

Blackfin Programming Skills

Agenda

◆ Day1

- | | |
|-------------|------------------------------|
| 9:00~12:00 | Blackfin 高级编程技术 |
| 12:00~13:00 | 午餐 |
| 13:00~14:00 | Blackfin 高级编程技术 |
| 14:00~15:30 | Blackfin DMA |
| 15:30~17:00 | Blackfin Memory/cache |

◆ Day2

- | | |
|-------------|---------------------------|
| 9:00~12:00 | Blackfin 外设 |
| 12:00~13:00 | 午餐 |
| 13:00~15:00 | Blackfin 系统设计 |
| 16:00~17:00 | Blackfin 工程开发与应用实例 |

Blackfin Advanced Programming Skills

Agenda

- ◆ **Blackfin Core Review**
- ◆ **Blackfin C/C++ Programming Skills & Hands on**
- ◆ **C and Assembly Mixed Programming & Hands on**
- ◆ **Blackfin Assembly Programming Skills & Hands on**

Blackfin Core Review

PROCESSOR CORE OVERVIEW

Abstract

◆ Traditional DSPs Core Architecture

-  DSP SHARC family DSPs
- 21XX ADSP-21xx family DSPs

◆ MSA Core Architecture

-  blackfin family embedded processors
-  TigerSHARC family embedded processors



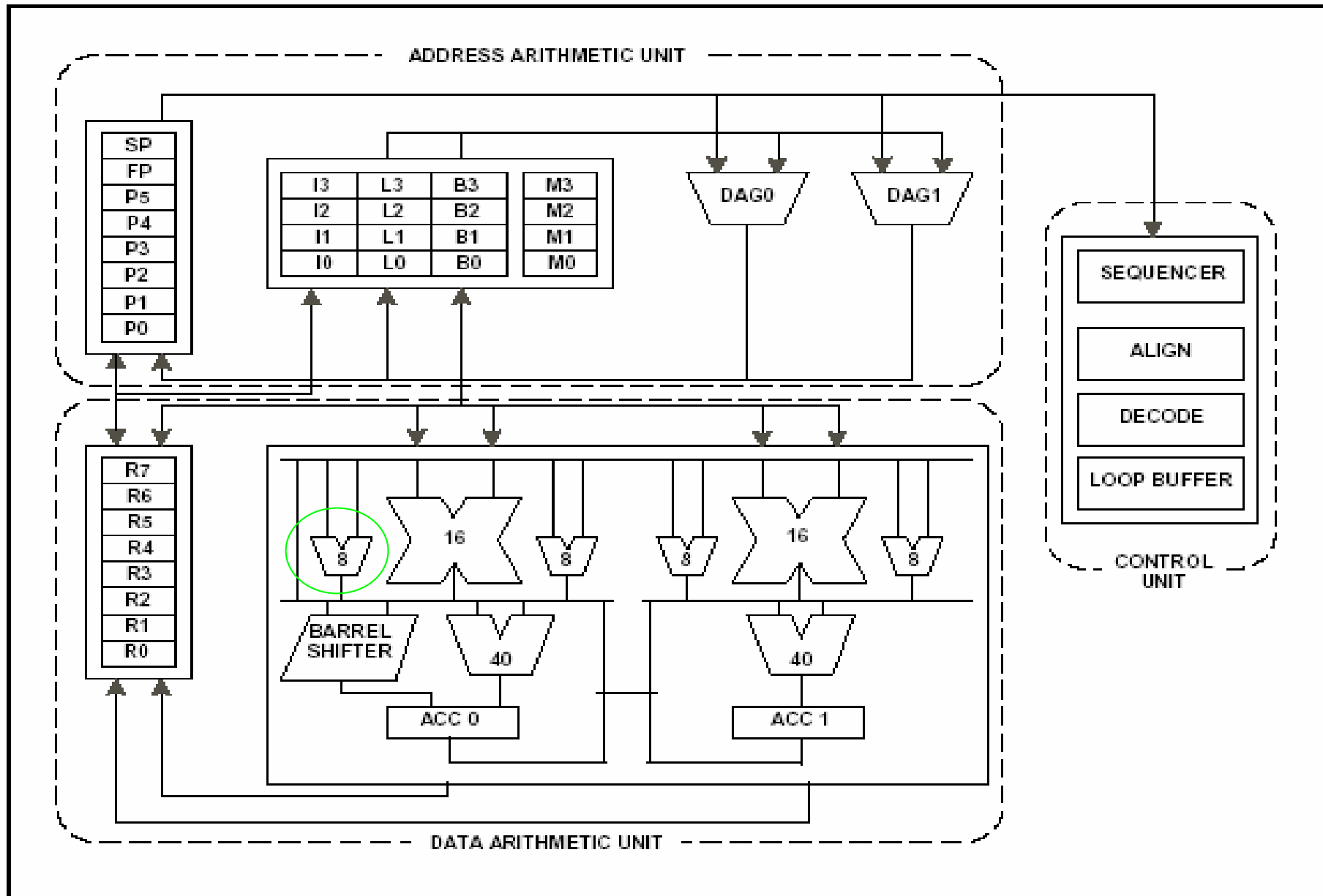
MSA Core Architecture

Highlight Feature

- ◆ **Orthogonal RISC-like Microprocessor Instruction Set**
- ◆ **Single-Instruction Multiple-Data (SIMD)**
- ◆ **Dynamic Power Management**
- ◆ **64-bit-wide Instruction-Fetch Bus**
- ◆ **ten-Stage Instruction Pipeline**
- ◆ **Multi-issue 64-bit instructions (VLIW)**

MSA Core Architecture

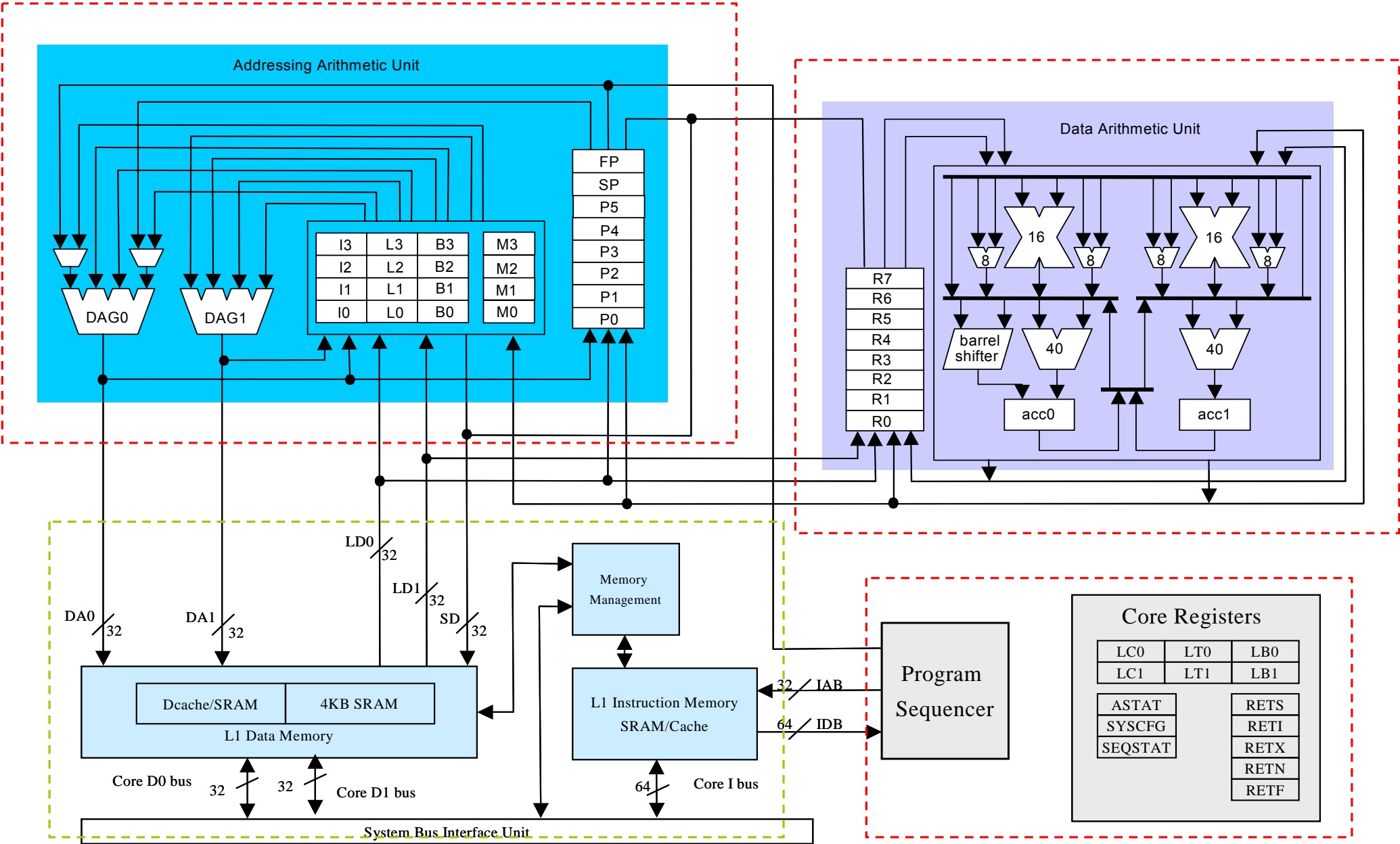
Modified Harvard Architecture



MSA Core Architecture Advantage

- ◆ **Optimized Instruction Set**
 - High density compiled code
- ◆ **Limited multi-issue capability**
 - Use many of the core resources in a single instruction cycle
- ◆ **Algebraic syntax**
 - Easy programming & read
- ◆ **Optimized Linker of C/C++ Compiler**
 - More sufficient software environment

ADSP-BF533 Core



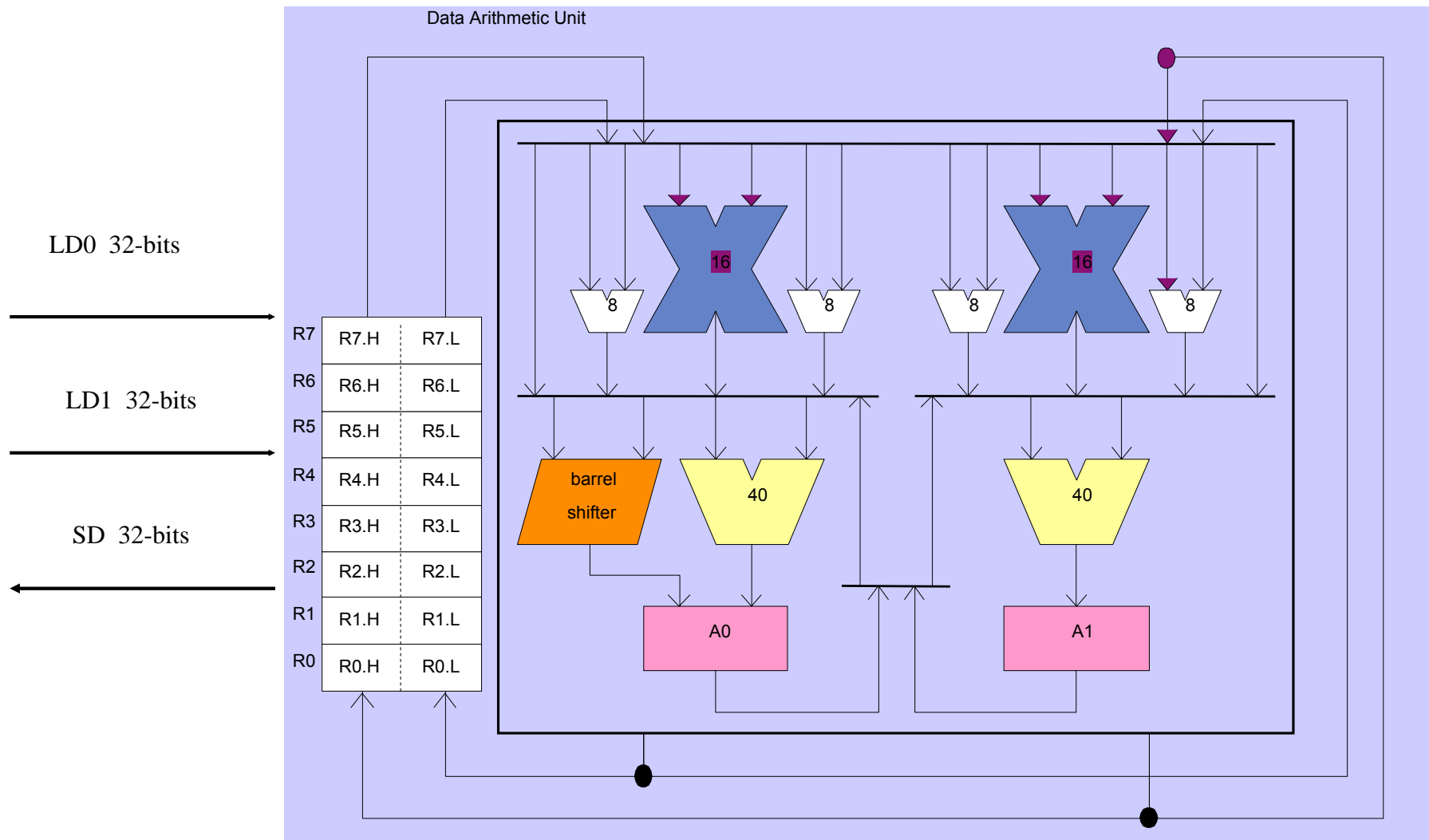
Registers and Register File

Accessing Registers

- ◆ **Blackfin Processors are register-intensive devices**
 - All computations are performed on data contained in registers
 - All peripherals are setup using registers
 - Memory is accessed using pointers in address registers

- ◆ **There are two ways to access registers on the ADSP-BF533**
 - **Core registers to be accessed directly by name**
 - ◆ R0-R7, A0-A1, P0-P5, FP, SP, USP, I0-I3, SYSCFG,
 - **Memory-mapped registers (MMRs)**
 - ◆ Core MMRs and System MMRs

Arithmetic Logic Unit (ALU) Architecture



Arithmetic Logic Unit (ALU)

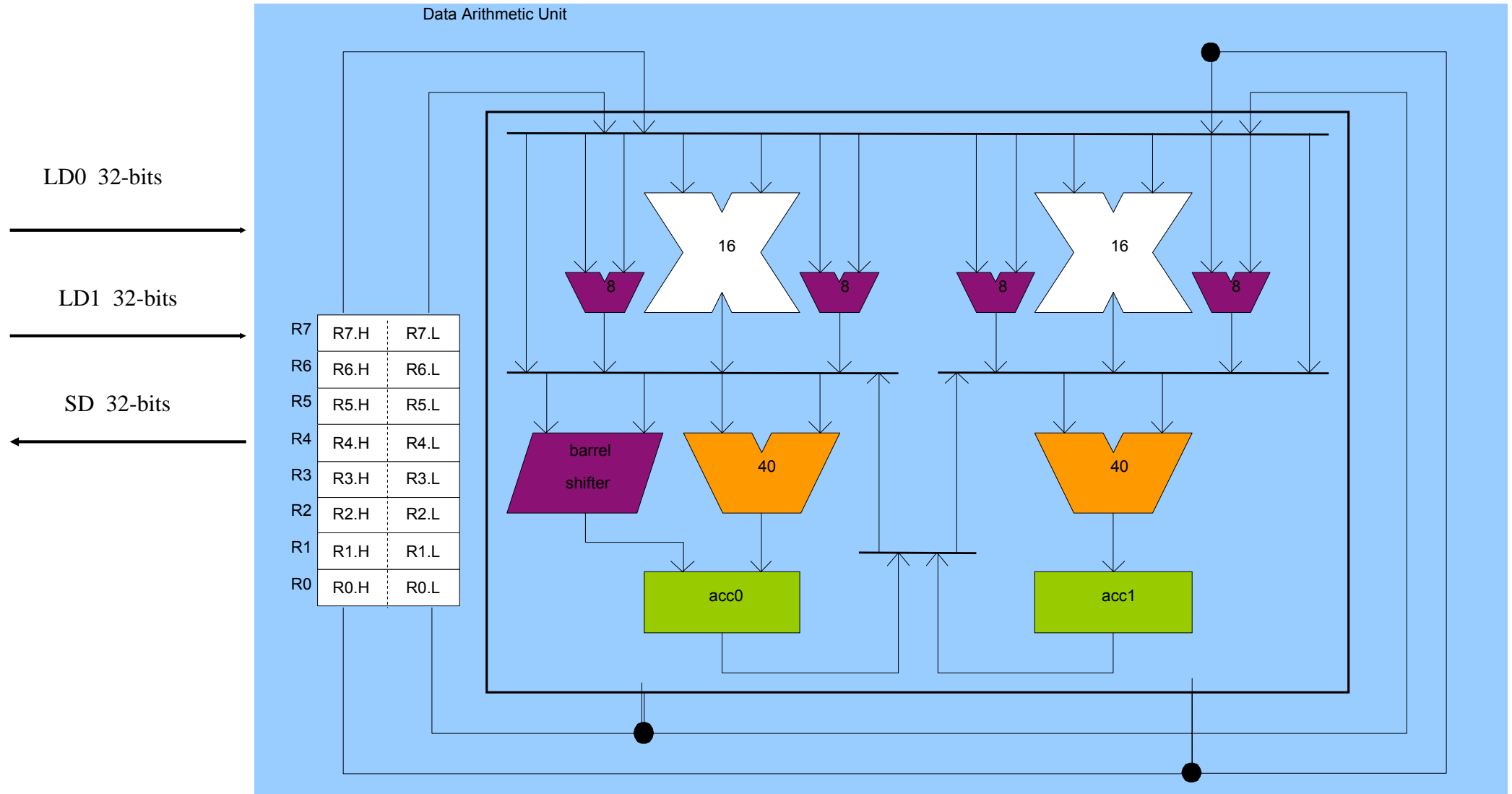
Review

- ◆ **Two 40-bit ALUs and four 8-bit video ALUs**

- ◆ **40-bit ALUs 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
 - Supports conditional instructions

- ◆ **40-bit ALU operations support the following operations:**
 - Single / Dual / Quad 16-Bit Operations
 - Single / Dual 32-Bit Operations

Multiply-Accumulators (MAC) Architecture



Multiply-Accumulators (MAC)

Review

◆ Two identical MACs

- Each can perform fixed point multiplication and multiply-and-accumulate operations on 16-bit fixed point input data and outputs 32-bit or 40-bit results depending the destination.

◆ Functions

- Multiplication
- Multiply-and-accumulate with addition (optional rounding)
- Multiply-and-accumulate with subtraction (optional rounding)
- Dual versions of the above

◆ Features

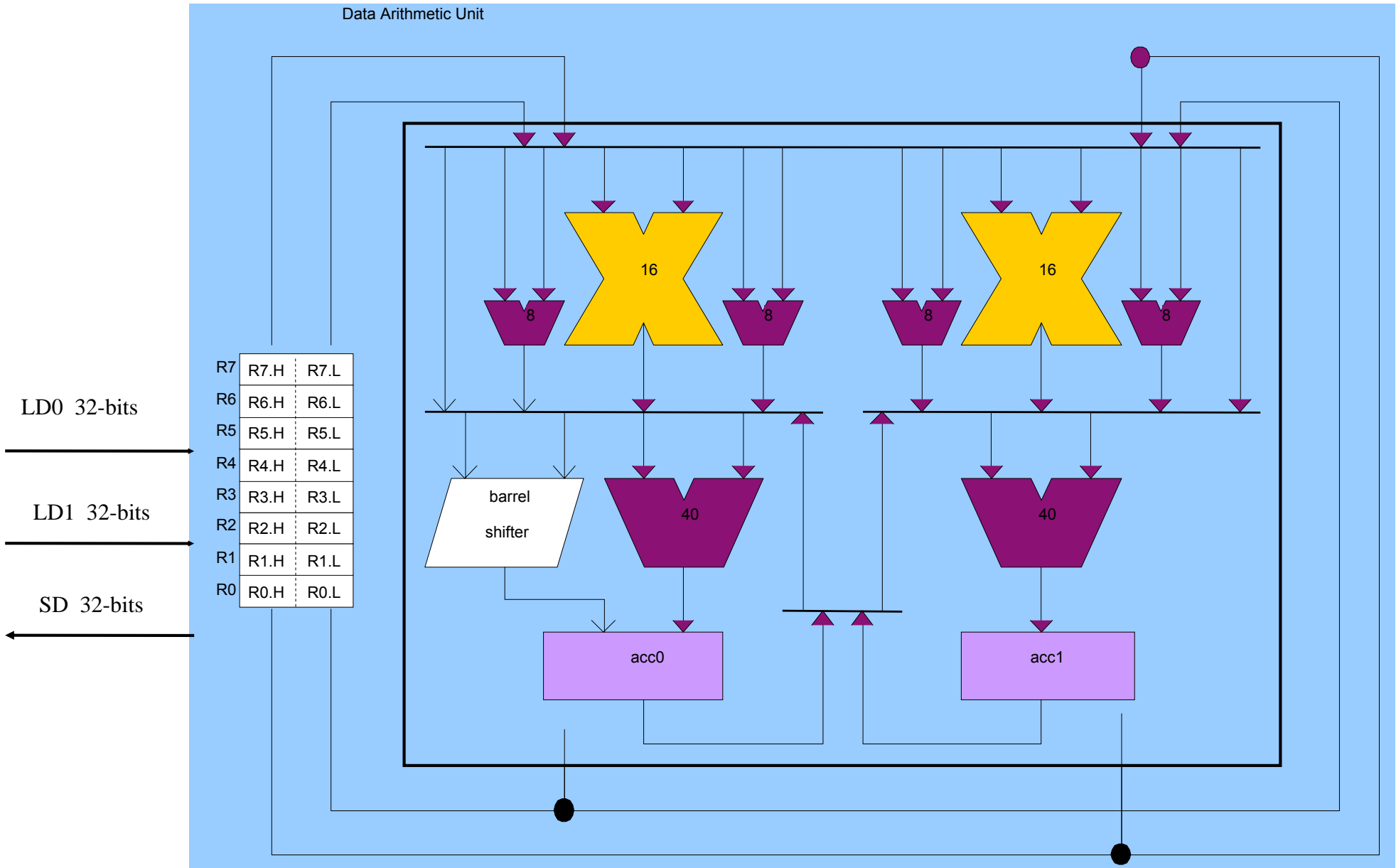
- Saturation of accumulator results
- Optional rounding of multiplier results

◆ Modes

- Fractional / Integer / Signed / Unsigned

Barrel-Shifter (Shifter) Architecture

Architecture



Barrel-Shifter (Shifter)

Overview

- ◆ **The shifter performs bitwise shifting for 16-bit, 32-bit or 40-bit inputs and yields 16-bit, 32-bit, or 40-bit outputs.**

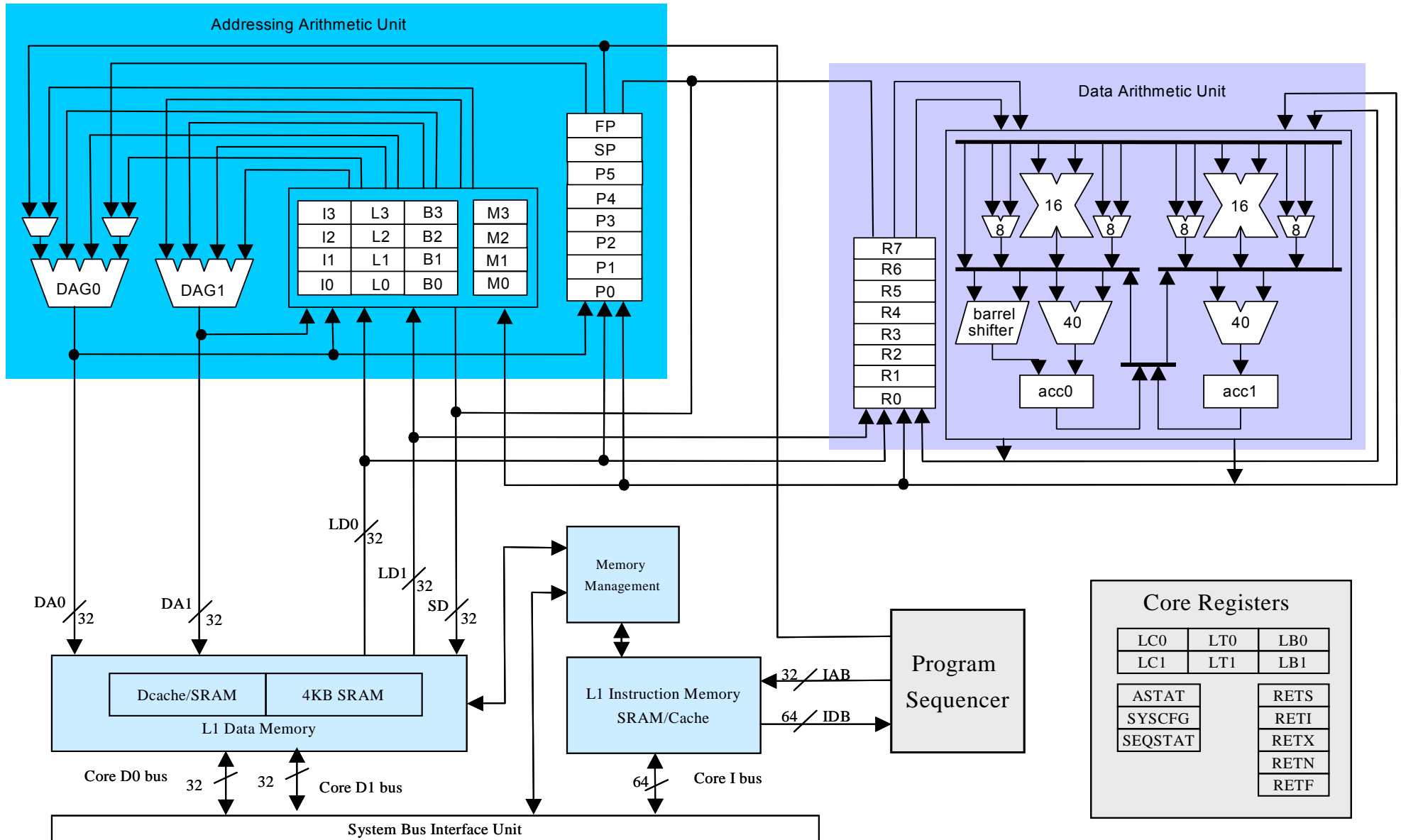
- ◆ **Functions**
 - **Arithmetic Shift**
 - **Logical Shift**

- ◆ **Operations**
 - **Rotate**

 - **Bit Operations**

 - **Field Extract and Deposit**

Memory Addressing

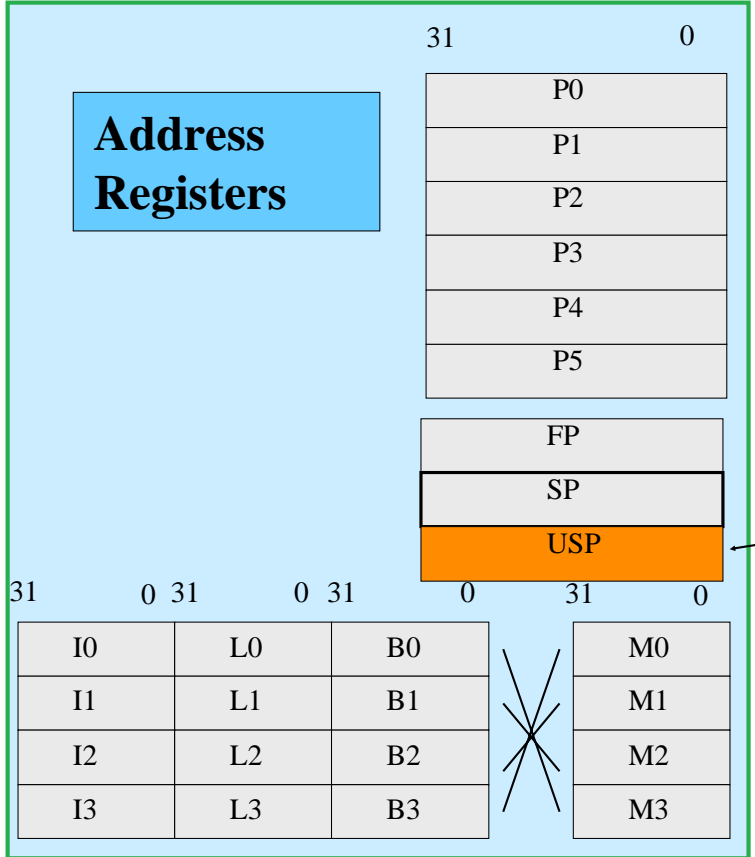


Memory Addressing Review

- ◆ **There are 4 units that generate addresses for memory accesses.**
 - Two Data Address Generators (DAG0 and DAG1)
 - The Program Sequencer
 - The DMA controller
- ◆ **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
- ◆ **All data accesses must be aligned to the data size**
- ◆ **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

Address Registers Review

- ◆ 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



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 Review

◆ 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
 - ◆ Bit Reversal (Modify only)
- **Pre-Modify with update (using Stack Pointer)**
- **Pre-Modify without update**

Program Sequencer Features

- ◆ **The Program Sequencer controls all program flow:**
 - **Maintains Loops, Subroutines, Jumps, Idle, Interrupts and Exceptions**
 - **Contains an 10-stage instruction pipeline**
 - **Includes Zero-Overhead Loop Registers**

Sequencer-Related Registers

Register Name	Description
SEQSTAT	Sequencer Status register
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
FR, SP	Frame Pointer and Stack Pointer.
SYSCFG	System Configuration Register
CYCLES, CYCLES2	Cycle Counters

Blackfin C/C++ Programming Skills

C and C++ Language Programming

- ◆ **Motivation: Portability, maintainability, time to market**
- ◆ **Full ANSI Language**
 - plus: // C++ style comments
 - other general programmability extensions
- ◆ **Full-featured library**
 - full standard math function support
 - additional DSP functions
 - basic I/O: printf, simple file I/O
- ◆ **Extensions tailored for DSP**
- ◆ **Highly effective optimizer**
- ◆ **Fully integrated into programming environment**
 - edit, build support
 - runtime system in place
 - source-language debugging

Optimization Control

- ◆ **Aim of compiler optimization:**
correct code + fast execution + small size
- ◆ **Optimization level**
 - Debug
 - Default
 - Procedural optimization
 - Profile-Guided Optimization (PGO)
 - Automatic Inlining
 - Interprocedural optimizations (IPA)

Optimization Control

◆ **Procedural Optimizations**

The compiler performs advanced, aggressive optimization on each procedure in the file being compiled.

◆ **Automatic Inlining**

The compiler automatically inlines C/C++ functions which are not necessarily declared as inline in the source code.

◆ **Interprocedural optimizations (IPA)**

Optimize across translation units instead of within individual translation units.

PGO (Profile-guided optimization)

- ◆ Profile-guided optimization (PGO) is a method of **tuning the compiler optimization strategy** for the typical run-time behavior of a program.
- ◆ PGO provides a means to inform the compiler about the application to obtain affect **branch prediction, improve loop transformations, and reduce code size.**
- ◆ The compiler can perform **advanced aggressive optimizations** using profiler statistics generated from running the application with representative data.
- ◆ PGO can be used in **conjunction** with interprocedural optimization (IPA) and automatic inlining.

C/C++ Programming DSP Specialize Support

◆ **Optimizer utilizes special hardware features automatically**

- inherent parallelism (multifunction instructions)
- zero-overhead loops
- MAC instructions

◆ **DSP-Language Extensions**

- DSP data types: complex, fract, saturating arithmetic

◆ **Intrinsics (built-in functions)**

- Efficient access to low-level machine capabilities
 - ◆ specialized operations
 - ◆ access system registers, I/O space, etc
 - ◆ global interrupt enable/disable
- uses library function model - fully portable syntax
- efficient: can easily compile into 1 or more instructions
- fully optimized

C/C++ Programming DSP Specialized Support (2)

◆ **Enhanced Library**

- **library implementations provided for additional types**

- ◆ float , long double
- ◆ complex (float, also fract16)
- ◆ fract16

- **DSP-specific functions available**

- ◆ basic vector and matrix operations
- ◆ filters and transforms
- ◆ statistical analyses

◆ **Easy to interface to assembly language routines.**

◆ **Inline asm() statement**

- **allows small amount of assembly language inserted into C for unusual situations**
- **intrinsics satisfy majority of cases, however**

C/C++ Compiler Language Extensions

supports extensions to the ANSI/ISO standard for the C and C++ languages

Keyword Extensions	Description
<code>inline</code>	Directs the compiler to integrate the function code into the code of its callers. For more information, see "Inline Function Support Keyword (inline)."
<code>asm()</code>	Places Blackfin core assembly language commands directly in your C/C++ program. For more information, see "Inline Assembly Language Support Keyword (asm)."
<code>bank (`string`)</code>	Specifies a name which the user assigns to associate declarations that reside in particular memory banks. For more information, see "Bank Type Qualifiers."
<code>section (`string`)</code>	Specifies the section in which an object or function is placed. For more information, see "Placement Support Keyword (section)."
<code>bool, true, false</code>	Specifies a Boolean type. For more information, see "Boolean Type Support Keywords (bool, true, false)."
<code>restrict</code>	Specifies restricted pointer features. For more information, see "Pointer Class Support Keyword (restrict)."

Placement of Code & Statical Variables

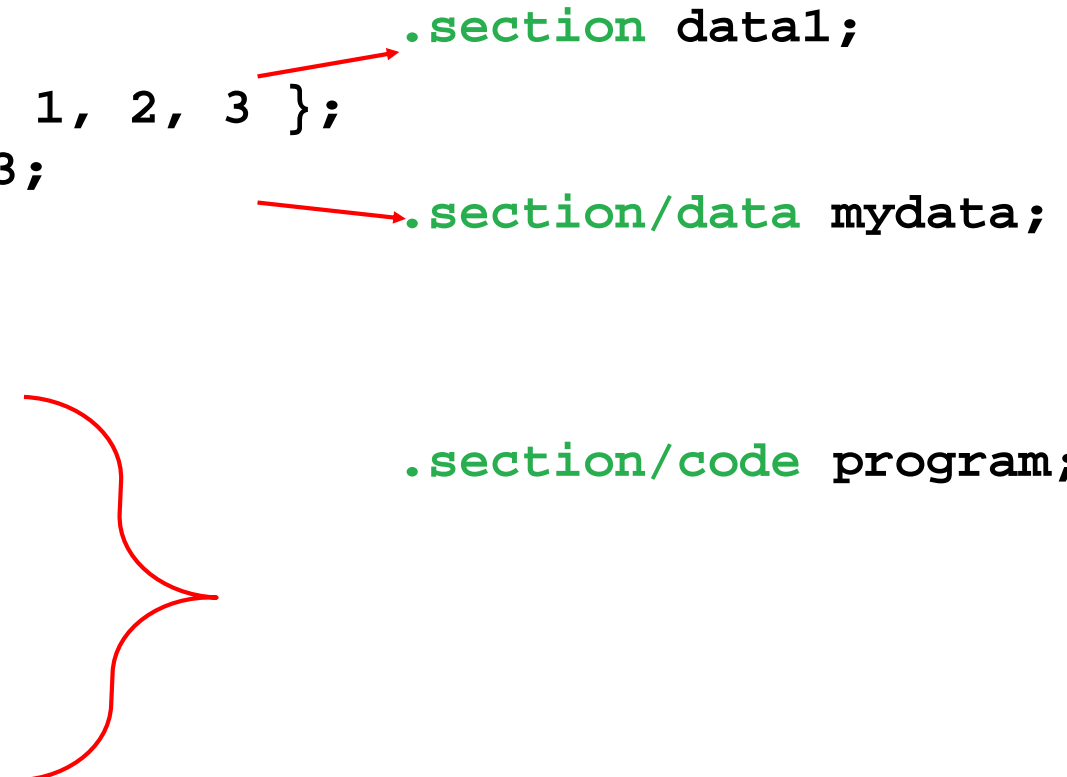
- ◆ Compiler uses default section names
- ◆ Section directive enables alternatives

```
#define N 3  
  
section("data1") int x[N] = { 1, 2, 3 };  
section ("mydata") int myvar3;  
  
void main (void) {  
    int i; long z=0;  
    for (i=0; i<N; i++) {  
        z += x[i] * y[i];  
    }  
    myvar3 = z;  
}
```

`.section data1;`

`.section/data mydata;`

`.section/code program;`



Handle sections in LDF file

MEMORY

```
{  
PROGRAM { TYPE(RAM) START(0xFFA00000) END(0xFFA04FFF) WIDTH(8) }  
  DATA_A      { TYPE(RAM) START(0xFF800000) END(0xFF807FFF) WIDTH(8) }  
  DATA_B      { TYPE(RAM) START(0xFF900000) END(0xFF907FFF) WIDTH(8) }  
  SCRATCH      { TYPE(RAM) START(0xFFB00000) END(0xFFB00FFF) WIDTH(8) }  
  SDRAM_MEM1   { TYPE(RAM) START(0x00000000) END(0x07FFF) WIDTH(8) }  
}
```

SECTIONS

```
{  
program{ INPUT_SECTIONS( $OBJECTS(program) $LIBRARIES(program)) } >PROGRAM  
  data_a {INPUT_SECTIONS($OBJECTS(L1_data_a) $LIBRARIES(L1_data_a))} >DATA_A  
  data_b {INPUT_SECTIONS($OBJECTS(L1_data_b) $LIBRARIES(L1_data_b))} >DATA_B  
  scratch{INPUT_SECTIONS($OBJECTS(scratchpad) $LIBRARIES(scratchpad))} >SCRATCH  
  mydata {INPUT_SECTIONS( $OBJECTS(mydata) $LIBRARIES(mydata))}  
  >SDRAM_MEM1  
}
```

DSP-specific Libraries

- ◆ **#include <filter.h>**
 - FIRs / IIRs / Convolution
 - FFTs / IFFTs (complex, real, radix-4, 2-dimensional)
 - A-law / μ -law compression & expansion
- ◆ **#include <window.h>**
 - Rectangular, Triangle, Hamming, Hanning, Blackman
 - Harris, Kaiser, Gaussian, Bartlett, Von Hann
- ◆ **#include <stats.h>**
 - Autocorrelation, Histogram, Correlation
- ◆ **Prototypes use fract/complex convention**

```
void fir_fr16 ( const fract16 inp[], fract16 out[],  
               int samples, fir_state *s);
```

Some extra library functions

- assert (assert.h)

C code „assert(x!=0);“ prints

ASSERT [x!=0] fails at ".\main.c":63

to console window in debug mode

- stdlib.h

– qsort, bsearch, rand, div, ...

- math.h

– sin, cos, asin, sinh, asinh, tan, log, log10,

– fabs, clip, max, min

Tuning C/C++ source code

How the Compiler Can Help

- ◆ **Compiler optimizer**

- ◆ **Statistical profiler**

- ◆ **PGO**

 - more accurate branch-prediction, improved loop transformations, and reduced code-size

- ◆ **IPA optimizers**

 - Optimize across translation units instead of within individual translation units

Using the Optimizer and Statistical Profiler

- ◆ **Default setting as non-optimized compilation assist programmers in diagnosing problems with initial coding**
- ◆ **The basic strategy for tuning a program is to present the algorithm in a way that gives the optimizer greatest freedom**
- ◆ **Write your C code in straightforward and simple manner**
- ◆ **Tuning source begins with an understanding of the hot spots. Statistical profiling will help you on that**
- ◆ **If building fully optimized application, relating assembly lines to the original source will be a problem**

Avoid Integer Division in Loops

- ◆ **Blackfin hardware does not provide direct support for 32-bit integer division**
- ◆ **The compiler will convert an integer division by a power of 2**
- ◆ **Compiler will call to a library function to issue a full division operation**
- ◆ **This will prevent the optimizer from using HW loop**

Indexed Arrays vs. Pointers

- ◆ **C programmer has two ways to access data from an array:**
 - Index from an invariant base pointer
 - Increasing a pointer
- ◆ **The pointer style introduces additional variables while array accesses must be transformed to pointers by the compiler**
- ◆ **The best strategy is to start with array notation and try using pointer if necessary**

```
void va_ind( short a[], short b[],
            short out[], int n)
{
    int i;
    for (i = 0; i < n; ++i)
        out[i] = a[i] + b[i];
}
```

Listing 1: Indexed Arrays

```
void va_ptr( short a[], short b[],
            short out[], int n)
{
    int i;
    short *pout = out, *pa = a, *pb = b;
    for (i = 0; i < n; ++i)
        *pout++ = *pa++ + *pb++;
}
```

Listing 2: Pointers

Initialize Constant Statically

For example

```
#include <stdio.h>
int val;      // initialized to zero
void init() {
    val = 3;  // re-assigned
}
void func() {
    printf("val %d",val);
}
int main() {
    init();
    func();
}
```

Bad:

IPA cannot see that val is a constant

```
#include <stdio.h>
const int val = 3; // initialized once
void init() {
}
void func() {
    printf("val %d",val);
}
int main() {
    init();
    func();
}
```

Good:

IPA knows val is 3.

Function Inlining

- ◆ **Avoids various costs such as program flow latencies, function entry and exit instructions and parameter passing overheads.**

```
inline int add(int a, int b) {  
    return (a+b);  
}
```

Try to Put Arrays into Different Memory Sections

- ◆ **The Blackfin Processor can support two memory operations on a single instruction line**
- ◆ **To make use of it, you should put two arrays in different memory blocks**
- ◆ **Modify your LDF to implement this method**
- ◆ **This can only be done for global data**

```
for (i=0; i<100; i++)  
    sum += a[i] * b[i];
```

Bad: compiler assumes that two memory accesses together may give a stall.

```
section("bank_a1") short a[100];  
section("bank_a2") short b[100];
```

Loop Control

- ◆ **Tip: Use int types for loop control variables and array indices.**
- ◆ **Tip: Use automatic variables for loop control and loop exit test.**

```
for (i=0; i<globvar; i++)  
    a[i] = 10;
```

Bad: may need to reload globvar on every iteration.

```
int upper_bound = globvar;  
for (i=0; i<upper_bound; i++)  
    a[i] = 10;
```

Good: easily becomes hardware loop.

Avoiding Non-Unit Strides

```
for (i=0; i<n; i+=3) {  
    // some code  
}
```

Bad: non-unit stride means division may be required.

```
for (i=0; i<100; i++)  
    for (j=0; j<100; j++)  
        sum += a[i][j];
```

Good: memory accesses contiguous in inner loop.

Avoiding Conditional Code in Loops

```
for (i=0; i<100; i++) {  
    if (mult_by_b)  
        sum1 += a[i] * b[i];  
    else  
        sum1 += a[i] * c[i];  
}
```

Bad: loop contains conditional code.

```
if (mult_by_b) {  
    for (i=0; i<100; i++)  
        sum1 += a[i] * b[i];  
} else {  
    for (i=0; i<100; i++)  
        sum1 += a[i] * c[i];  
}
```

Good: two simple loops can be optimized well.

Using Built-In Functions in Code Optimization

```
long dot_product (short *a, short *b) {  
    int i;  
    long sum=0;  
    for (i=0; i<100; i++) {  
        /* this line is performance critical */  
        sum += (((long)a[i]*b[i]) << 1);  
    }  
    return sum;  
}
```

} **Bad: uses shifts to implement fractional multiplication.**

```
#include <math.h>  
fract32 dot_product(fract16 *a, fract16 *b) {  
    int i;  
    fract32 sum=0;  
    for (i=0; i<100; i++) {  
        /* this line is performance critical */  
        sum += __builtin_mult_fr1x32(a[i],b[i]);  
    }  
    return sum;  
}
```

} **Good: uses builtins to implement fractional multiplication.**

Hands on

Image _ processing

- ◆ **Port c program to visualDSP and configure LDF file.**
- ◆ **dump source image data to the Sdram memory, run c program and read cycle count.**
- ◆ **optimize c code step by step, then repeat read cycle count, compare cycle count .**

MEMORY

```
{ MEM_L1_DATA_B {  
  TYPE(RAM) WIDTH(8)  
  START(0xFF900100) END(0xFF907FFF)  
}  
MEM_L1_DATA_B_STACK {  
  TYPE(RAM) WIDTH(8)  
  START(0xFF900000) END(0xFF9000FF)  
}  
/*bsz_L1_data_b*/  
MEM_L1_DATA_A_CACHE {  
  TYPE(RAM) WIDTH(8)  
  START(0xFF806000) END(0xFF807FFF)  
}  
MEM_L1_DATA_A {  
  TYPE(RAM) WIDTH(8)  
  START(0xFF800000) END(0xFF805FFF)  
}  
/*data1*/  
}
```

PROCESSOR P0

{

.....

stack

{

ldf_stack_space = .;

ldf_stack_end = ldf_stack_space + MEMORY_SIZEOF(MEM_L1_SCRATCH);

} >MEM_L1_SCRATCH

.....

}

C and Assembly Mixed Programming

C/C++ and Assembly Interface

- ◆ **C-Callable Assembly Language Functions**
- ◆ **Assembly Language Statements Within a C Function (In-Line Assembly)**
- ◆ **Calling C/C++ Functions from Assembly Programs**
- ◆ **Associate C Variables with Assembly Language Symbols**

C-Callable Assembly Language Functions

- ◆ **Several Issues Involved When Writing C-Callable Assembly Language Functions**
 - **The name of any external entry point with an underscore .**
 - **Register Usage**
 - ◆ ***“Dedicated”* Registers**
 - ◆ ***“Call Preserved”* Registers**
 - ◆ ***“Scratch”* Registers**
 - **Argument Passing**
 - ◆ **First Three Arguments Passed in R0, R1 and R2, respectively**
 - ◆ **Arguments Four and Beyond Passed on Stack**
 - 4th Parameter Is Closest to SP at [FP+20], 5th at [FP+24], etc.
 - ◆ **Return Values of 32 Bits or Less Stored in R0**
 - Overflows To R1 for Return Values of 33 to 64 Bits
 - Anything Over 64 Bits Is Allocated on Stack but Passed as Pointer in a Hidden Argument in P0

C/C++ Compiler Register Uses

Dedicated Registers

◆ Registers that C/C++ Compiler Reserves for its Own Use

<i>REGISTER</i>	<i>VALUE</i>	<i>MODIFICATION RULES</i>
L0 – L3	0	See Note below
SP	Stack Pointer	Stack Management Only, Restore

FP

Frame Pointer

Stack Management Only, Restore

◆ L0-L3 Rules:

- ◆ The L0.L3 registers define the lengths of the DAG's circular buffers. The compiler makes use of the DAG registers, both in linear mode and in circular buffering mode. The compiler assumes that the Length registers are zero, both on entry to functions and on return from functions, and will ensure this is the case when it generates calls or returns. Your application may modify the Length registers and make use of circular buffers, but you must ensure that the Length registers are appropriately reset when calling compiled functions, or returning to compiled functions. Interrupt handlers must store and restore the Length registers, if making use of DAG registers.

C/C++ Compiler Register Uses

Call Preserved Registers

- ◆ **May be Used in an Assembly Function**
- ◆ **Contents Should Be Saved and Restored**
- ◆ **Values Assumed to be Preserved Across Function Calls**

Call-Preserved Registers Are:

P3 - P5

R4 - R7

C/C++ Compiler Register Uses

Scratch Registers

**Contents DO NOT Need to Be Saved/Restored
Use Freely in Assembly Sub-Routines**

P0	Used as the Aggregate Return Pointer.
P1—P2	
R0—R3	The first three words of the argument list are always passed in R0, R1 and R2 if present (R3 is not used for parameters)
LB0—LB1	
LC0—LC1	
LT0—LT1	
ASTAT	including CC
A0—A1	
I0—I3	
B0—B3	
M0—M3	

C-Callable Assembly Language Functions

◆ Macros in `asm_sprt.h` Provided to Make Function Calling Easier

- Save/Restore Preserved Registers (`pushs`, `pops`)
- Restore Frame and Stack Pointers (`exit`)

`pushs(x) ; // Save value in register onto stack`

`pushs(R5); -> [- -SP] = R5;`

`pops(x) ; // Read value off top of stack to a register`

`pops(R5); -> R5 = [SP++];`

`exit ; // Restore stack/frame pointers and jump to return address`

`exit; -> P0 = [FP + 0x4];
JUMP (P0);`

In-Line Assembly Language

◆ In-Line Assembly Is Accomplished Using the `asm()` Construct

Example:

```
asm("R0 = w[p0];");  
asm("BITSET(R0,7);");  
asm("ssync;");
```

Note: Can Produce Less Efficient Compiled Code – Optimizer Might Re-Sequence Instructions for Optimal Performance

Mixed C/Assembly Naming Conventions

<i>C PROGRAM</i>	<i>ASSEMBLY ROUTINE</i>
<code>int c_var; /* global variable */</code>	<code>.extern _c_var;</code>
<code>void c_func();</code>	<code>.extern _c_func;</code>
<code>extern int asm_var</code>	<code>.global _asm_var;</code>
<code>extern void asm_func();</code>	<code>.global _asm_func; _asm_func:</code>

To name an assembly symbol that corresponds to a C symbol, add an underscore prefix to the C symbol. Declare as a global variable in C program and as EXTERN in assembly routine

To use an assembly function or variable in your C program, declare the symbol with .GLOBAL directive in assembly routine and as EXTERN in the C program

Hands On

Image _ processing

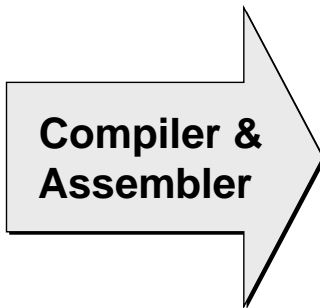
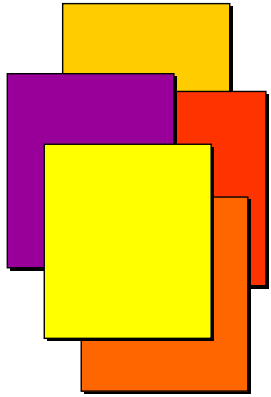
- ◆ **Open the corresponding .s file using a text editor.**
- ◆ **Compile C program and build .asm file.**
- ◆ **Call asm program from c project.**

Blackfin Assembly Programming Skills

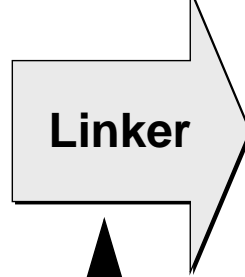
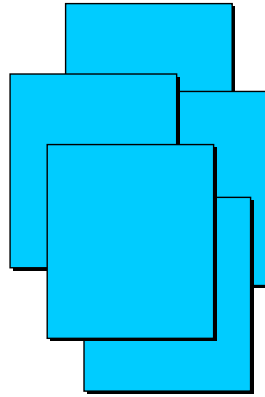
Software Development Flow

What Files Are Involved?

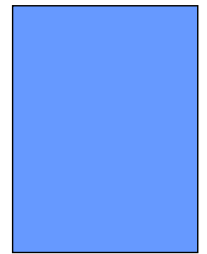
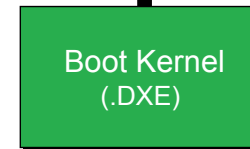
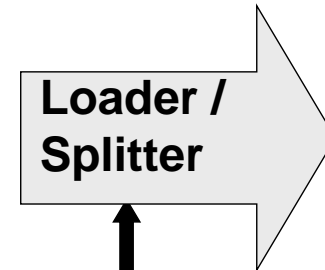
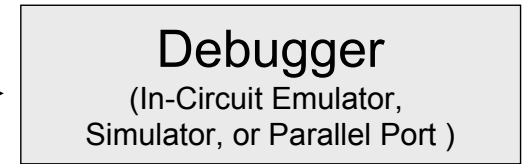
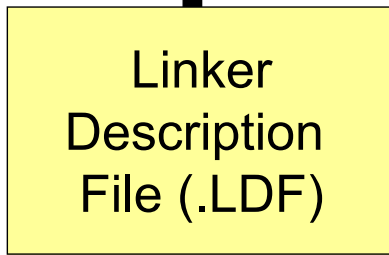
Source Files
(.C and .ASM)



Object Files
(.DOJ)



Executable
(.DXE)



Boot Image
(.LDR)

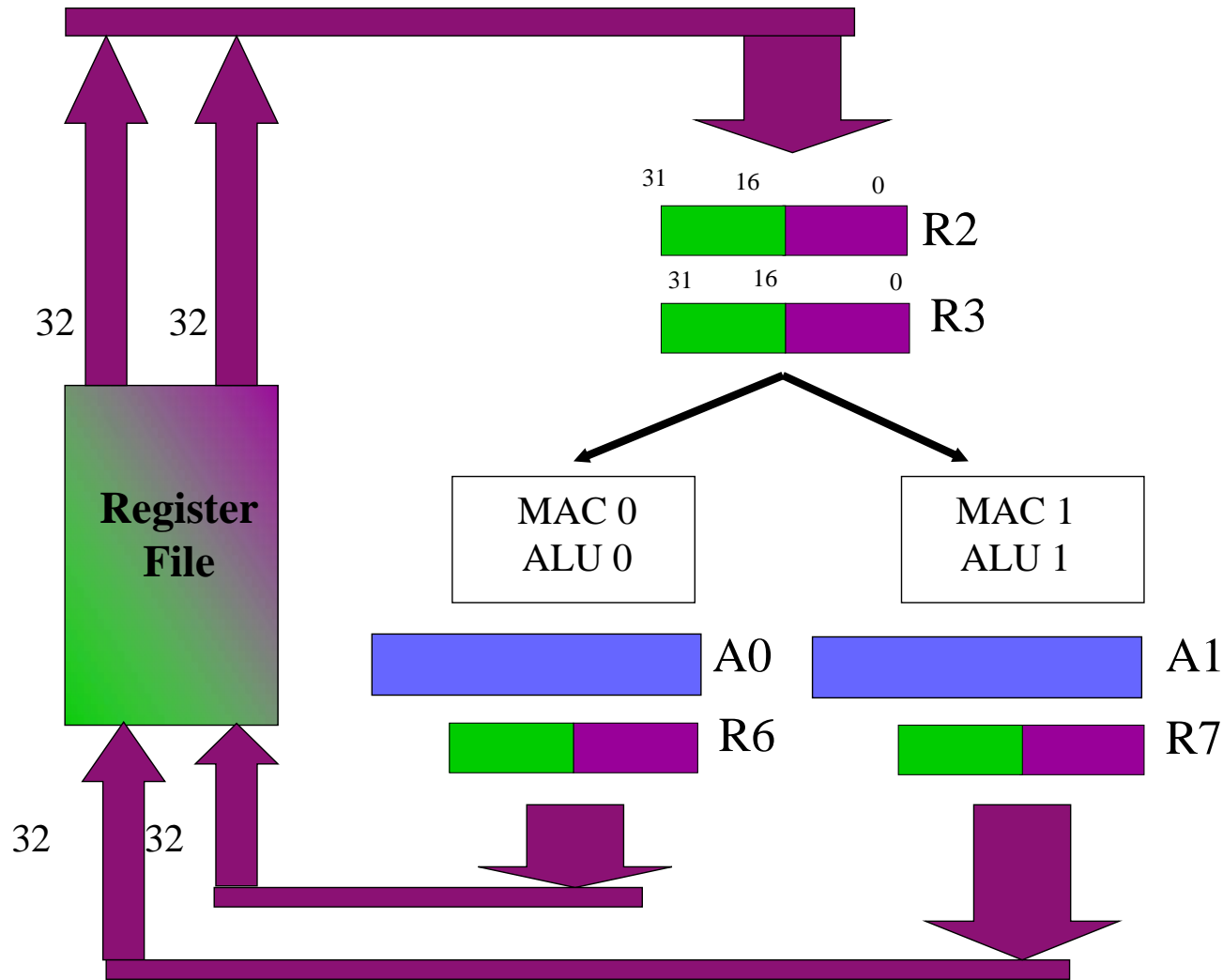
Instruction Set Overview

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

MSA Data Types

- ◆ **8-bit bytes**
 - signed or unsigned integers
- ◆ **16-bit half-words (little-endian)**
 - signed or unsigned integers
 - signed fractional (1.15)
- ◆ **32-bit words (little-endian)**
 - signed or unsigned integers
 - signed fractional (1.31)

Register View of Math



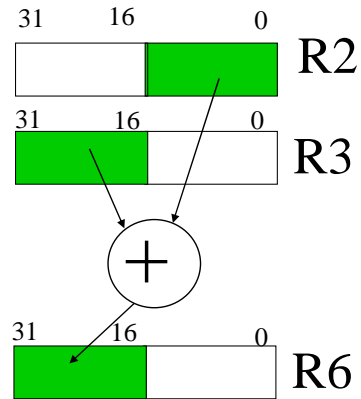
Dual ALU / MAC functions are “Vector” functions

Two Pairs of operands are available

ALU Capabilities

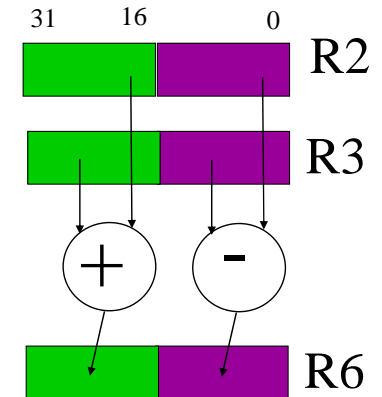
**Single
16-bit addition**

$$R6.H = R2.L + R3.H$$



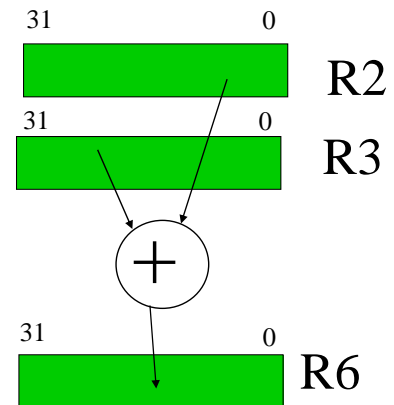
**Dual
16-bit addition**

$$R6 = R2 + | - R3$$



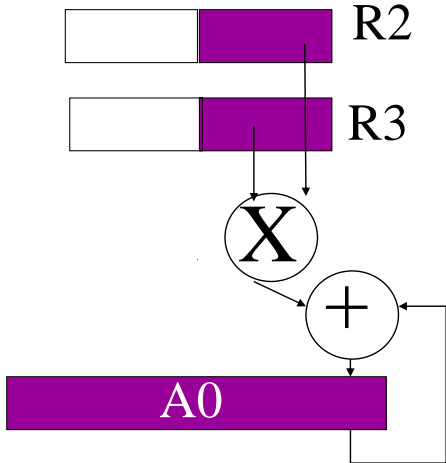
32-bit addition

$$R6 = R2 + R3$$

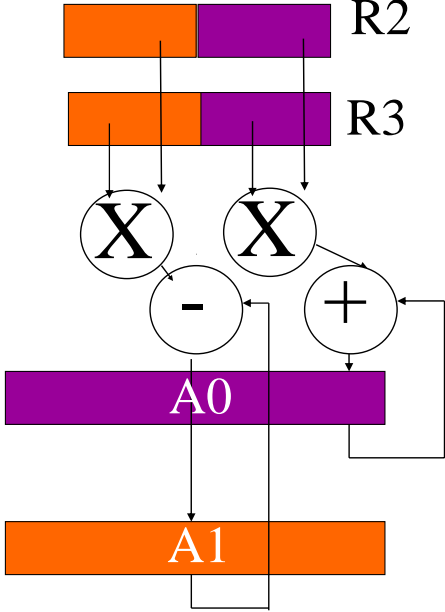
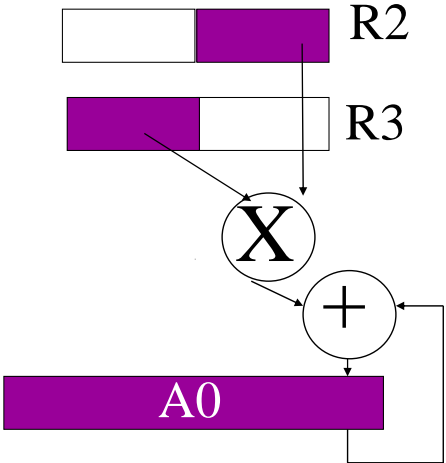


MAC Capabilities

$A0+ = R2.L * R3.L$

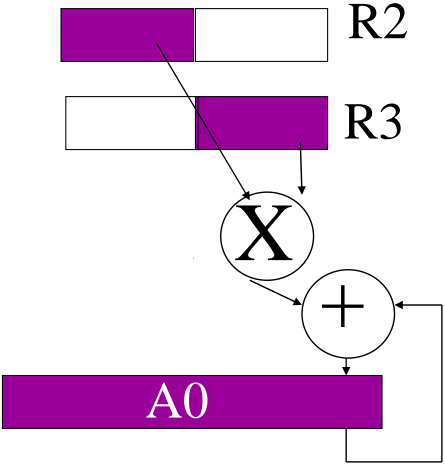


$A0+ = R2.L * R3.H$

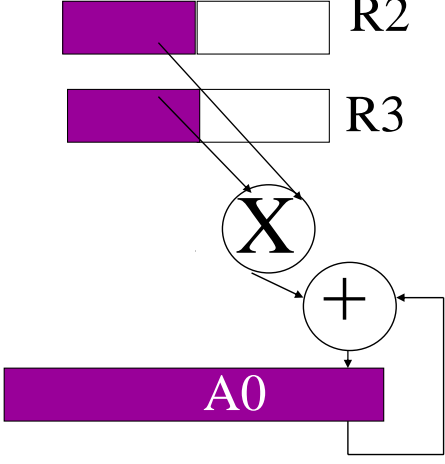


$A1- = R2.H * R3.H, A0+ = R2.L * R3.L$

$A0+ = R2.H * R3.L$



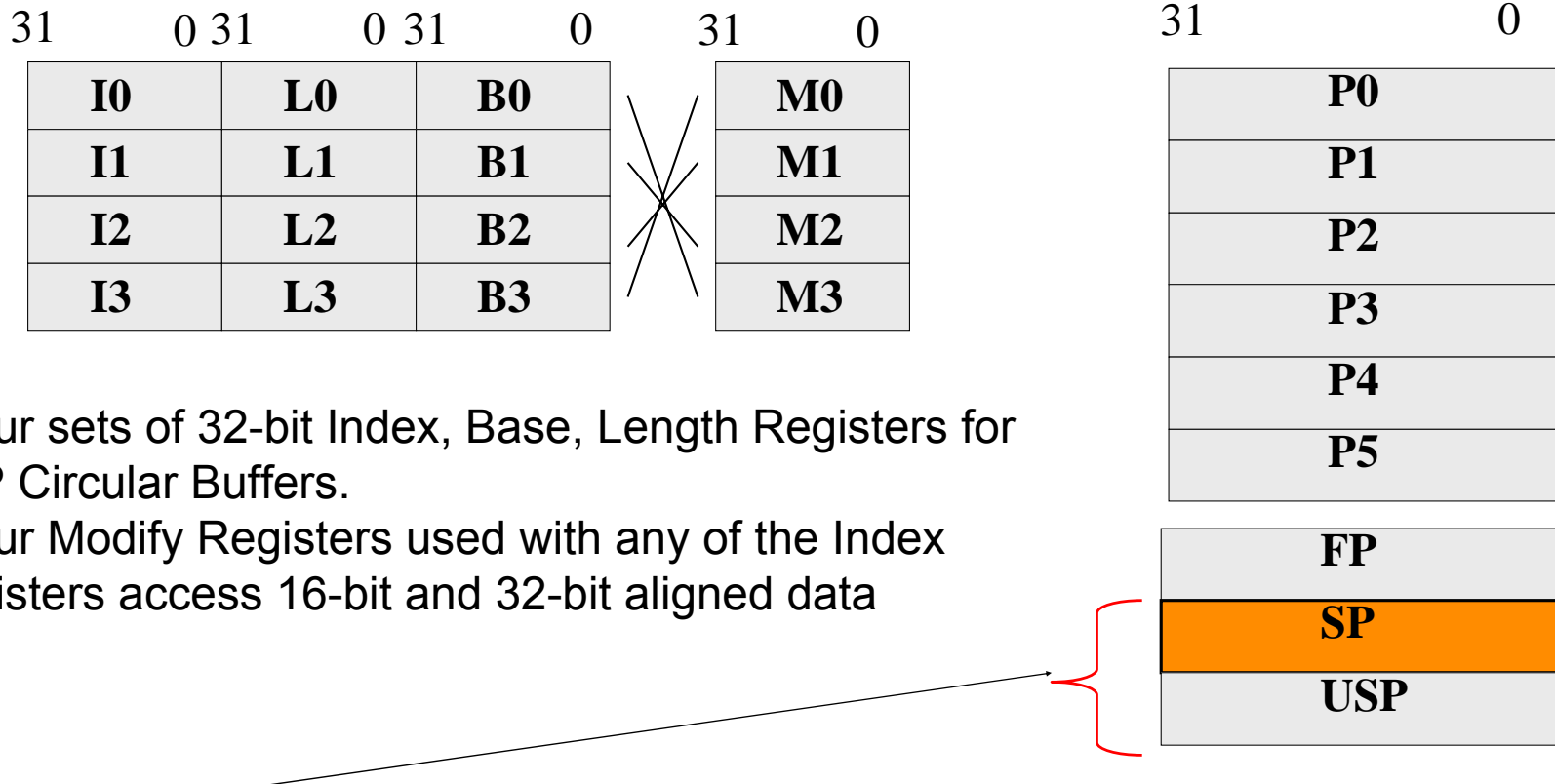
$A0+ = R2.H * R3.H$



Shifter Capabilities

- ◆ **Arithmetic shift**
- ◆ **Logical shift**
- ◆ **Add with shift**
- ◆ **Shift with add**
- ◆ **Rotate**
- ◆ **Field Extract & Deposit**

Data Address Generators



- Four sets of 32-bit Index, Base, Length Registers for DSP Circular Buffers.
- Four Modify Registers used with any of the Index Registers access 16-bit and 32-bit aligned data

- Six 32-bit Pointer Registers for general use to access 8, 16 and 32-bit data

MSA Addressing Modes

◆ DSP Addressing Modes – Index Registers

● Indirect, auto-increment, auto-decrement for loads/stores

- ◆ $R0 = [I2];$
- ◆ $R0.H = W[I0++];$ // W implies a 2-byte increment
- ◆ $[I2--] = R0;$

● Post-modify with non-unity stride for 32-bit loads/stores

- ◆ $R1 = [I2 ++ M1];$

● General Addressing Modes – Pointer Registers

● Indirect, auto-increment, auto-decrement, indexed with immediate offset for 8/16/32-bit loads/stores to

- ◆ $R3 = [P0];$
- ◆ $R7.L = W[P1++];$
- ◆ $R2 = B[P2--];$ // B implies a 1-byte decrement

● Post-modify with non-unity stride for 16/32-bit loads/stores

- ◆ $R0.H = W[P1++P2];$

Blackfin Assembly Optimization

Advanced Instructions – Vector Operations

- ◆ **SIMD feature of Blackfin Processor**
- ◆ **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)**

Instruction for Viterbi Decoding Algorithm

- ◆ **VIT_MAX and Add_on_Sign**
- ◆ **A FEC (Forward Error Correction) Technique used in wireless digital communication system, as Convolutional Coding with Viterbi Decoding**
- ◆ **Particularly suited to a channel in which the transmitted signal is corrupted mainly by additive white gaussian noise (AWGN)**
- ◆ **More detail at <http://home.netcom.com/~chip.f/viterbi/tutorial.html>**

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

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXBB

A0 if ASR

00000000

BBXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

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

$$\text{dest_hi} = \text{dest_lo} = \text{SIGN}(\text{src0_hi}) * \text{src1_hi} + \text{SIGN}(\text{src0_lo}) * \text{src1_lo}$$

src_reg_0

a1

a0

src_reg_1

b1

b0

dest

(sign_adjusted_b1) +
(sign_adjusted_b0)

(sign_adjusted_b1) +
(sign_adjusted_b0)

◆ Example

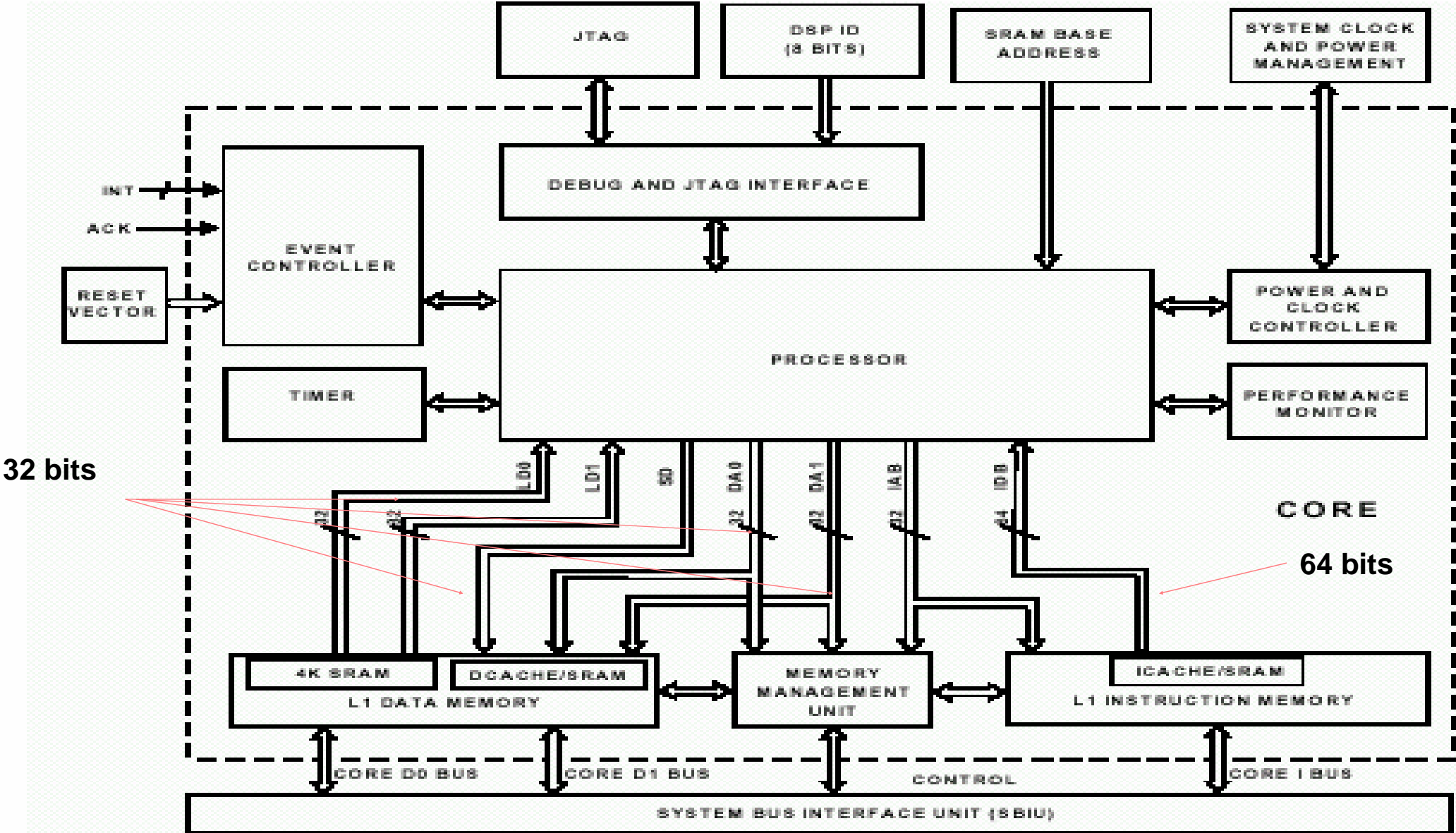
// r2.h = -2, r3.h = 23

// 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

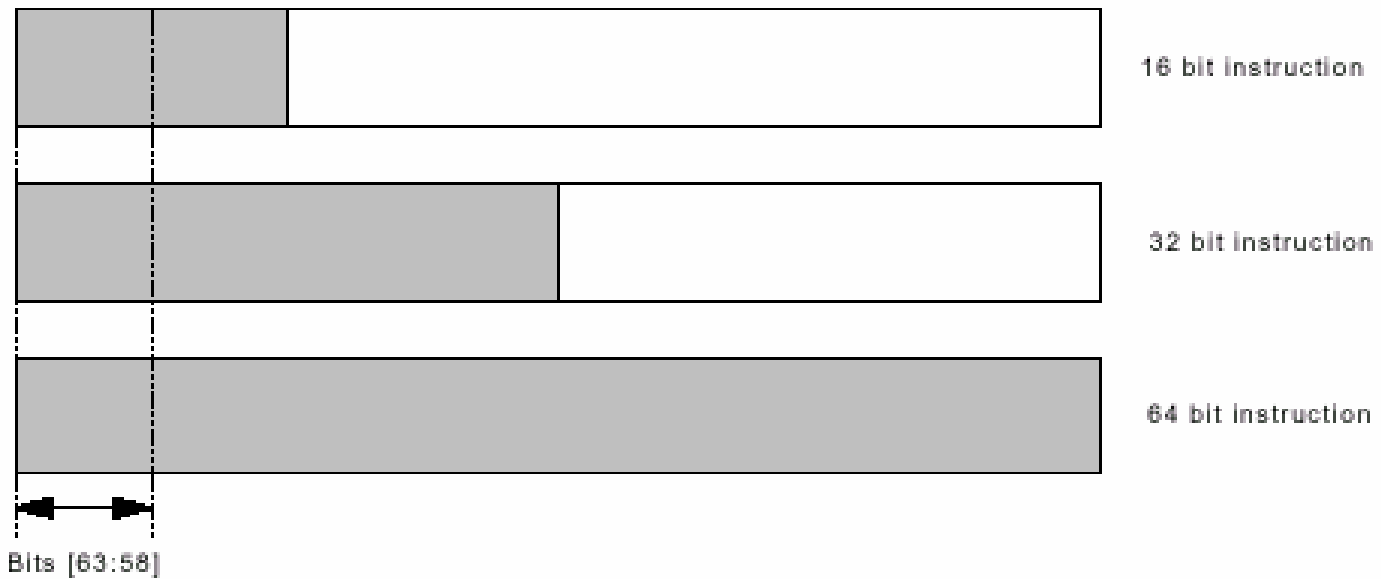
MSA Core



Instruction Fetch

- 64-bit instruction line can fetch between 1 and 4 instructions

One 64-bit instruction			
One 32-bit instruction		One 32-bit instruction	
One 16-bit instruction	One 16-bit instruction	One 16-bit instruction	One 16-bit instruction
One 32-bit instruction		One 16-bit instruction	One 16-bit instruction



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 lreg 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 lreg 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

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

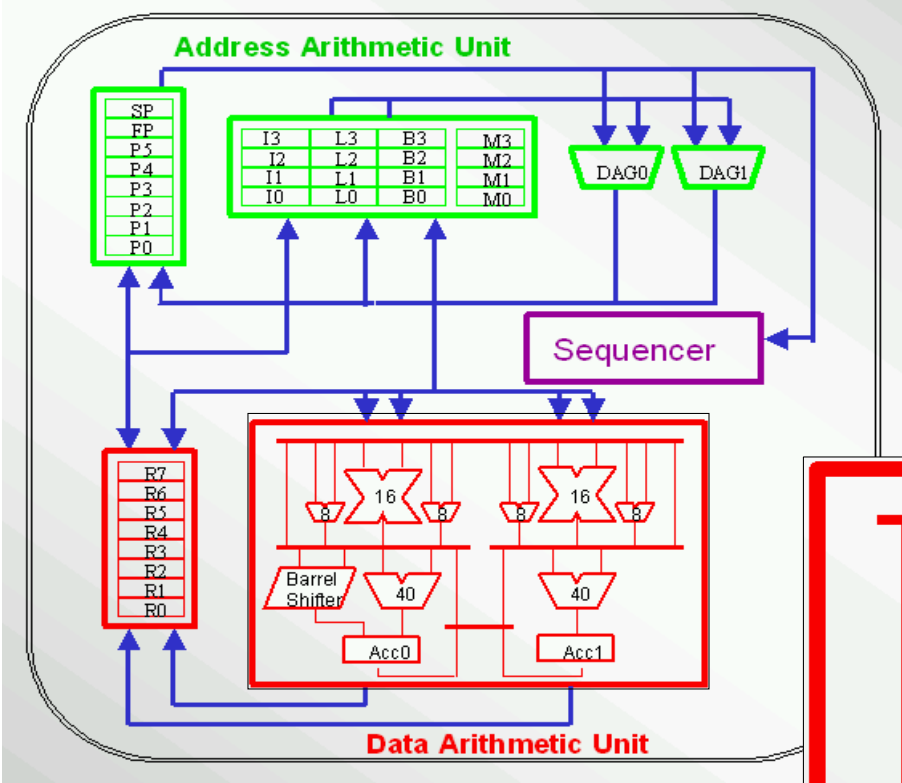
- ◆ The following example shows how to compute a division using a signed integer and divisor

```
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). */
```

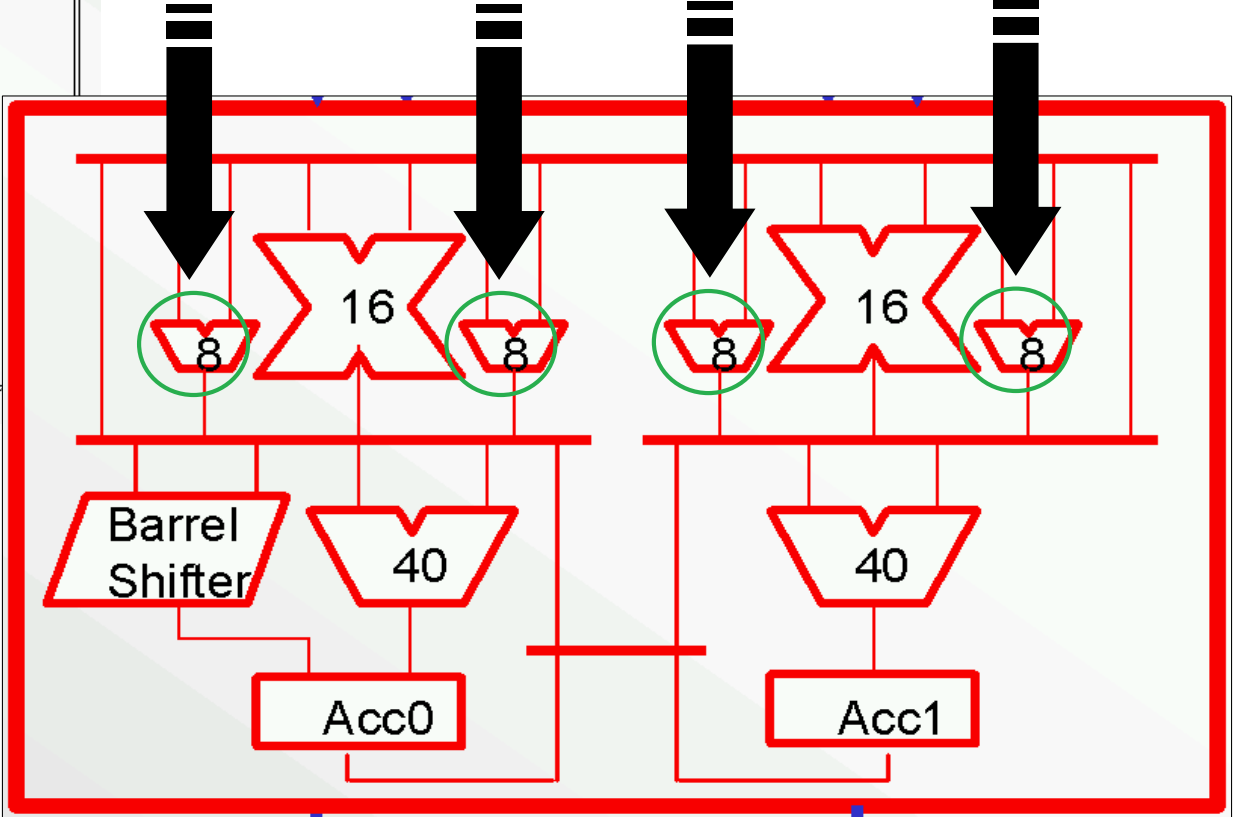
- ◆ **Hands-on for Division:** C cycle _____ ASM Cycle _____

8-Bit ALU Instructions – Video Pixel Operations

8-Bit Video ALUs



Four Video ALUs

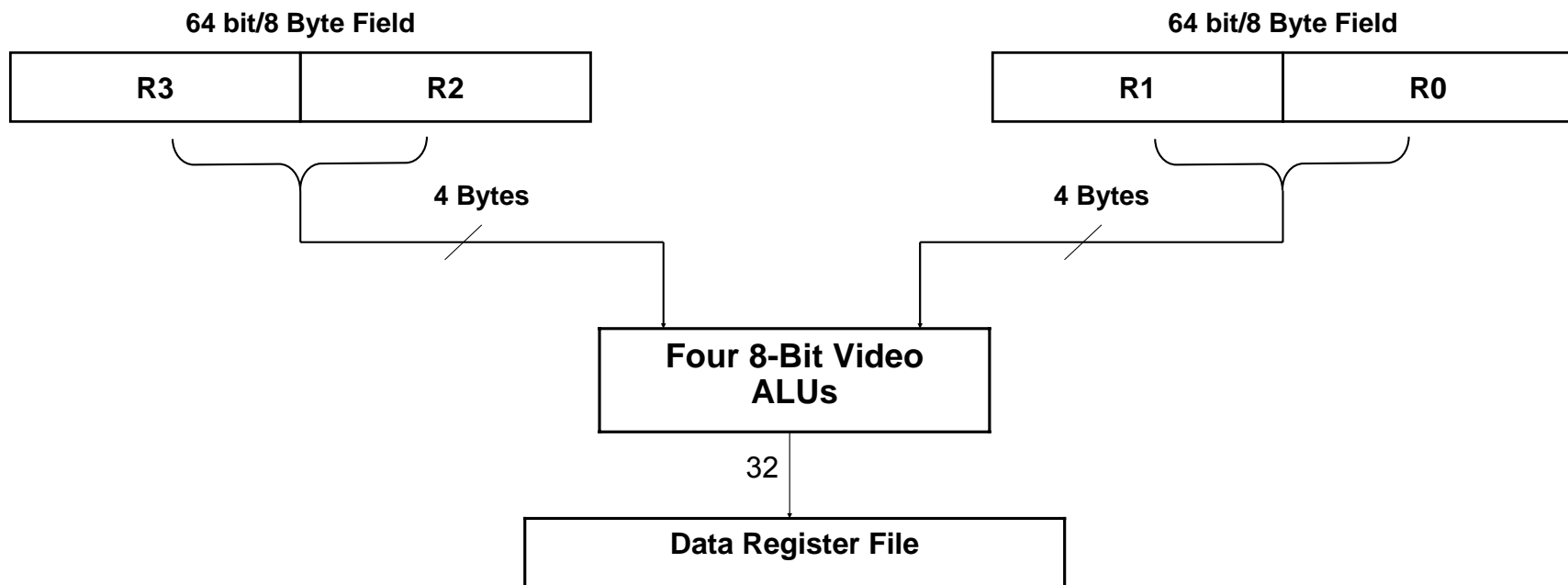


Video Pixel Operations

- ◆ **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 can be used to**
 - **align bytes**
 - **disable exceptions that result from misaligned 32-bit memory accesses**
 - **perform dual and quad 8-bit and 16-bit add, subtract and averaging operations**

8-Bit ALU Operations

- ◆ 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



Instruction Summary

◆ **Align related:**

- **ALIGN8, ALIGN16, ALIGN24**
- **DISALIGNEXCPT**

◆ **Addition related:**

- **BYTEOP16P – Quad 8-Bit Add**
- **BYTEOP3P – Dual 16-Bit Add/Clip**

◆ **Subtraction related:**

- **BYTEOP16M – Quad 8-Bit Subtract**
- **SAA – Quad 8-Bit Subtract-Absolute-Accumulate**

◆ **Averaging related:**

- **BYTEOP1P – Quad 8-Bit Average in Byte**
- **BYTEOP2P – Quad 8-Bit Average in Half Word**

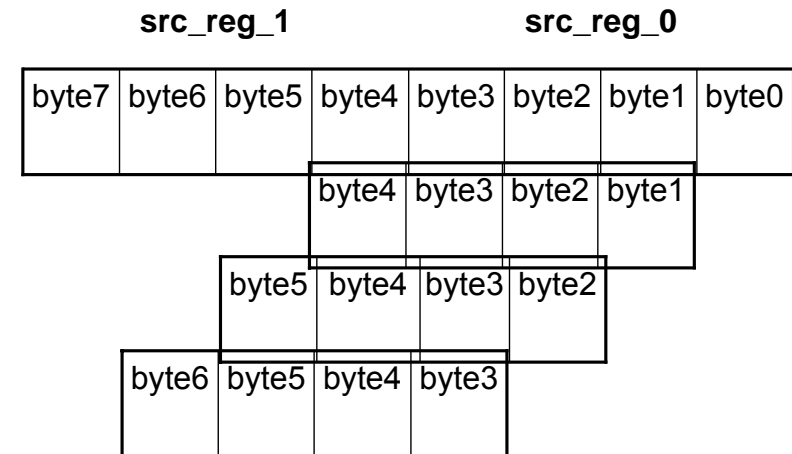
◆ **Pack and unpack:**

- **BYTEPACK – Quad 8-Bit Pack**
- **BYTEUNPACK – Quad 8-Bit Unpack**

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 */
```

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 Operation

◆ BYTEOP16P (Quad 8-bit Add)

- Adds eight unsigned bytes to result in four 16-bit words

◆ General Form

- $(\text{dest_reg_1}, \text{dest_reg_0}) = \text{BYTEOP16P}(\text{src_reg_0}, \text{src_reg_1}) [(R)]$
- 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

- $(r1, r2) = \text{BYTEOP16P}(r3:2, r1:0);$

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	

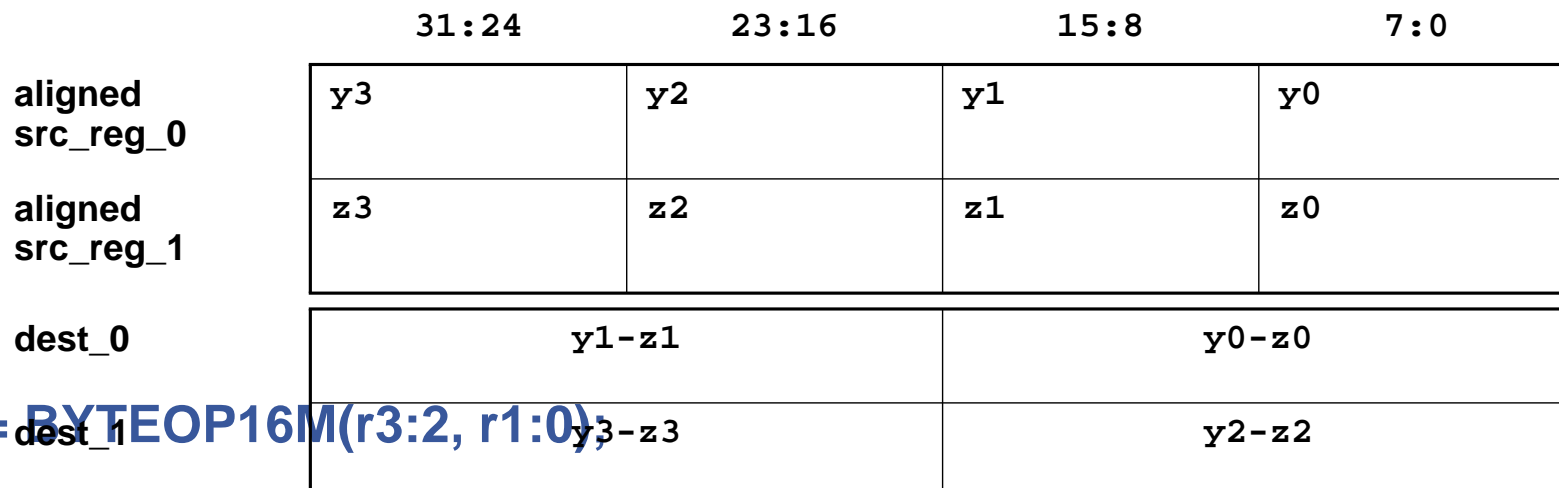
Subtraction Operation

◆ BYTEOP16M (Quad 8-bit Subtract)

- Subtracts eight unsigned bytes to result in four sign-extended 16-bit words

◆ General Form

- $(\text{dest_reg_1}, \text{dest_reg_0}) = \text{BYTEOP16M}(\text{src_reg_0}, \text{src_reg_1}) [(R)]$
- source data chosen by I0 and I1 from register pairs R3:2 and R1:0



◆ Example

- $(r1, r2) = \text{BYTEOP16M}(r3:2, r1:0);$

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	

Quad-Byte-Sum Absolute Difference

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

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

SAA Operation

◆ 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	<code>a(i,j+3)</code>	<code>a(i,j+2)</code>	<code>a(i,j+1)</code>	<code>a(i,j)</code>
aligned src_reg_1	<code>b(i,j+3)</code>	<code>b(i,j+2)</code>	<code>b(i,j+1)</code>	<code>b(i,j)</code>
A0 (H:L)	<code>+= a(i,j+1)-b(i,j+1) </code>		<code>+= a(i,j)-b(i,j) </code>	
A1 (H:L)	<code>+= a(i,j+3)-b(i,j+3) </code>		<code>+= a(i,j+2)-b(i,j+2) </code>	

◆ 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

$\text{dest_reg_1} = \text{a1.l} + \text{a1.h}, \text{dest_reg_0} = \text{a0.l} + \text{a0.h}$

◆ Example

$\text{r4} = \text{a1.l} + \text{a1.h}, \text{r7} = \text{a0.l} + \text{a0.h};$

Hands-on for SAA

◆ Using SAA to implement the following subroutine:

Let's assume :

I0 contains the start address of target block (TAR_BLK)

I1 contains the start address of reference block (REF_BLK)

The length of both block is 64

Please use SAA to implement SAD algorithm

Reference code for SAA

```
A1=A0=0 || R0 = [I0++] || R2 = [I1++];  
LSETUP (MAD_START, MAD_END) LC0=P5;  
MAD_START:  
SAA (R1:0,R3:2) || R1 = [I0++] || R3 = [I1++];  
SAA (R1:0,R3:2) (R) || R0 = [I0++] || R2 = [I1++];  
SAA (R1:0,R3:2) || R1 = [I0 ++] || R3 = [I1++];  
MAD_END: SAA (R1:0,R3:2) (R) || R0 = [I0++] || R2 = [I1++];  
        R3=A1.L+A1.H,R2=A0.L+A0.H;  
        R0 = R2 + R3 (NS);
```

Averaging Operation – Byte

- ◆ **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**

◆ Example

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

	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)

BYTEOP1P 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

Hands-on for BYTEOP1P

◆ Assembly Code:

P5= 16;

LSETUP(LOOP_ST,LP_END) LC0=P5;

DISALGNEXCPT || R0 = [I0++] || R2 = [I1++];

LOOP_ST: DISALGNEXCPT || R1 = [I0++] || R3 = [I1++];

R6 = BYTEOP1P(R1:0,R3:2) || R0 = [I0++] || R2 = [I1++];

R7 = BYTEOP1P(R1:0,R3:2)(R) || R1 = [I0++] || [I3++] = R6 ;

DISALGNEXCPT || R3 = [I1++] || [I3++] = R7;

R6 = BYTEOP1P(R1:0,R3:2) || R0 = [I0++M1] || R2 = [I1++M1];

R7 = BYTEOP1P(R1:0,R3:2)(R) || R0 = [I0++] || [I3++] = R6 ;

LP_END: DISALGNEXCPT || R2 = [I1++] || [I3++M2] = R7;

Averaging Operation – Half-Word

◆ 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!

BYTEOP2P 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

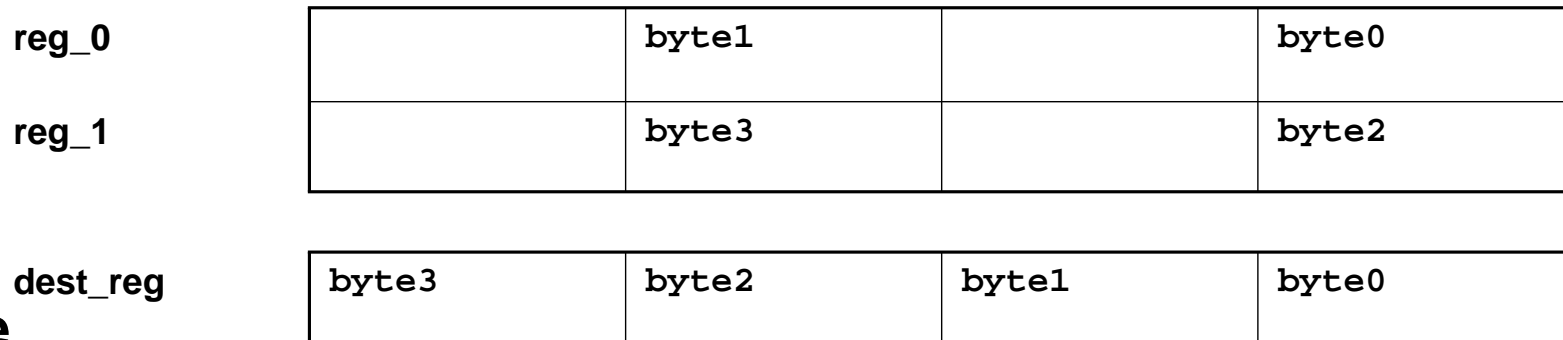
Quad-Byte Pack

◆ BYTEPACK (Quad 8-bit Pack)

- Prepares data for 8-bit ALU operations

◆ General Form

`dest_reg = BYTEPACK(src_reg_0, src_reg_1)`



◆ Example

`/* r3 = 0x0034 0012, r4 = 0x0078 0056 */`

`r2 = BYTEPACK(r3, r4);`

`/* r2 = 0x7856 3412 */`

Quad-Byte Unpack

◆ BYTEUNPACK (Quad 8-bit Unpack)

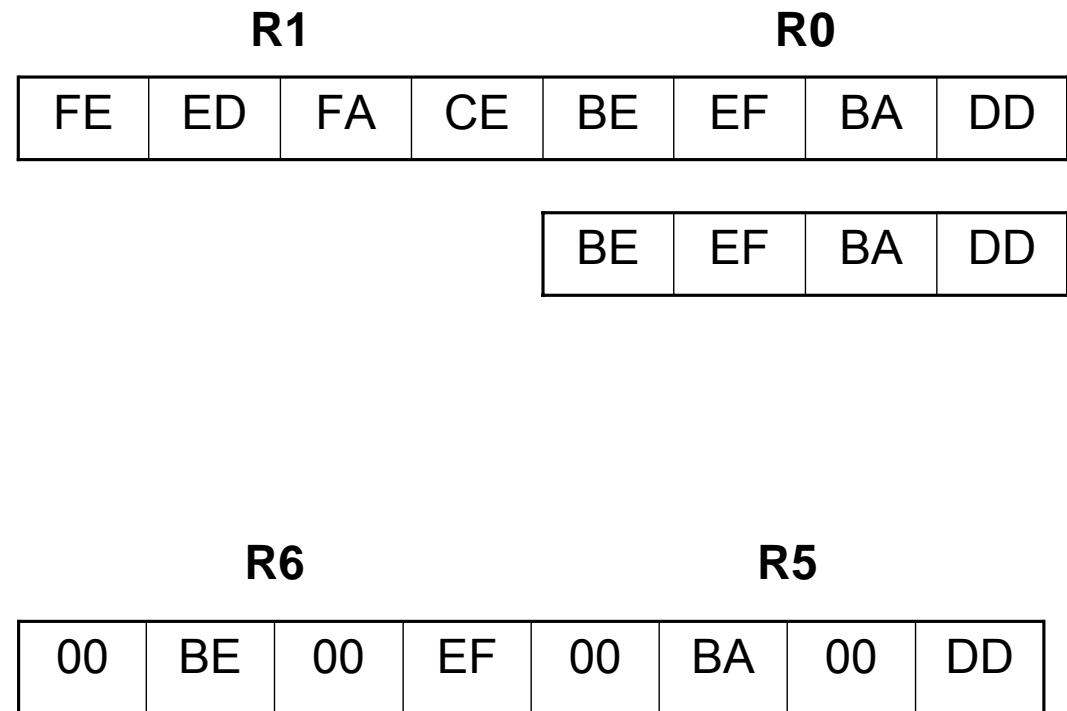
- Inverse of BYTEPACK, but includes an additional alignment mechanism

● General Form

- ◆ $(\text{dest_reg_1}, \text{dest_reg_2}) = \text{BYTEUNPACK src_reg_pair}$
- ◆ source data chosen by I0 from register pairs R3:2 and R1:0

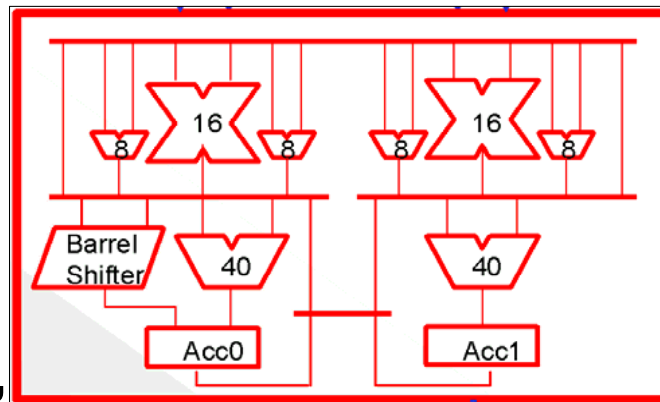
● Example

- ◆ $(r6, r5) = \text{BYTEUNPACK r1:0};$

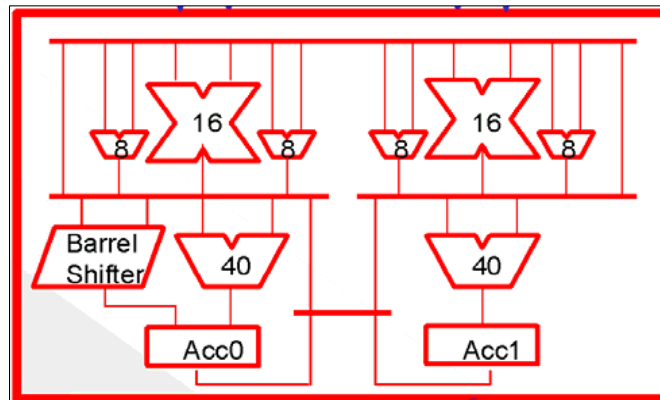


40-Bit vs 8-Bit ALUs

- ◆ 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



- ◆ $(r1, r2) = \text{BYTEOP16P}(r3:2, \dots)$ to perform 4 add operations



Hands On

- ◆ **Image enhanced algorithm (Image _ processing sub-program)**
- ◆ **Optimize assembly program of Image enhanced algorithm step by step .**
- ◆ **Read cycle count of each step and compare optimized performance.**

Using Assembly language – Software loop

Main loop code:

```
R7 = N ;
```

```
begin_loop :
```

```
    R0.L = W[I0++] ;
```

```
    R1.L = W[I1++] ;
```

```
    R3 = R0.L * R1.L (IS) ;
```

```
    R3 = R2 + R3 ;
```

```
    W[I2++] = R3.L ;
```

```
    R7 += -1 ;
```

```
    CC = AZ ;
```

```
    if !CC JUMP begin_loop ;
```

Using Assembly language – hardware loop

Main loop code:

P1 = N ;

LSETUP (begin_loop, end_loop) LC0=P1 ;

begin_loop:

R0.L = W[I0++] ;

R1.L = W[I1++] ;

R3 = R1.L * R2.L (IS) ;

R3 = R2 + R3 ;

end_loop: W[I2++] = R3.L ;

Using Assembly language – 32 bit register/ operation

Main loop code:

P1 = N>>1 ;

LSETUP (begin_loop, end_loop) LC0=P1;

begin_loop :

R0 =[I0++];

R1 =[I1++];

R3.H = R0.H * R1.H, R3.L = R0.L*R1.L (IS);

R3 = R2 +|+ R3 ;

end_loop: [I2++] = R3;

Using Assembly language – multi-instructions

P1 = N>>1 ;

LSETUP (begin_loop, end_loop) LC0=P1;

begin_loop :

mnop ||R0 =[I0++] || R1 =[I1++] ;

R3.H = R0.H * R1.H, R3.L = R0.L*R1.L (IS) ;

R3 = R2 +|+ R3 ;

end_loop: [I2++] = R3;

Using Assembly language – Loop analysis

Loop code:

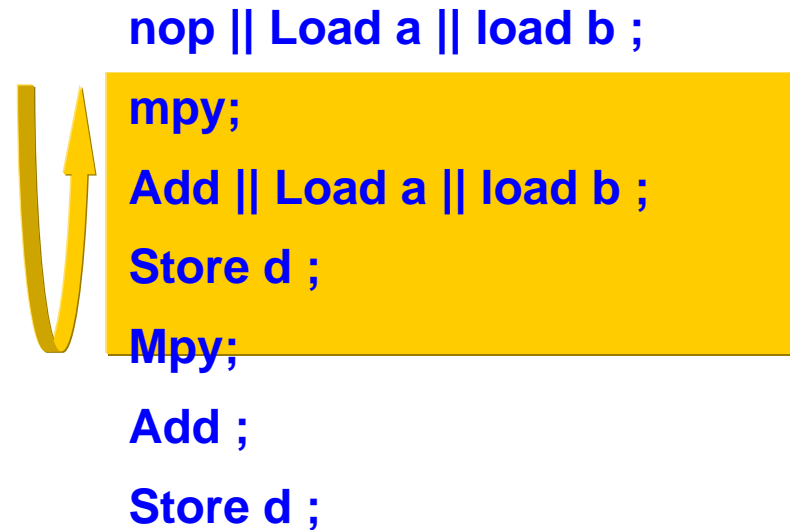
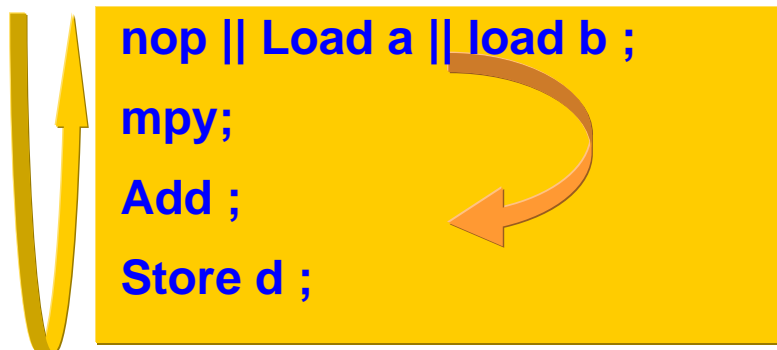
begin_loop :

mnop ||R0 =[I0++] || R1 =[I1++] ;

R3.H = R0.H * R1.H, R3.L = R0.L*R1.L (IS) ;

R3 = R2 +|+ R3 ;

end_loop: [I3++] = R3;



Using Assembly language – Simple software pipeline

Main loop code:

```
mnop ||R0 =[I0++] || R1 =[I1++] ;
```

```
P1 = (N>>1)-1 ;
```

```
LSETUP (begin_loop, end_loop) LC0=P1;
```

```
begin_loop :
```

```
    R3.H = R0.H * R1.H, R3.L = R0.L*R1.L (IS) ;
```

```
    R3 = R2 +|+ R3 ||R0 =[I0++] || R1 =[I1++] ;
```

```
end_loop: [I2++] = R3;
```

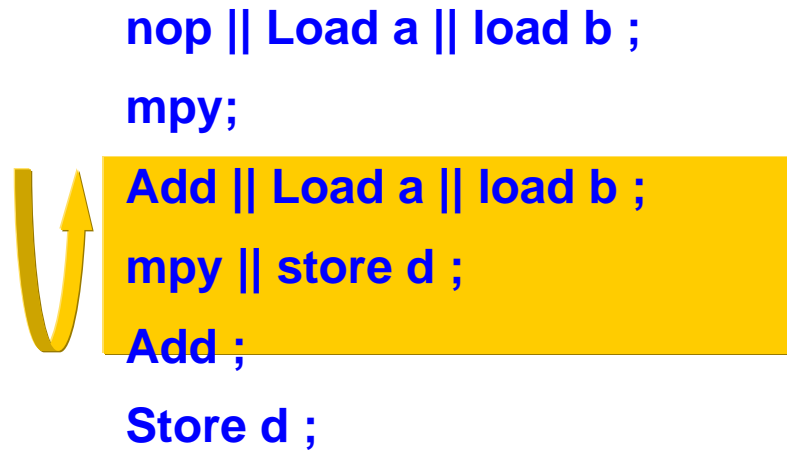
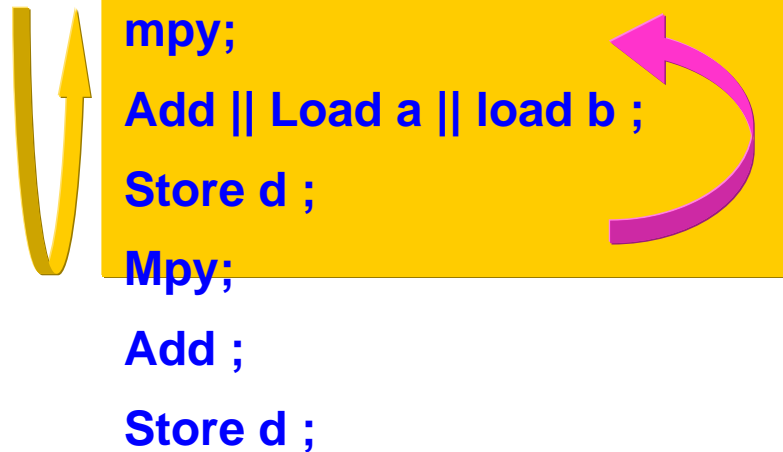
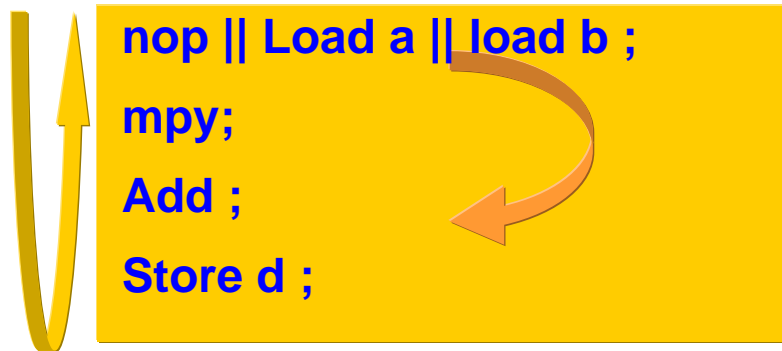
```
    R3.H = R0.H * R1.H, R3.L = R0.L*R1.L (IS) ;
```

```
    R3 = R2 +|+ R3;
```

```
    [I2++] = R3;
```

Using Assembly language - Loop analysis co.

Loop code:



Using Assembly language – Advanced software pipeline

Main loop code:

```
mnop ||R0 =[I0++] || R1 =[I1++] ;
```

```
R3.H = R0.H * R1.H, R3.L = R0.L*R1.L (IS) ;
```

```
P1 = (N>>1)-1 ;
```

```
LSETUP (begin_loop, end_loop) LC0=P1;
```

begin_loop :

```
    R3 = R2 +|+ R3 || R0 =[I0++] || R1 =[I1++] ;
```

```
end_loop: R3.H = R0.H * R1.H, R3.L = R0.L*R1.L (IS) || [I2++] = R3 ;
```

```
    R3 = R2 +|+ R3;
```

```
    [I2++] = R3;
```

Q&A