Section 9

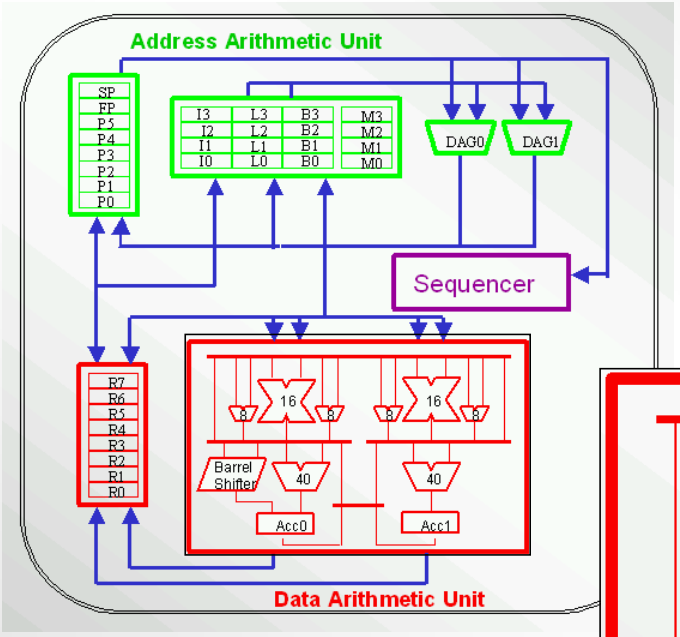
Advanced Instructions

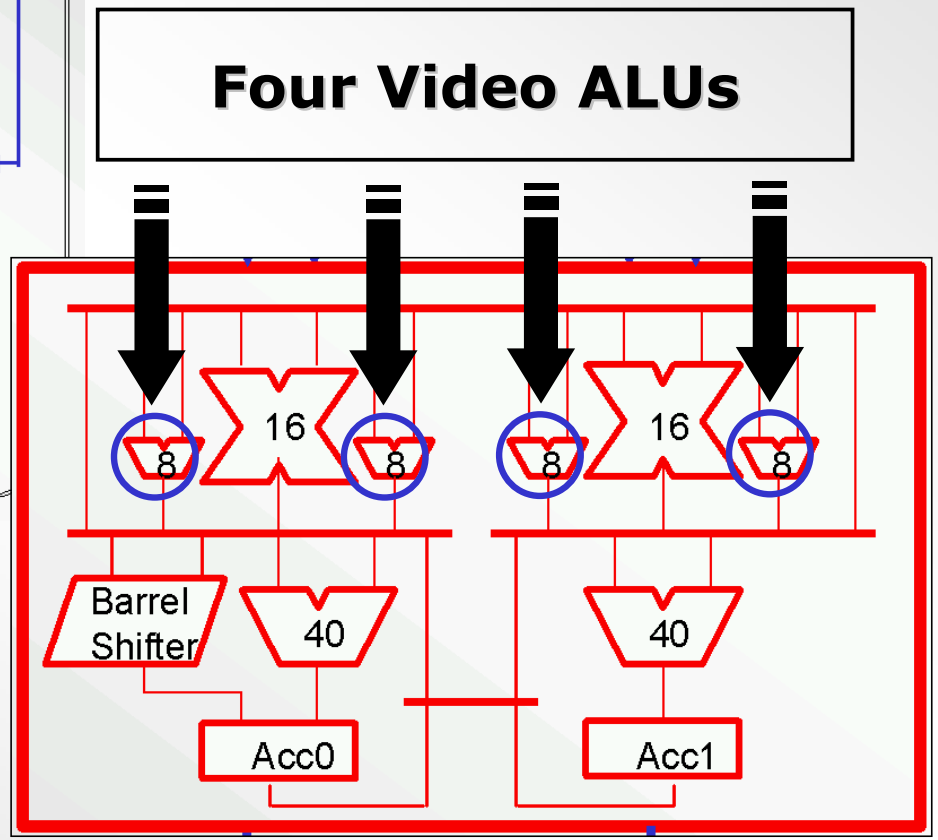
Instruction Set Overview

Program Flow Control	Load/Store
Move	Stack Control
Control Code Bit Management	Logical Operations
Bit Operations	Shift/Rotate Operations
Arithmetic Operations (Miscellaneous)	External Event Management
Cache Control	**8-Bit ALU Video Pixel Operations (Video Pixel Operations)**
Vector Operations	**Issuing Parallel Instructions**

8-Bit ALU Instructions
(Video Pixel Operations)

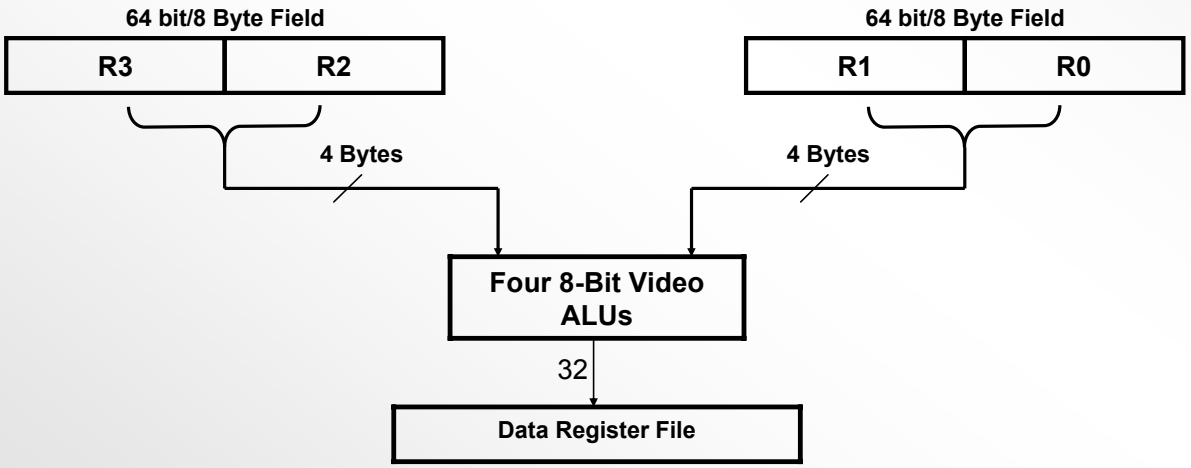
8-Bit Video ALUs





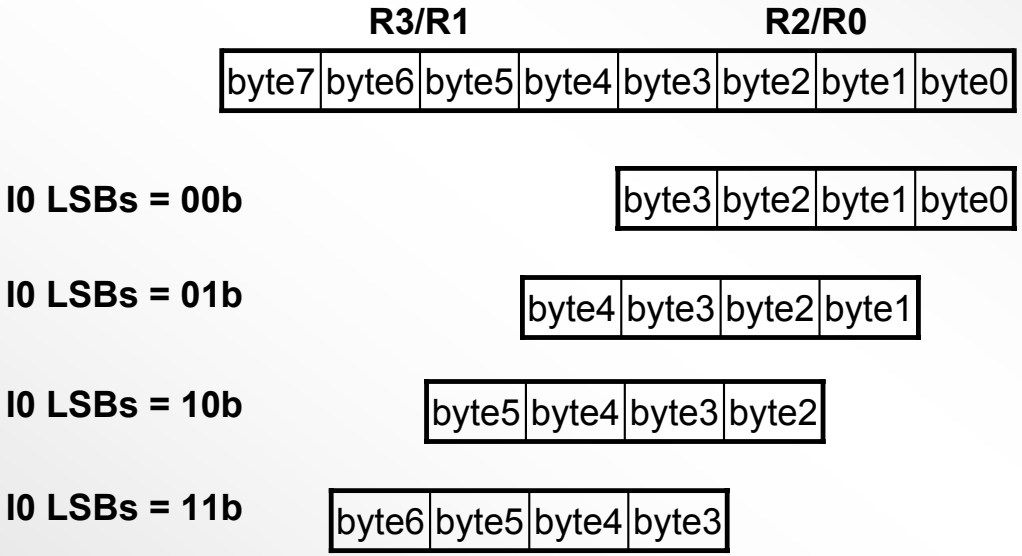
8-Bit ALU Operations

- Four 8-bit ALUs provide parallel computational power targeted mainly for video operations
- Each 8-Bit ALU instruction takes one cycle to complete
- These instructions may operate on one, two, three, or four 8-bit input pairs
- For the computational instructions, inputs from the data register file are structured in two 32-bit words, formed from two 64-bit fields in the register pairs R3:2 and R1:0



I0 and I1 for Byte Alignment

- In instructions that use a register pair for input, we must choose a 4-byte field from an 8-byte meta-register (R3:2 or R1:0)
- The least significant bits DAG register I0 (for src_reg_0, the first pair in the syntax) or I1 (for src_reg_1, the second pair in the syntax) is used for choosing the 4-byte field



- In some instructions, the (r) option allows the order of the registers in each pair to be reversed, resulting in the register pairs (R2:3 or R0:1)

Byte Alignment Exception Disable

- **DISALGNEXCPT**
  - **Disable alignment exception on parallel load/store instructions**
  - **Affects only misaligned 32-bit load instructions that use I-register indirect addressing**
  - **General Form**

    `DISALGNEXCPT` **(used in parallel with memory loads)**
  - **Example**

```
// i0 is FF80 0001 (byte-aligned)
// i1 is FF80 0008 (4-byte-aligned)

// The instruction below will cause an exception due to alignment of i0
r1 = [i0++] || r3 = [i1++];

// The instruction below will disable this exception before doing the
   memory load
DISALGNEXCPT || r1 = [i0++] || r3 = [i1++];
```

Addition

- BYTEOP16P (Quad 8-bit Add)
  - Adds eight unsigned bytes to result in four 16-bit words
- General Form
  - (dest_reg_1, dest_reg_0) = BYTEOP16P(src_reg_0, src_reg_1) [(*R*)]
  - source data chosen by I0 and I1 from register pairs R3:2 and R1:0

	31:24	23:16	15:8	7:0
aligned src_reg_0	y3	y2	y1	y0
aligned src_reg_1	z3	z2	z1	z0

dest_reg_0	y1+z1	y0+z0
dest_reg_1	y3+z3	y2+z2

- Example
  - (r1, r2) = BYTEOP16P(r3:2, r1:0);

Addition Example

// i0 = 0x0000 0000

// i1 = 0x0000 0000

// r3 = 0x0F0D 0B09, r2 = 0x0705 0301

// r1 = 0x0E0C 0A08, r0 = 0x0604 0200

(r1, r2) = BYTEOP16P(r3:2, r1:0);

	31:24	23:16	15:8	7:0
aligned src_reg_0	0x07	0x05	0x03	0x01
aligned src_reg_1	0x06	0x04	0x02	0x00

r2	0x03 + 0x02 = 0x0005	0x01 + 0x00 = 0x0001
r1	0x07 + 0x06 = 0x000D	0x05 + 0x04 = 0x0009

Subtraction

- BYTEOP16M (Quad 8-bit Subtract)
  - Subtracts eight unsigned bytes to result in four sign-extended 16-bit words
- General Form
  - (dest_reg_1, dest_reg_0) = BYTEOP16M(src_reg_0, src_reg_1) [(*R*)]
  - source data chosen by I0 and I1 from register pairs R3:2 and R1:0

	31:24	23:16	15:8	7:0
aligned src_reg_0	y3	y2	y1	y0
aligned src_reg_1	z3	z2	z1	z0

dest_0	y1-z1	y0-z0
dest_1	y3-z3	y2-z2

- Example
  - (r1, r2) = BYTEOP16M(r3:2, r1:0);

Subtraction Example

// i0 = 0x0000 0000

// i1 = 0x0000 0001

// r3 = 0x0F0D 0B09, r2 = 0x0705 0301

// r1 = 0x0C09 0908, r0 = 0x0604 0200

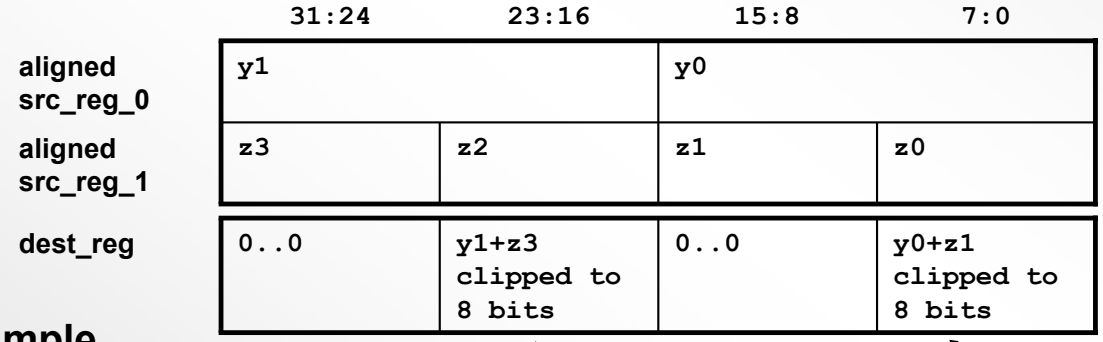
(r1, r2) = BYTEOP16M(r3:2, r1:0) (r);

	31:24	23:16	15:8	7:0
aligned src_reg_0	0x0F	0x0D	0x0B	0x09
aligned src_reg_1	0x00	0x0C	0x09	0x09

r2	0x0B - 0x09 = 0x0002	0x09 - 0x09 = 0x0000
r1	0x0F - 0x00 = 0x000F	0x0D - 0x0C = 0x0001

Addition with Clipping

- BYTEOP3P (Dual 16-bit Add/Clip)
  - Adds two 8-bit unsigned values to two 16-bit signed values, and limits the result to the 8-bit range [0,255]
- General Form
  - dest_reg = BYTEOP3P(src_reg_0, src_reg_1) (*opt*)
  - source data chosen by I0 and I1 from register pairs R3:2 and R1:0



- Example
  - r3 = BYTEOP3P(r1:0, r3:2) (lo);
    - (lo) loads the lower bytes in the half-words
    - (hi) loads the upper bytes in the half-words

Addition with Clipping Example

// i0 = 0x0000 0001

// i1 = 0x0000 0002

// r3 = 0x0F0D 0B09, r2 = 0x0705 0301

// r1 = 0x0101 0100, r0 = 0x0100 FF01

r4 = BYTEOP3P(r1:0, r3:2) (lo);

	31:24	23:16	15:8	7:0
aligned src_reg_0	0x0001		0x00FF	
aligned src_reg_1	0x0B	0x09	0x07	0x05
r4	0x00 (zero-filled)	0x0001 + 0x0B = 0x0C	0x00 (zero-filled)	0x00FF + 0x07 = 0x106 -> (clipped to 0xFF)

Quad-Byte Averaging (1)

- BYTEOP1P (Quad 8-bit Average – Byte)
- Averages four unsigned byte pairs to produce four 8-bit results
- General Form
  - dest_reg = BYTEOP1P(src_reg_0, src_reg_1) [(*opt*)]
  - source data chosen by I0 and I1 from register pairs R3:2 and R1:0

	31:24	23:16	15:8	7:0
aligned src_reg_0	y3	y2	y1	y0
aligned src_reg_1	z3	z2	z1	z0
dest_reg	avg(y3,z3)	avg(y2,z2)	avg(y1,z1)	avg(y0,z0)

- Example
  - r5 = BYTEOP1P(r1:0, r3:2);

Quad-Byte Averaging (1) Example

// i0 = 0x0000 0001

// i1 = 0x0000 0000

// r3 = 0x0F0D 0B09, r2 = 0x0705 0301

// r1 = 0x0E0C 0A08, r0 = 0x0604 0200

r5 = BYTEOP1P(r1:0, r3:2) (t); // (t) flag for result truncation

	31:24	23:16	15:8	7:0
aligned src_reg_0	0x08	0x06	0x04	0x02
aligned src_reg_1	0x07	0x05	0x03	0x01
R5	0x07	0x05	0x03	0x01

Quad-Byte Averaging (2)

- BYTEOP2P (Quad 8-bit Average – Half-Word)
  - Averages two unsigned byte quadruples to produce two 8-bit results
- General Form
  - dest_reg = BYTEOP2P(src_reg_0, src_reg_1) (*opt*)
  - source data chosen by I0 only from register pairs R3:2 and R1:0

	31:24	23:16	15:8	7:0
aligned src_reg_0	y3	y2	y1	y0
aligned src_reg_1	z3	z2	z1	z0
dest_reg	0..0	avg(y3,y2,z3,z2)	0..0	avg(y1,z1,y0,z0)

- Example
  - r6 = BYTEOP2P(r1:0, r3:2) (RNDL);
    - // RNDL = round up, and load the result into the lower bytes
- The I0 register aligns both src_reg_0 and src_reg_1!

Quad-Byte Averaging (2) Example

- // i0 = 0x0000 0003 // the i0 register aligns both src_reg_0 and src_reg_1
- // r3 = 0x0F0D 0B09, r2 = 0x0705 0301
- // r1 = 0x0E0C 0A08, r0 = 0x0604 0200

- r6 = BYTEOP2P(r1:0, r3:2) (RNDL);

	31:24	23:16	15:8	7:0
aligned src_reg_0	0x0D	0x0B	0x09	0x07
aligned src_reg_1	0x0C	0x0A	0x08	0x06
R6	0x00	0x0C	0x00	0x08

Quad-Byte-Sum Absolute Difference (1)

- **SAA (Quad 8-bit Subtract-Absolute-Accumulate)**
  - **Subtracts four pair of bytes, takes the absolute value of each difference, and accumulates each result into a 16-bit accumulator half**

$$SAD = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} \left| a(i,j) - b(i,j) \right|$$

  - ***N* is typically 8 or 16 (corresponding to blocks of 8x8 and 16x16 pixel, respectively)**
  - **Useful for block-based video motion estimation**

Quad-Byte-Sum Absolute Difference (2)

- General Form
  - SAA(src_reg_0, src_reg_1) [(*opt*)]
  - source data chosen by I0 and I1 from register pairs R3:2 and R1:0

	31:24	23:16	15:8	7:0
aligned src_reg_0	a(i,j+3)	a(i,j+2)	a(i,j+1)	a(i,j)
aligned src_reg_1	b(i,j+3)	b(i,j+2)	b(i,j+1)	b(i,j)

A0 (H:L)	+=|a(i,j+1)-b(i,j+1|	+=|a(i,j)-b(i,j)|
A1 (H:L)	+=|a(i,j+3)-b(i,j+3)|	+=|a(i,j+2)-b(i,j+2)|

- Example
  - // used in a loop that iterates over an image block
  - SAA(r1:0, r3:2) || r0 = [i0++] || r2 = [i1++];

Dual 16-bit SAA Accumulator Extract

- Dual 16-bit Accumulator Extraction with Addition
  - Adds the two upper half-words and the two lower half-words of each accumulator, and places each result in a 32-bit data register
  - Used to format the data for the Quad 8-bit Subtract-Absolute-Accumulate instruction
- General Form

  dest_reg_1 = a1.l + a1.h, dest_reg_0 = a0.l + a0.h
- Example

  r4 = a1.l + a1.h, r7 = a0.l + a0.h;

Quad-Byte Pack

- **BYTEPACK (Quad 8-bit Pack)**
  - **Prepares data for 8-bit ALU operations**
- **General Form**

  dest_reg = BYTEPACK(src_reg_0, src_reg_1)

reg_0		byte1		byte0
reg_1		byte3		byte2

dest_reg	byte3	byte2	byte1	byte0

- **Example**

```
/* r3 = 0x0034 0012, r4 = 0x0078 0056 */
r2 = BYTEPACK(r3, r4);
/* r2 = 0x7856 3412 */
```

Quad-Byte Unpack

- **BYTEUNPACK (Quad 8-bit Unpack)**
  - **Inverse of BYTEPACK, but includes an additional alignment mechanism**
  - **General Form**
    - **(dest_reg_1, dest_reg_2) = BYTEUNPACK src_reg_pair**
    - **source data chosen by I0 from register pairs R3:2 and R1:0**
  - **Example**

R1				R0			
FE	ED	FA	CE	BE	EF	BA	DD

I0 LSBs = 00b

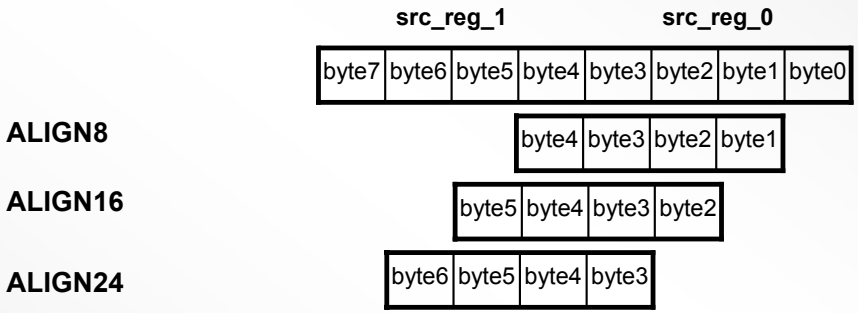
BE	EF	BA	DD

- **(r6,r5) = BYTEUNPACK r1:0;**

R6				R5			
00	BE	00	EF	00	BA	00	DD

Byte Alignment

- ALIGN8, ALIGN16, ALIGN24
  - In general, 32-bit accesses must be 32-bit aligned, and 16-bit accesses must be 16-bit aligned; Otherwise, an exception will occur. The ALIGNx utility instructions help to realign data with byte granularity. These instructions are useful when only alignment, and not arithmetic, are required
  - Copy a contiguous four-byte unaligned word from a combination of two registers

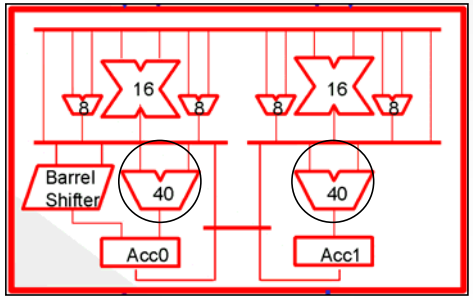


  - General Form
    - dest_reg = ALIGNx(src_reg_1, src_reg_0)
  - Example

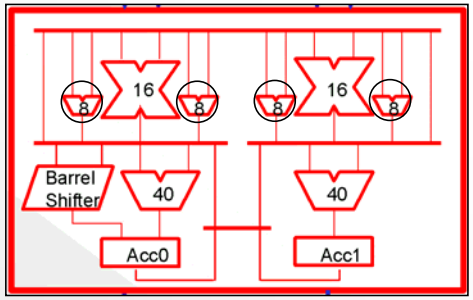
```
/* r3 = 0xABCD 1234, r4 = 0xBEEF DEAD */
      r0 = align8(r3, r4);
/* r0 = 0x34BE EFDE */
```

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



Vector Operations

Vector Operations

- Vector instructions perform two simultaneous operations on 16-bit values
- The following are the supported vector instructions

Add on Sign	VIT_MAX (Compare-Select)
ABS	Add/Subtract
Arithmetic/Logical Shift	MAX/MIN
Multiply/Multiply-Accumulate	SEARCH
Negate	PACK

(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

Vector Add/Subtract

- **Dual 16-bit operations**
  - **Implicit vector syntax, no (v) needed**

```
r5 = r3+|+r4;
```

- **Quad 16-bit operations (identical inputs)**
  - **Implicit vector syntax, no (v) needed**

```
r5 = r3+|+r4, r7 = r3-|-r4;
```

- **Dual 32-bit operations (identical input)**

```
r2 = r0 + r1, r3 = r0 - r1;
```

- **Dual 40-bit accumulator operations (identical inputs)**
  - **The result is 32 bits**

```
r4 = a1 + a0, r6 = a1 - a0;
```

Multiply/Multiply-Accumulate

- **Vector multiply (simultaneous MAC0 and MAC1 execution)**

```
r2.h = r7.l*r6.h, r2.l = r7.h*r6.h;
```

- **Multiply-accumulate (result is 40-bit accumulator)**

```
a1 += r2.l*r3.h, a0 += r2.h*r3.h;
```

- **Multiply-accumulate (result is 16-bit half Dreg)**

```
r2.h = (a1+=r7.l*r6.h), r2.l = (a0+=r7.h*r6.h);
```

- **Multiply-accumulate (result is 32-bit Dreg)**

```
r3 = (a1+=r6.h*r7.h), r2 = (a0+=r6.l*r7.l);
```

- **Note: The above operations all support the normal arithmetic options (FU, IS, IU, T, TFU, S2RND, ISS2, M)**

Vector Search

- Vector SEARCH
  - Used in a loop to locate a maximum or minimum element in an array of 16-bit packed data
  - Modes supported: > (GT), >= (GE), < (LT), <= (LE)
  - Compares the high and low 16-bit, signed half-words in the source register to the values in the accumulators, and updates the accumulators with those values that satisfy the criteria
  - The destination registers contain the addresses of the last value pair to update the accumulators (this address must reside in p0)

```
LSETUP(loop, loop) lc0 = p1>>1;  // set up the loop
   /* search for the last minimum in all but the last element of the
   array */
   loop: (r1,r0) = SEARCH r2 (LE) || r2 = [p0++];
(r1,r0) = SEARCH r2 (LE);
```

Issuing Parallel Instructions

Variable Instruction Lengths

- **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 are 16 bits long due to the importance of code density**

    r3.l = w[i3++];
- **32-bit instructions**
  - **most control-type instructions with a literal value in the expression are 32 bits long**

    [p0 + 4368] = r0;
  - **most arithmetic instructions are 32 bits in length**

    r3.l = r3.h * r2.h;
- **Multi-issue 64-bit instructions**
  - **certain 32-bit instructions can be issued simultaneously with a pair of specific 16-bit instructions to perform three distinct operations from one 64-bit instruction**
  - **The delimiter symbol to separate instructions in a multi-issue instruction is a double pipe character "||"**

Issuing 64-Bit 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 Processor Instruction Set Reference contains a table that lists which instructions can be placed in a particular multi-issue slot

Parallel Issue Examples

- Below are a few examples of 64-bit multi-issue instructions:
  - 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++];

r1 = [i0++] || r3 = [i1++];          // an implicit MNOP is placed in the 32-bit slot
                                     // by the assembler
```

  - 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

Code Optimization

Avoiding Stalls

- Most common numeric operations have no instruction latency, but a combination of instructions that are close to each other may cause stalls to occur
- Application Note EE-197 provides instruction combinations with associated stall info
- Use the Pipeline Viewer to identify stall conditions and eliminate these conditions

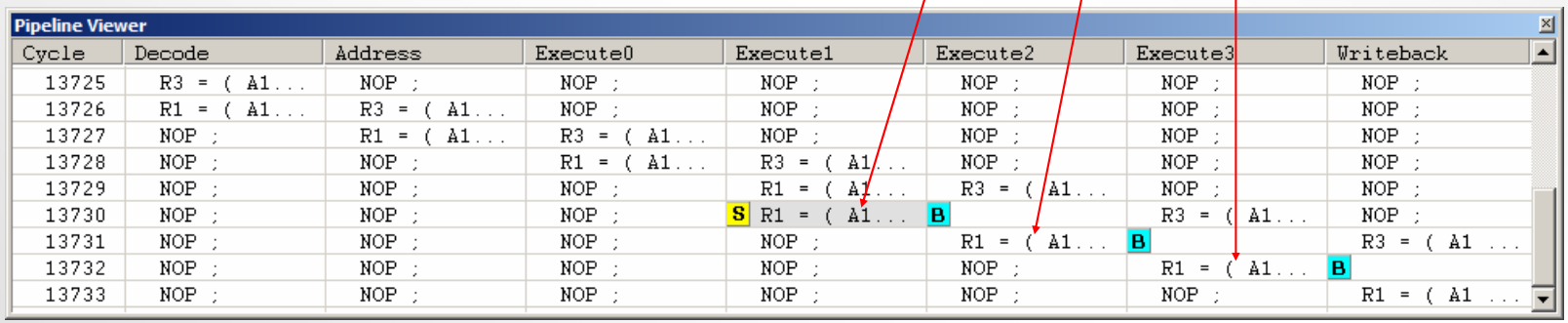
Simple Stall Examples

- Use of a P-reg loaded in the previous instruction causes a 3 cycle stall
  - p0=[p1++];
  - r0=[p0];
- Use of a P-reg which was transferred from D-reg in the previous instruction causes a 4 cycle stall
  - p0=r0;
  - p1=p0+p2;
- Use of a DAG-reg which was transferred from a D-reg causes 4 cycle stall
  - i0=r0;
  - r1=[i0++];
- Back to back multiplication where the result of first multiplication is used as an operand of the second multiplication causes 1 cycle stall
  - r3 = (a1+=r1.l*r2.l);
  - r1 = (a1+=r3.l*r2.l);

VisualDSP++ Pipeline Viewer

- **Integrated pipeline viewer in the VisualDSP++ simulator helps find *stalls*, *kills*, and *multi-cycle* instructions**
- **Display formats: Address, Disassembly, or Opcode**

Inst. Fetch1	Inst. Fetch2	Inst. Fetch3	Inst. Decode	Address Calc	Ex1	Ex2	Ex3	WB	
	Inst. Fetch1	Inst. Fetch2	Inst. Fetch3	Inst. Decode	Address Calc	Ex1	Ex2	Ex3	WB



Reference Material

Advanced Instructions

Viterbi Instructions - VIT_MAX

- VIT_MAX
  - used in the Add-Compare-Select function of Viterbi decoders

dest_reg = VIT_MAX(src_reg_0, src_reg_1) (ASL);

dest_reg = VIT_MAX(src_reg_0, src_reg_1) (ASR);

src_reg_0	y1	y0
src_reg_1	z1	z0

dest	max(y1,y0)	max(z1,z0)

A0 if ASL	00000000	XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXBB
A0 if ASR	00000000	BBXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

BB=00	z0 and y0 are maxima
BB=01	z0 and y1 are maxima
BB=10	z1 and y0 are maxima
BB=11	z1 and y1 are maxima

VIT_MAX Example

```
// r3 = 0xFFFF 0000
// r2 = 0x0000 FFFF
// a0 = 0x00 0000 0000

r5 = VIT_MAX(r3, r2) (ASL);

// r5 = 0x0000 0000
// a0 = 0x00 0000 0002
```

Viterbi Instructions - Add on Sign

- Add on Sign
  - used to compute the branch metric in each butterfly

  dest_hi = dest_lo = SIGN(src0_hi)*src1_hi + SIGN(src0_lo)*src1_lo

src_reg_0	a1	a0
src_reg_1	b1	b0
dest	(sign_adjusted_b1) + (sign_adjusted_b0)	(sign_adjusted_b1) + (sign_adjusted_b0)

- Example

```
// r2.h = -2, r3.h = 23
// r2.l = -2001, r3.l = 1234
r7.h = r7.l = SIGN(r2.h)*r3.h + SIGN(r2.l)*r3.l;
// r7.h = r7.l = -1257
```

Miscellaneous Arithmetic Operations

32-Bit Multiply

- The Blackfin instruction set includes an instruction to multiply to 32-bit integers
- This instruction cannot be used to multiply fractional data. However, EE-186 describes a method to perform 32-bit fractional multiplication in multiple instructions
- General Form
  - dest_reg *= multiplier_reg;
- Example
  - r3 *= r0;
- The functional description of this instruction is equivalent to the following
  - dest_reg = (dest_reg * multiplier_reg) % 2^{32}
- There is no built-in mechanism to detect overflows
- A common application to this instruction is random number generation by the congruence method

Division (DIVS, DIVQ)

- Two divide primitive instructions (DIVS and DIVQ) are used in the nonrestoring conditional add-subtract division algorithm
- The dividend is a 32-bit value and the divisor is a 16-bit value
- DIVS
  - Initialize for DIVQ. Set the AQ flag based on the signs of the 32-bit dividend and the 16-bit divisor. Left shift the dividend 1 bit. Copy AQ into the dividend LSB
- General Form
  - DIVS(dividend_register, divisor_register)
- DIVQ
  - Based on the AQ flag, either add or subtract the divisor from the dividend. Then set the AQ flag based on the MSBs of the 32-bit dividend and the 16-bit divisor. Left shift the dividend one bit. Copy the logical inverse of AQ into the dividend LSB.
- General Form
  - DIVQ(dividend_register, divisor_register)

Signed Division Example

- The following example shows how to compute a division using a signed integer and divisor

```
p0 = 15 ; /* Evaluate the quotient to 16 bits. */
r0 = 70 ; /* Dividend, or numerator */
r1 = 5 ;  /* Divisor, or denominator */
r0 <<= 1 ; /* Left shift dividend by 1 needed for integer
           division */
divs (r0, r1) ; /* Evaluate quotient MSB. Initialize AQ
           flag and dividend for the DIVQ loop. */
loop .div_prim lc0=p0 ; /* Evaluate DIVQ p0=15 times. */
loop_begin .div_prim ;
divq (r0, r1) ;
loop_end .div_prim ;
r0 = r0.l (x) ;  /* Sign extend the 16-bit quotient to
           32bits. */
/* r0 contains the quotient (70/5 = 14). */
```