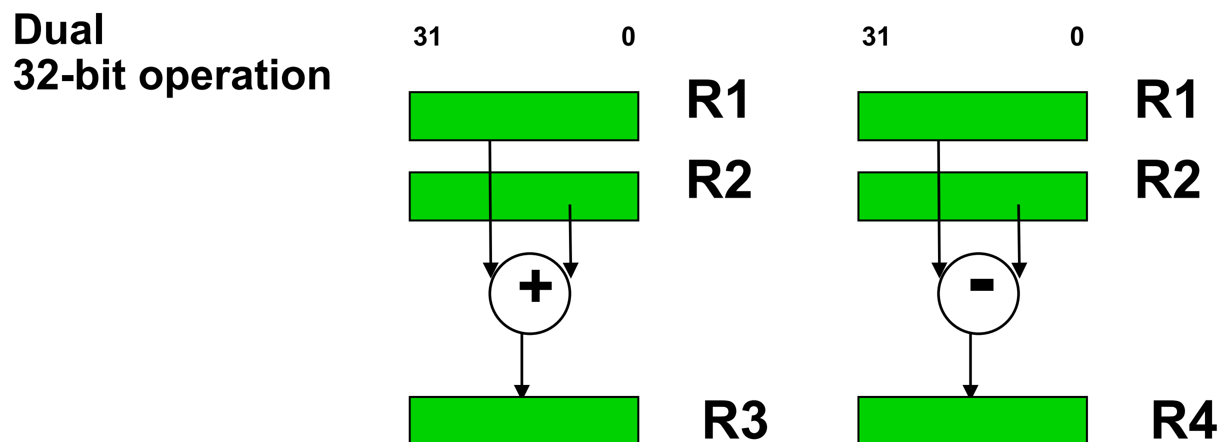


ALU Operations: Dual 32-Bit Operations

- **Dual 32-bit Addition, Subtraction Operations**
  - Any two 32-bit registers may be used as inputs.
  - Two 32-bit result is deposited in designated 32-bit register.
- **General Form:**

  Dreg = Dreg + Dreg, Dreg = Dreg – Dreg;
- **Example:**

  R3 = R1 + R2, R4 = R1 - R2;



ALU Operation: Options and Examples

- **opt_mode_0 supports the Dual and Quad 16-Bit Operations versions of this instruction.**
- **opt_mode_1 supports the Dual 32-bit and 40-bit operations.**
- **opt_mode_2 supports the Quad 16-Bit Operations versions of this instruction.**

Examples:

R6 = R0 -|+ R1 (s);

R7 = R3 -|- R6 (SCO);

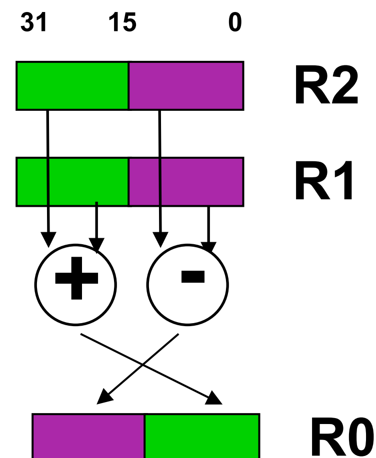
Mode	Option	Description
opt_mode_0	S	Saturate the results at 16 bits.
	CO	Cross option. Swap the order of the results in the destination register.
	SCO	Saturate and cross option. Combination of (S) and (CO) options.
opt_mode_1	S	Saturate the results at 16 or 32 bits, depending on the operand size.
opt_mode_2	ASR	Arithmetic shift right. Halve the result (divide by 2) before storing in the destination register. If specified with the S (saturation) flag in Quad 16-Bit Operand versions of this instruction, the scaling is performed before saturation.
	ASL	Arithmetic shift left. Double the result (multiply by 2, truncated) before storing in the destination register. If specified with the S (saturation) flag in Quad 16-Bit Operand versions of this instruction, the scaling is performed before saturation.

ALU Operations: Dual 16-Bit Cross Options

- High result is placed in the low half of designated result register.
- Low result is placed in the high half of designated result register.

- Example:

R0 = R2 +|- R1 (CO); // R0.L=R2.H+R1.H, R0.H=R2.L-R1.L



Rounding Instructions

- The Round to Half-Word instruction rounds a 32-bit, normalized-fraction number into a 16-bit, normalized-fraction number by extracting and saturating bits 31–16, then discarding bits 15–0. The instruction supports only biased rounding, which adds a half LSB (in this case, bit 15) before truncating bits 15–0. The ALU performs the rounding.
- The RND_MOD bit in the ASTAT register has no bearing on the rounding behavior of this instruction.

```
Dreg_lo_hi =Dreg (RND) ; /* round and saturate the source to 16 bits. */

/* If r6 = 0xFFFC FFFF, then rounding to 16-bits with . . . */
r1.l = r6 (rnd) ;       // . . . produces r1.l = 0xFFFD
// If r7 = 0x0001 8000, then rounding . . .
r1.h = r7 (rnd) ;       // . . . produces r1.h = 0x0002
```

Rounding Instructions

- The Add/Subtract – Prescale Up instruction combines two 32-bit values to produce a 16-bit result as follows:
  - Prescale up both input operand values by shifting them four places to the left
  - Add or subtract the operands, depending on the instruction version used
  - Round and saturate the upper 16 bits of the result
  - Extract the upper 16 bits to the dest_reg
  - Dreg_lo_hi = Dreg + Dreg (RND12) ;
  - Dreg_lo_hi = Dreg - Dreg (RND12) ;
- The Add/Subtract -- Prescale Down instruction combines two 32-bit values to produce a 16-bit result as follows:
  - Prescale down both input operand values by arithmetically shifting them four places to the right
  - Add or subtract the operands, depending on the instruction version used
  - Round the upper 16 bits of the result
  - Extract the upper 16 bits to the dest_reg
  - Dreg_lo_hi = Dreg + Dreg (RND20) ;
  - Dreg_lo_hi = Dreg - Dreg (RND20) ;

Logical Operations

32-bit ALU Logical Operations

AND

General Form:

```
Dreg = Dreg & Dreg;
```

Example:

```
R4 = R4 & R3;
```

OR

General Form:

```
Dreg = Dreg | Dreg;
```

Example:

```
R4 = R4 | R3;
```

NOT

General Form:

```
Dreg = ~Dreg;
```

Example:

```
R3 = ~ R4;
```

XOR

General Form:

```
Dreg = Dreg ^ Dreg;
```

Example:

```
R4 = R4 ^ R3;
```

Bit, Shift and Rotate Operations

BIT Operations

- The Bit Set instruction sets the bit designated by bit_position in the specified D-register. It does not affect other bits in the D-register. The bit_position range of values is 0 through 31, where 0 indicates the LSB, and 31 indicates the MSB of the 32-bit D-register.
- The Bit Toggle instruction inverts the bit designated by bit_position in the specified D-register. The instruction does not affect other bits in the D-register.
- The Bit Field Deposit instruction merges the background bit field in backgnd_reg with the foreground bit field in the upper half of foregnd_reg and saves the result into dest_reg. The user determines the length of the foreground bit field and its position in the background field.

Blackfin Instruction Set: BIT Operations

BIT Operations

BITCLR

BITCLR (register, bit_position)

BITSET

BITSET (register, bit_position)

BITTGL

BITTGL (register, bit_position)

BITTST

CC = BITTST (register, bit_position)
CC = ! BITTST (register, bit_position)

Blackfin Instruction Set: BIT Operations

BIT Operations

DEPOSIT

dest_reg = DEPOSIT (backgnd_reg, foregnd_reg)
dest_reg = DEPOSIT (backgnd_reg, foregnd_reg) (X)

EXTRACT

dest_reg = EXTRACT (scene_reg, pattern_reg) (Z)
dest_reg = EXTRACT (scene_reg, pattern_reg) (X)

BITMUX

BITMUX (source_1, source_0, A0) (ASR)
BITMUX (source_1, source_0, A0) (ASL)

ONES (One's Population Count)

dest_reg = ONES src_reg

SHIFT/ROTATE Operations

- The Add with Shift instruction combines an addition operation with a one- or two-place logical shift left. Of course, a left shift accomplishes a x2 multiplication on sign-extended numbers. Saturation is not supported. The Add with Shift instruction does not intrinsically modify values that are strictly input. However, dest_reg serves as an input as well as the result, so dest_reg is intrinsically modified.

- The Arithmetic Shift instruction shifts a registered number a specified distance and direction while preserving the sign of the original number. The sign bit value back-fills the left-most bit positions vacated by the arithmetic right shift.

- The Logical Shift instruction logically shifts a register by a specified distance and direction. Logical shifts discard any bits shifted out of the register and backfill vacated bits with zeros.

Blackfin Instruction Set: SHIFT/ROTATE Operations

SHIFT/ROTATE Operations

Add with Shift

dest_pntr = (dest_pntr + src_reg) << 1
dest_pntr = (dest_pntr + src_reg) << 2
dest_reg = (dest_reg + src_reg) << 1
dest_reg = (dest_reg + src_reg) << 2

Shift with Add

dest_pntr = adder_pntr + (src_pntr << 1)
dest_pntr = adder_pntr + (src_pntr << 2)

Arithmetic Shift

dest_reg >>>= shift_magnitude
dest_reg = src_reg >>> shift_magnitude (opt_sat)
dest_reg = src_reg << shift_magnitude (S)
accumulator = accumulator >>> shift_magnitude
dest_reg = ASHIFT src_reg BY shift_magnitude (opt_sat)
accumulator = ASHIFT accumulator BY shift_magnitude

Blackfin Instruction Set: SHIFT/ROTATE Operations

SHIFT/ROTATE Operations

Logical Shift

dest_pntr = src_pntr >> 1
dest_pntr = src_pntr >> 2dest_pntr = src_pntr << 1
dest_pntr = src_pntr << 2dest_reg >>= shift_magnitude
dest_reg <<= shift_magnitude
dest_reg = src_reg >> shift_magnitude
dest_reg = src_reg << shift_magnitude
dest_reg = LSHIFT src_reg BY shift_magnitude

ROT (Rotate)

dest_reg = ROT src_reg BY rotate_magnitude
accumulator_new = ROT accumulator_old BY rotate_magnitude

Event Controller

Events (Interrupts / Exceptions)

- **The Event Controller manages 5 types of Events:**
  - Emulation (via SW or external pin)
  - Reset (via SW or external pin)
  - Non-Maskable Interrupt (NMI) - for events that require immediate processor attention (via SW or external pin)
  - Exception
  - Interrupts
    - Global Interrupt Enable
    - Hardware Error
    - Core Timer
    - 9 General-Purpose Interrupts for servicing peripherals

Interrupts vs. Exceptions

INTERRUPTS

- **Hardware-generated**
  - Asynchronous to program flow
  - Requested by a peripheral
- **Software-generated**
  - Synchronous to program flow
  - Generated by RAISE instruction
- **All instructions preceding the interrupt in the pipeline are killed**

The ADSP-BF53x is always in Supervisor Mode while executing Event Handler software and can be in User Mode only while executing application tasks.

EXCEPTIONS

- **Service Exception**
  - Return address is the address following the excepting instruction
  - Never re-executed
  - EXCPT instruction is in this category
- **Error Condition Exception**
  - Return address is the address of the excepting instruction
  - Excepting instruction will be re-executed
- **Exception Handler must insure the appropriate return address in RETX based on exception type**

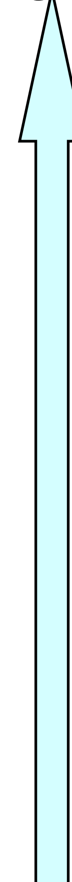
Exception Causes

Priority	Exception	EXCAUSE
1	Unrecoverable Event	0x25
2	I-Fetch Multiple CPLB Hits	0x2D
3	I-Fetch Misaligned Access	0x2A
4	I-Fetch Protection Violation	0x2B
5	I-Fetch CPLB Miss	0x2C
6	I-Fetch Access Exception	0x29
7	Watchpoint Match	0x28
8	Undefined Instruction	0x21
9	Illegal Combination	0x22
10	Illegal use protected resource	0x2E
11	DAG0 Multiple CPLB Hits	0x27
12	DAG0 Misaligned Access	0x24
13	DAG0 Protection Violation	0x23
14	DAG0 CPLB Miss	0x26
15	DAG1 Multiple CPLB Hits	0x27
16	DAG1 Misaligned Access	0x24
17	DAG1 Protection Violation	0x23
18	DAG1 CPLB Miss	0x26
19	EXCPT instruction	m- field
20	Single Step	0x10
21	Trace Buffer	0x11

Event Priorities

Event Number	Event Class	Name	MMR Location
EVT0	Emulation	EMU	0xFFE0 2000
EVT1	Reset	RST	0xFFE0 2004
EVT2	NMI	NMI	0xFFE0 2008
EVT3	Exception	EVX	0xFFE0 200C
EVT4	Reserved	Reserved	0xFFE0 2010
EVT5	Hardware Error	IVHW	0xFFE0 2014
EVT6	Core Timer	IVTMR	0xFFE0 2018
EVT7	Interrupt 7	IVG7	0xFFE0 201C
EVT8	Interrupt 8	IVG8	0xFFE0 2020
EVT9	Interrupt 9	IVG9	0xFFE0 2024
EVT10	Interrupt 10	IVG10	0xFFE0 2028
EVT11	Interrupt 11	IVG11	0xFFE0 202C
EVT12	Interrupt 12	IVG12	0xFFE0 2030
EVT13	Interrupt 13	IVG13	0xFFE0 2034
EVT14	Interrupt 14	IVG14	0xFFE0 2038
EVT15	Interrupt 15	IVG15	0xFFE0 203C

Highest



Lowest

Event Management Instructions

- Disable Interrupts
  - CLI
- Enable Interrupts
  - STI
- RAISE (Force Interrupt / Reset)
  - RAISE
- EXCPT (Force Exception)
  - EXCPT
- Idle
  - IDLE
- EMUEXCPT (Force Emulation)
  - EMUEXCPT

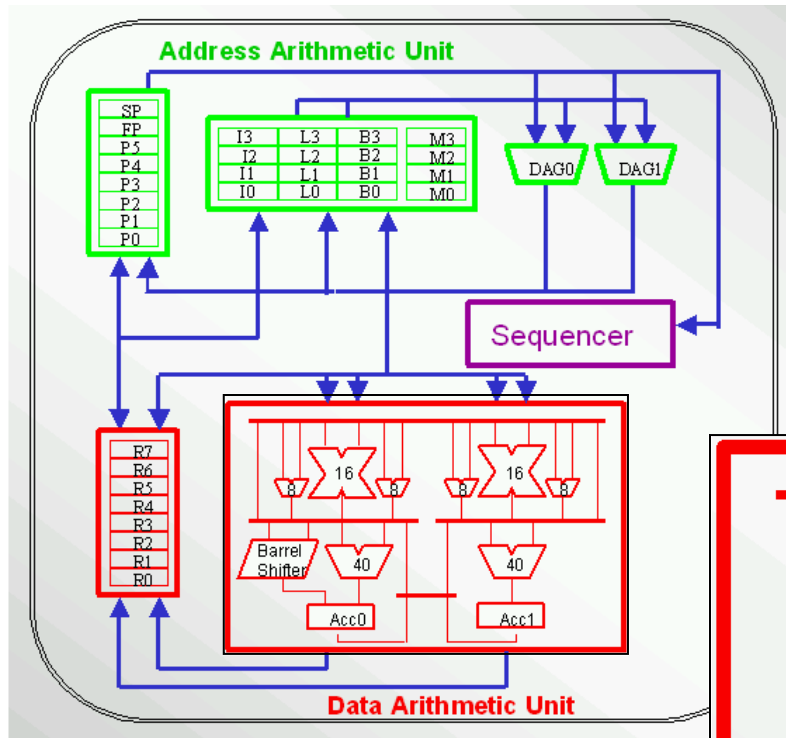
SSYNC and CSYNC instructions

- **SSYNC instruction synchronizes "the System", executing everything in the processor pipeline, and completing all pending reads and writes from peripherals.**
  - Until SSYNC completes, no further instructions can enter the pipeline.

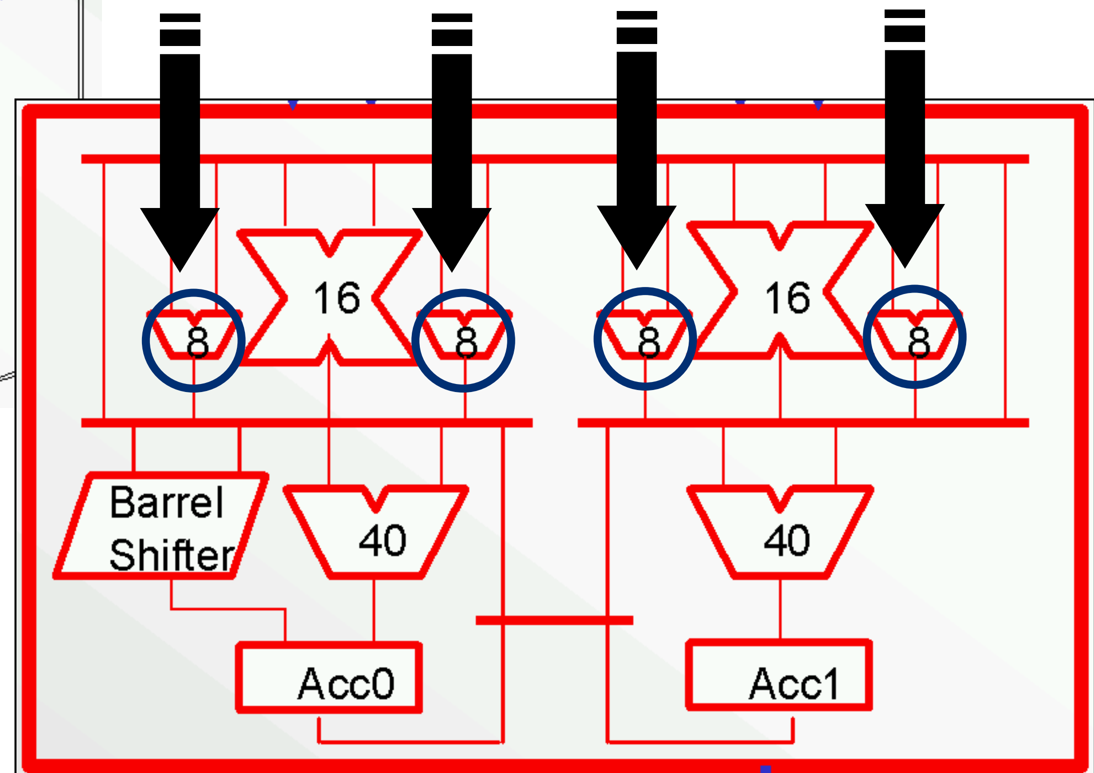
- **CSYNC instruction synchronizes "the Core", executing everything in the processor pipeline**
  - CSYNC is typically used after Core MMR writes to prevent imprecise behavior.

Video Pixel Operations

8-Bit Video ALUs

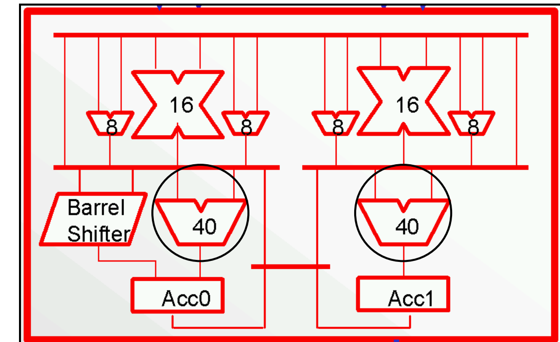


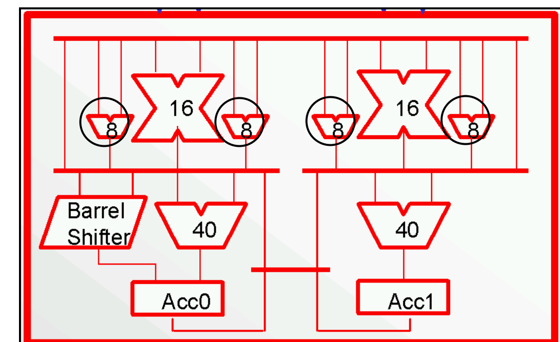
Four Video ALUs



40-Bit vs 8-Bit ALUs

- 8-Bit ALUs cannot be used in parallel with the regular 40-Bit ALUs
- R1 = R2 +|+ R3 uses the 40-Bit ALUs to perform 2 add operations
- (r1, r2) = BYTEOP16P(r3:2, r1:0) uses the 8-Bit ALUs to perform 4 add operations





Source Registers Contain	31.....24	23.....16	15.....8	7.....0
aligned_src_reg_0	y3	y2	y1	y0
aligned_src_reg_1	z3	z2	z1	z0

Destination Registers Receive	31.....24	23.....16	15.....8	7.....0
aligned_src_reg_0	y1 + z1		y0 + z0	
aligned_src_reg_1	y3 + z3		y2 + z2	

Video Pixel Instructions

ALIGN8, ALIGN16, ALIGN24

dest_reg = ALIGN8 (src_reg_1, src_reg_0)

dest_reg = ALIGN16 (src_reg_1, src_reg_0)

dest_reg = ALIGN24 (src_reg_1, src_reg_0)

DISALGNEXCPT

DISALGNEXCPT

BYTEOP3P (Dual 16-Bit Add / Clip)

dest_reg = BYTEOP3P (src_reg_0, src_reg_1) (LO)

dest_reg = BYTEOP3P (src_reg_0, src_reg_1) (HI)

BYTEOP3P (Dual 16-Bit Add / Clip)

dest_reg = BYTEOP3P (src_reg_0, src_reg_1) (LO, R)

dest_reg = BYTEOP3P (src_reg_0, src_reg_1) (HI, R)

Dual 16-Bit Accumulator Extraction with Addition

dest_reg_1 = A1.L + A1.H, dest_reg_0 = A0.L + A0.H

BYTEOP16P (Quad 8-Bit Add)

(dest_reg_1, dest_reg_0) = BYTEOP16P (src_reg_0, src_reg_1)

(dest_reg_1, dest_reg_0) = BYTEOP16P (src_reg_0, src_reg_1) (R)

Video Pixel Instructions

BYTEOP1P (Quad 8-Bit Average –Byte)

dest_reg = BYTEOP1P (src_reg_0, src_reg_1)

dest_reg = BYTEOP1P (src_reg_0, src_reg_1) (T)

dest_reg = BYTEOP1P (src_reg_0, src_reg_1) (R)

dest_reg = BYTEOP1P (src_reg_0, src_reg_1) (T, R)

BYTEOP2P (Quad 8-Bit Average –Half-Word)

dest_reg = BYTEOP2P (src_reg_0, src_reg_1) (RNDL)

dest_reg = BYTEOP2P (src_reg_0, src_reg_1) (RNDH)

dest_reg = BYTEOP2P (src_reg_0, src_reg_1) (TL)

dest_reg = BYTEOP2P (src_reg_0, src_reg_1) (TH)

BYTEOP2P (Quad 8-Bit Average –Half-Word)

dest_reg = BYTEOP2P (src_reg_0, src_reg_1) (RNDL, R)

dest_reg = BYTEOP2P (src_reg_0, src_reg_1) (RNDH, R)

dest_reg = BYTEOP2P (src_reg_0, src_reg_1) (TL, R)

dest_reg = BYTEOP2P (src_reg_0, src_reg_1) (TH, R)

BYTEPACK (Quad 8-Bit Pack)

dest_reg = BYTEPACK (src_reg_0, src_reg_1)

BYTEOP16M (Quad 8-Bit Subtract)

(dest_reg_1, dest_reg_0) = BYTEOP16M (src_reg_0, src_reg_1)

(dest_reg_1, dest_reg_0) = BYTEOP16M (src_reg_0, src_reg_1) (R)

BLACKfin

Vector Operations

Vector Operations

- Users can take advantage of vector instructions to perform simultaneous operations on multiple 16-bit values, including add, subtract, multiply, shift, negate, pack, and search.

Vector Instructions

- **Add on Sign**
  - dest_hi = dest_lo = SIGN (src0_hi) * src1_hi + SIGN (src0_lo) * src1_lo
- **Vector Add / Subtract**
  - dest = src_reg_0 –|+ src_reg_1
  - dest = src_reg_0 +|–src_reg_1
  - dest = src_reg_0 –|–src_reg_1
  - dest_0 = src_reg_0 +|+ src_reg_1,
  - dest_1 = src_reg_0 –|–src_reg_1
  - dest_0 = src_reg_0 +|–src_reg_1,
  - dest_1 = src_reg_0 –|+ src_reg_1
  - dest_0 = src_reg_0 + src_reg_1,
  - dest_1 = src_reg_0 –src_reg_1
  - dest_0 = A1 + A0, dest_1 = A1 –A0
  - dest_0 = A0 + A1, dest_1 = A0 –A1

Vector Instructions

- Vector Arithmetic Shift
  - dest_reg = src_reg >>> shift_magnitude (V)
  - dest_reg = ASHIFT src_reg BY shift_magnitude (V)
- Vector Logical Shift
  - dest_reg = src_reg >> shift_magnitude (V)
  - dest_reg = src_reg << shift_magnitude (V)
  - dest_reg = LSHIFT src_reg BY shift_magnitude (V)
- Vector MAX
  - dest_reg = MAX (src_reg_0, src_reg_1) (V)

Vector Instructions

- VIT_MAX (Compare-Select)
  - dest_reg = VIT_MAX (src_reg_0, src_reg_1) (ASL)
  - dest_reg = VIT_MAX (src_reg_0, src_reg_1) (ASR)
  - dest_reg_lo = VIT_MAX (src_reg) (ASL)
  - dest_reg_lo = VIT_MAX (src_reg) (ASR)
- Vector MIN
  - dest_reg = MIN (src_reg_0, src_reg_1) (V)
- Vector Negate (Two's Complement)
  - dest_reg = –source_reg (V)
- Vector PACK
  - Dest_reg = PACK (src_half_0, src_half_1)
- Vector SEARCH
  - (dest_pointer_hi, dest_pointer_lo) = SEARCH src_reg (searchmode)
- Vector ABS
  - dest_reg = ABS source_reg (V)
- Vector Add / Subtract
  - dest = src_reg_0 +|+ src_reg_1

(v) syntax

- The (v) syntax is used to distinguish between 32-bit operations and vector 16-bit operations

- Given these initial conditions:

```
/* r1 = 0xFFF7 7FFF, r0 = 0x000A 8000 */
```

  – 32-bit MIN instruction looks like this:

```
r7 = MIN(r1, r0);
/* r7 = 0xFFF7 7FFF */
```

- To make a Vector MIN instruction, append a (v) to the end

```
r7 = MIN(r1, r0) (v);
/* r7 = 0xFFF7 8000 */
```

- Also in this class of instructions: MAX, ABS, ASHIFT, LSHIFT, Negate