

Power Architecture™  
DEVELOPER CONFERENCE '07



## MPC5500 Signal Processing Engine

9.25.2007

Rebeca Delgado, Freescale Semiconductor, Inc.

Power.ORG ™

# Presentation Outline



- » SPE overview
- » 200+ SIMD Instructions
- » Embedded Floating Point APU
- » Programming Interface Model (PIM)
- » Code Examples

# MPC5554



## Core

- **40-132 MHz Power Architecture™ ISA e200z6 Core**
  - Integer binary user mode compatible with RCP (MPC500)
  - New SIMD module for DSP and floating point features

## Memory

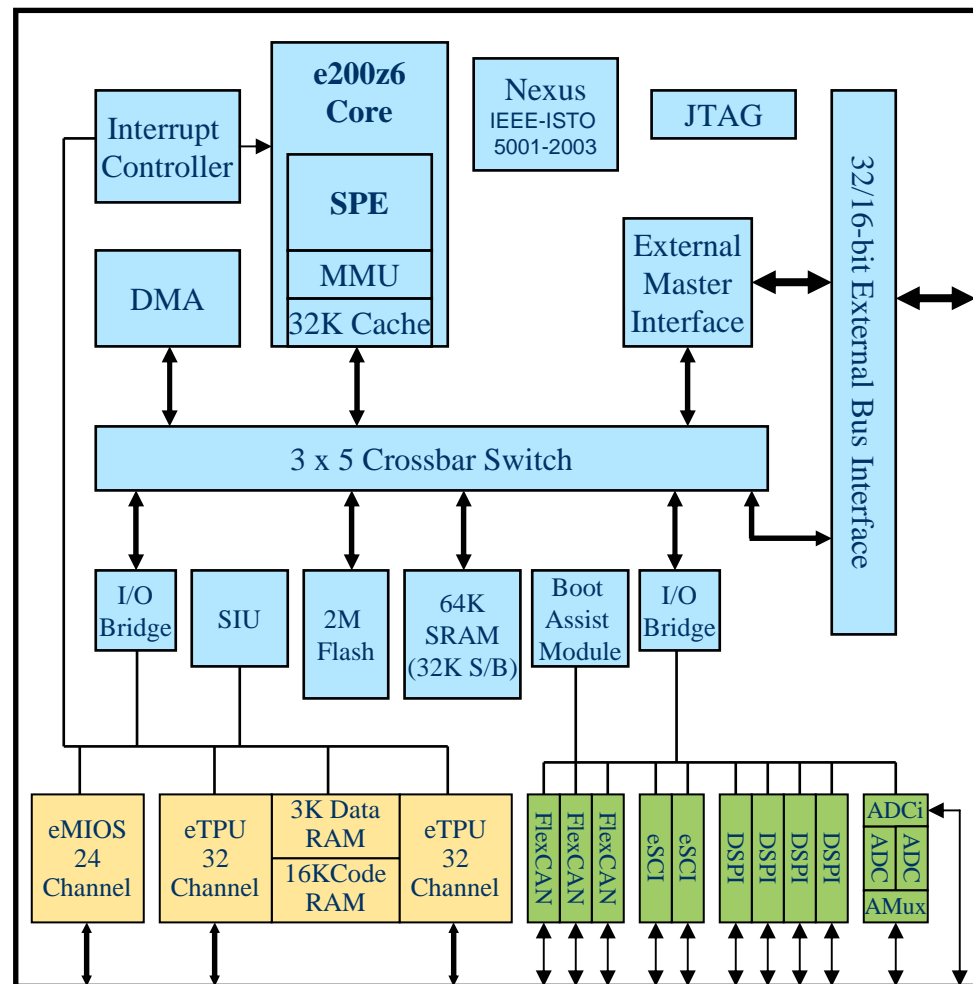
- **2 MB RWW Flash with ECC**
- **115K Total SRAM (including cache and eTPU memory)**
  - 64K Data RAM (including 32K with standby) with ECC
  - 32K unified-cache (with line locking)
  - 19K for eTPU (16k code & 3k parameters)

## I/O

- **88 Timed I/O Channels**
  - 2 x 32 channel eTPU
  - 24 channel eMIOS with unified channels
- **3 x FlexCAN**—compatible with TouCAN, 64 buffers each
- **2 x eSCI**
- **4 x DSPI** 16 bits wide up to 6 chip selects each
  - Standard SPI with continuous mode and DMA support
  - Pin serialization (similar to PPM)
- **40-Channel Dual ADC**—up to 12 bit and up to 1.25µs conversions
- 6 queues with triggering and DMA support

## System

- **FM-PLL**
- **64 Channel DMA Controller**
- **308 Source Interrupt Controller**
- **Nexus IEEE-ISTO 5001-2003 Class 3+**
- **MPC500 Compatible External Bus Interface** supporting 'classic' and burst external flash
- **5/3.3V IO, 5V ADC, 3.3V/1.8V bus, 1.5V core** (from internal regulator controller)
- **416 PBGA** package



# Signal Processing Engine (SPE)

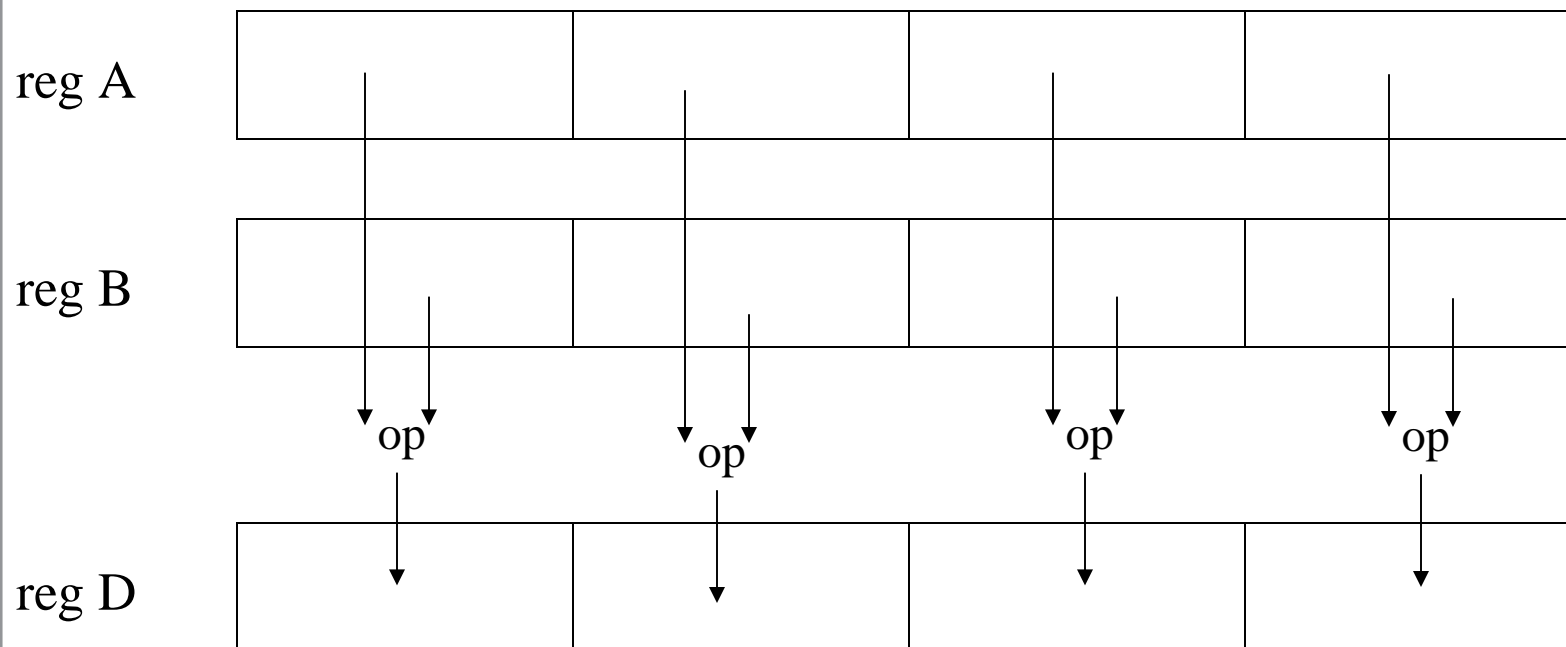


- » SPE Auxiliary Processing Unit (APU) accelerates signal processing routines such as FIR, FFT and Kalman Filters
- » SPE accelerates data movement, e.g., copying bytes
- » Designed to reduce or eliminate need for low end DSP chips in embedded applications such as automotive engine control

# SIMD Review



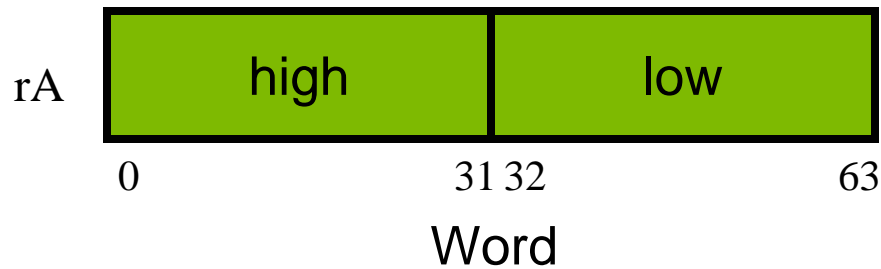
- » Multiple data elements stored in single register
- » Same operation performed on each set of data



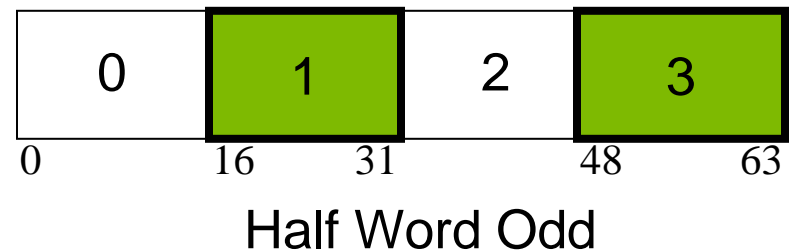
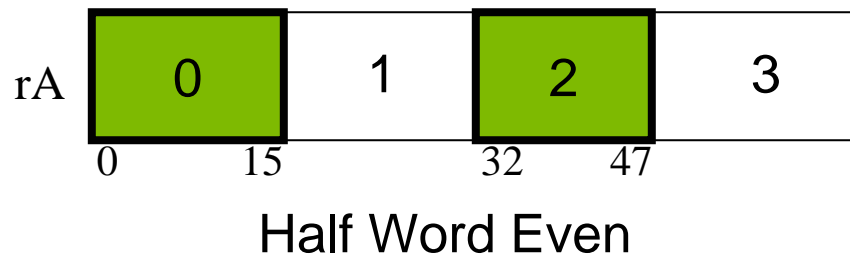
# SPE SIMD



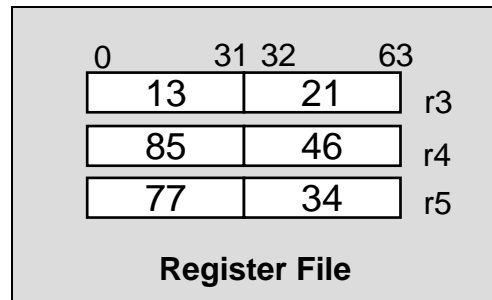
- » 16-bit or 32-bit integer or fractional data
- » Single precision floating point data



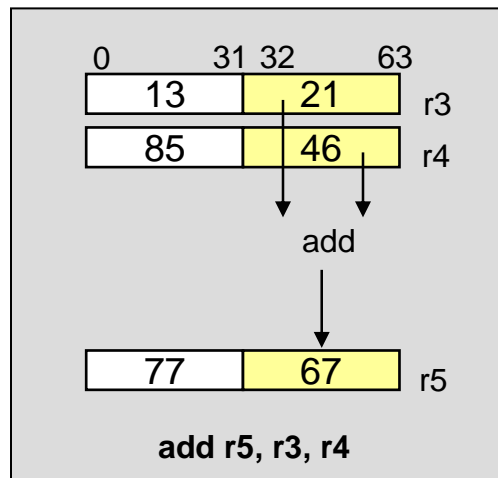
2-way  
Parallelism  
Only!



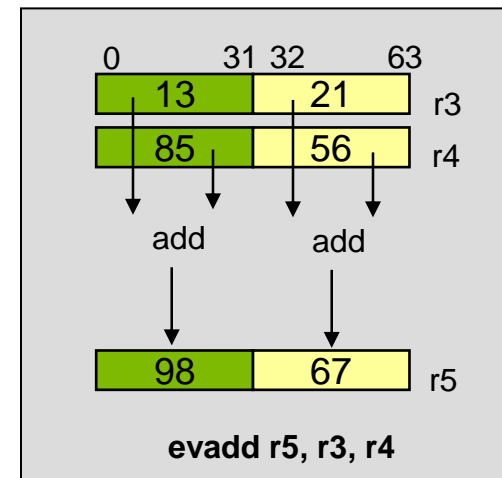
# Merged Register File



**Classic PowerPC®:** add r5, r3, r4



**SPE:** evadd r5, r3, r4



# Merged Register File



- » e200 core with SIMD
  - 32 64-bit registers
- » Advantages of merged registers
  - Smaller die size
  - Context switch time is significantly reduced
  - No expensive data movement between register banks

# SPE 64-bit Accumulator

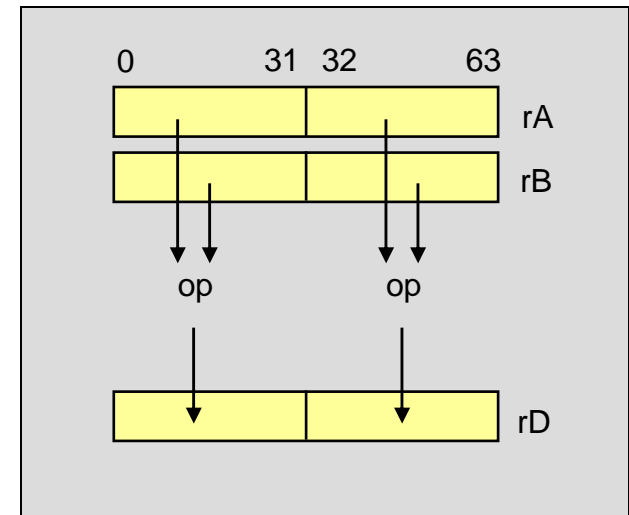


- » Holds two 32-bit values or one 64-bit value
- » Supports back to back or single cycle completion of MAC operations
  - Half word integer and fractional
  - Word low integer
  - Word integer and fractional

# Simple Operations



- » Arithmetic: add, add immediate, subtract, subtract immediate, absolute value, negate, sign extend, count leading sign, count leading zeroes, splat immediate, round
- » Logical: and, and complement, or, or complement, nand, nor, xor, xnor
- » Compare: greater than, less than, equal, select
- » Shifts: shift left, shift right, rotate
- » bit reversed increment



Power Architecture™ '07  
DEVELOPER CONFERENCE



# SIMD Instructions

Power.ORG ™

# Load/Store Instructions



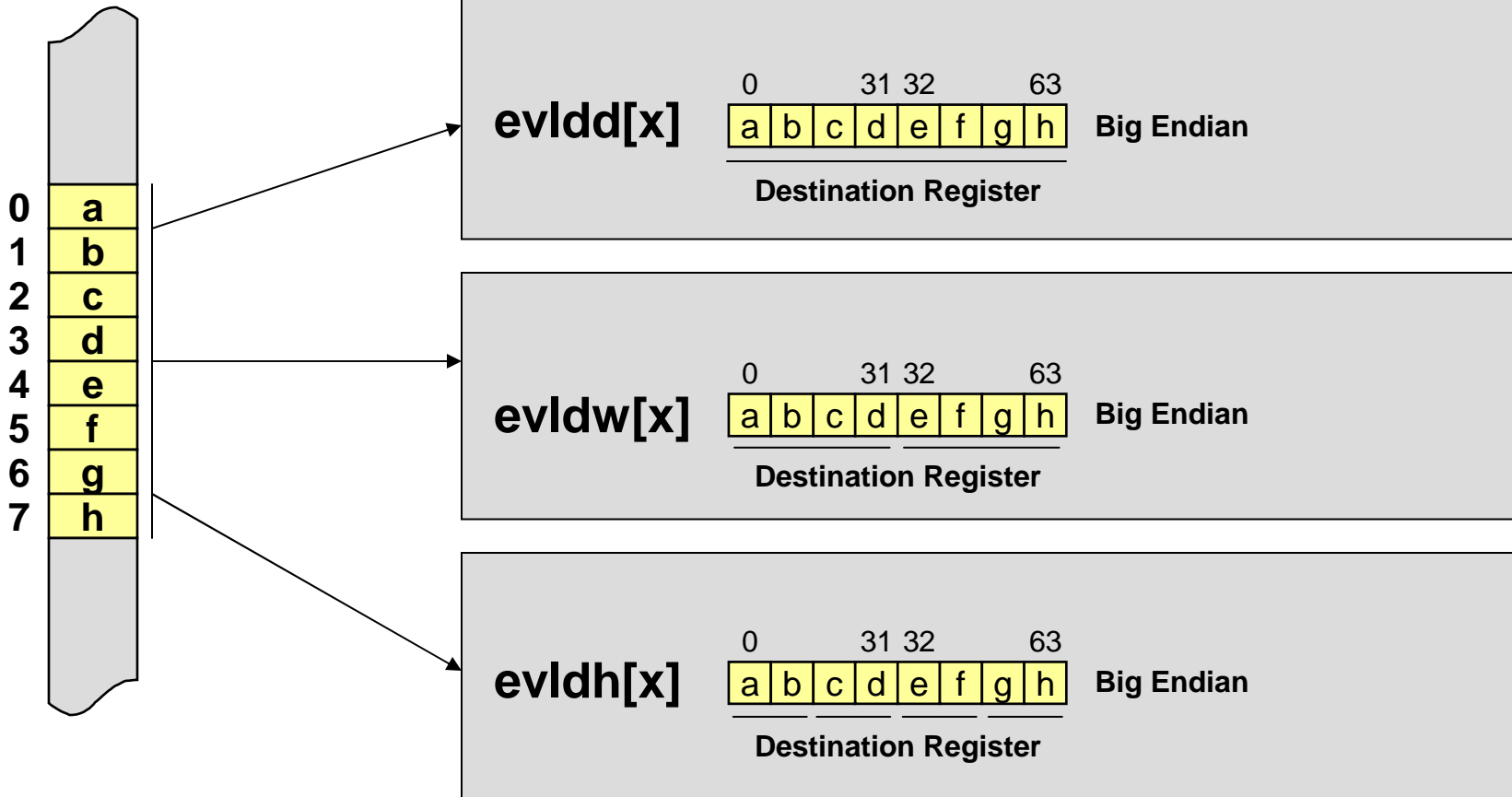
- » 22 load and 14 store instructions
  - Double word, word, and half word
- » Two forms of load/store instructions
  - Indexed and immediate
- » **Speed up data movement by 2x**
- » Data must be correctly aligned
  - Exception if data is not correctly aligned
  - SPE permits misaligned data
- » True big and little endian

# Load Double Word

as 1 double word, 2 words, or 4 half words



## Memory

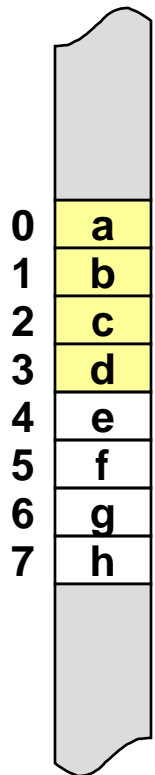


Double word aligned!!

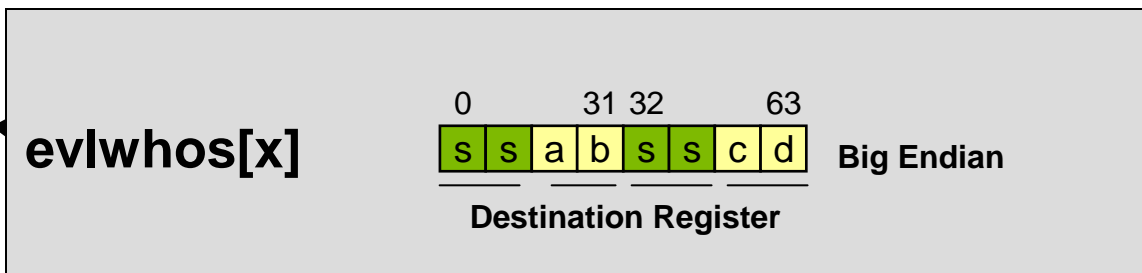
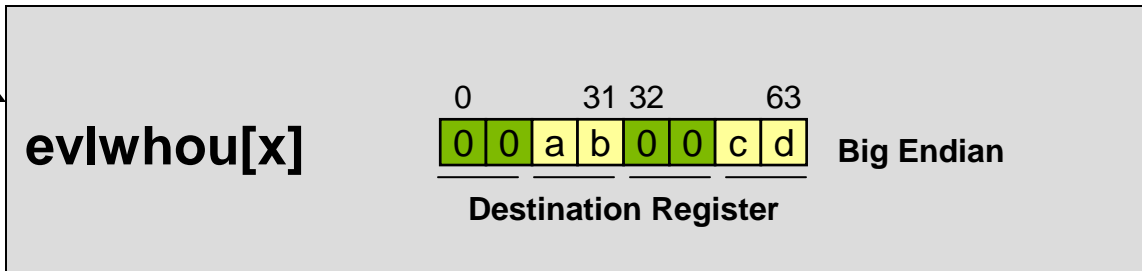
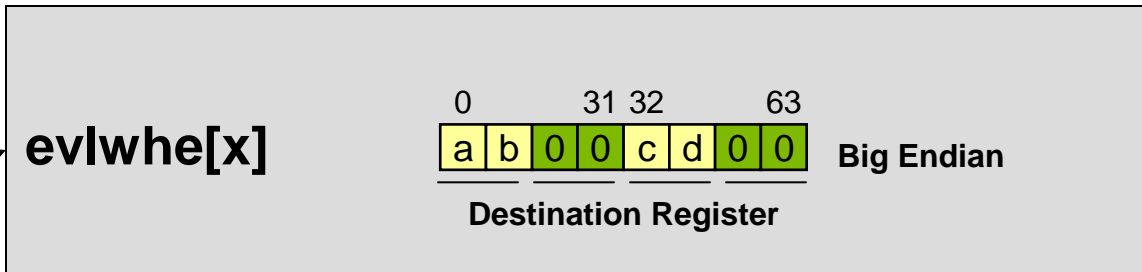
# Load Word as Half Words



## Memory



Word aligned!!

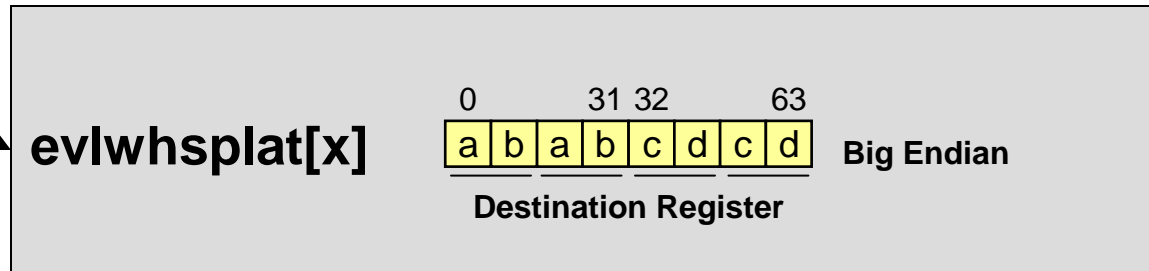
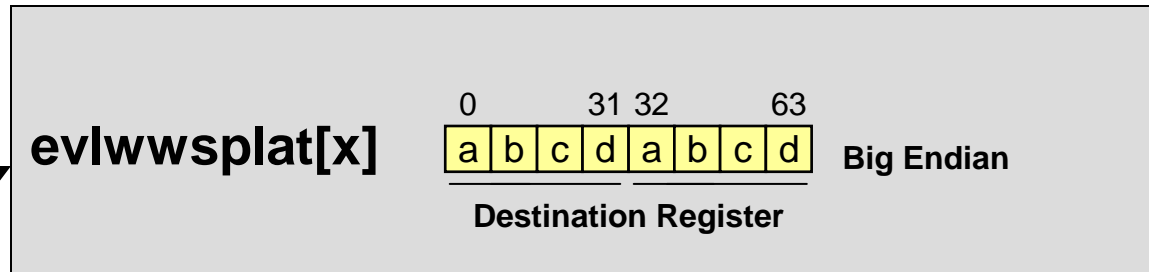
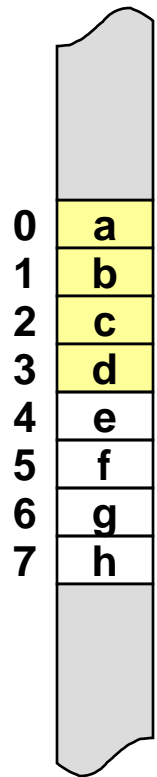


# Load Word Splat

as Word or Half Word



## Memory

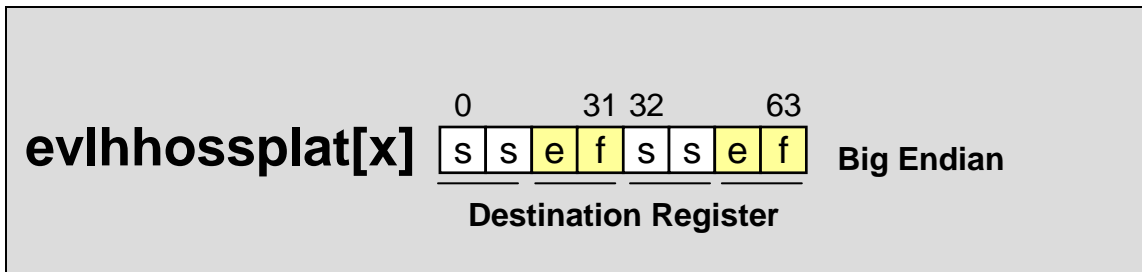
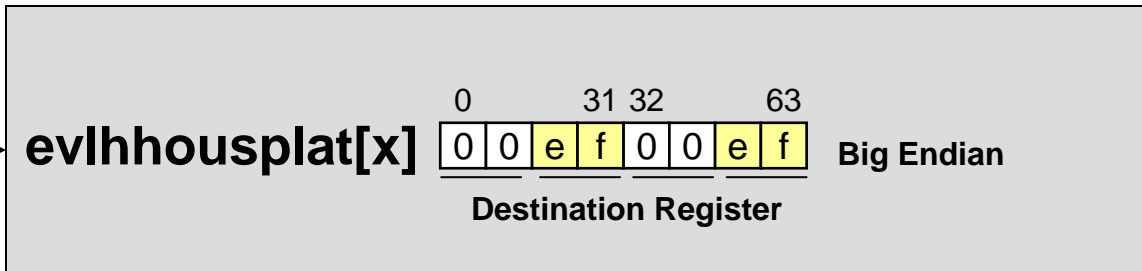
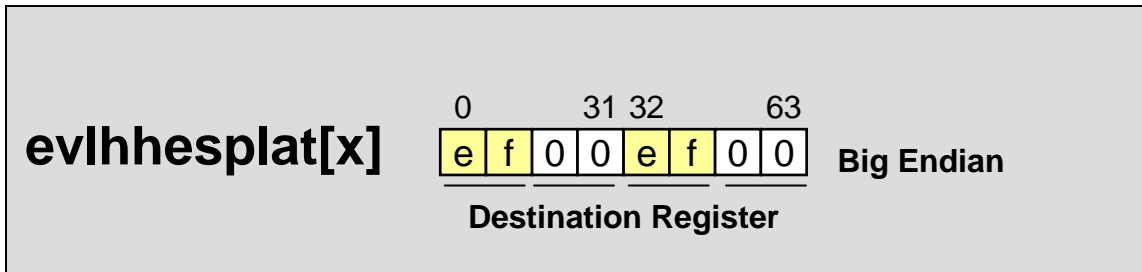
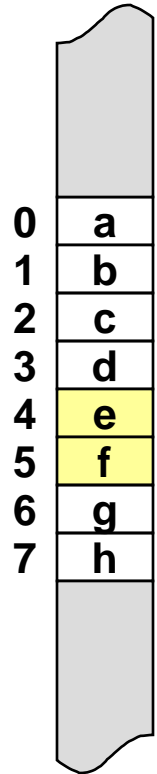


Word aligned!!

# Load Half Word Splat as Half Words



## Memory



Half word aligned!!

# SPE Data Types



- » 16-bit signed and unsigned integer
- » 16-bit fractional
- » 32-bit signed and unsigned integer
- » 32-bit fractional
- » 32-bit or single precision floating point

# 16-bit signed integer and fractional



## 16-bit signed integer

MSB													LSB			
$-2^{15}$	$2^{14}$	$2^{13}$													$2^1$	1

data values in range  $[-2^{15}, 2^{15})$

## 16-bit fractional

MSB													LSB			
-1	$2^{-1}$	$2^{-2}$													$2^{-14}$	$2^{-15}$

data values in range  $[-1, 1)$

# Multiply Accumulate Instructions



» evm | MULTIPLY | DATA TYPE | ACCUMULATE

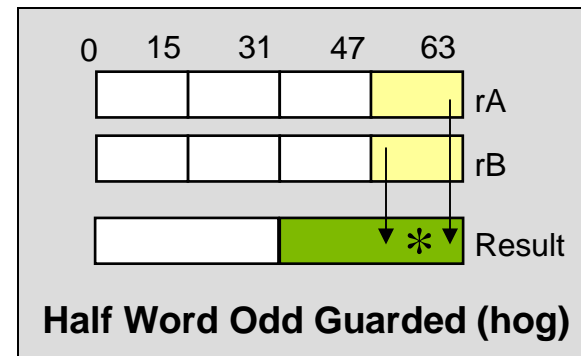
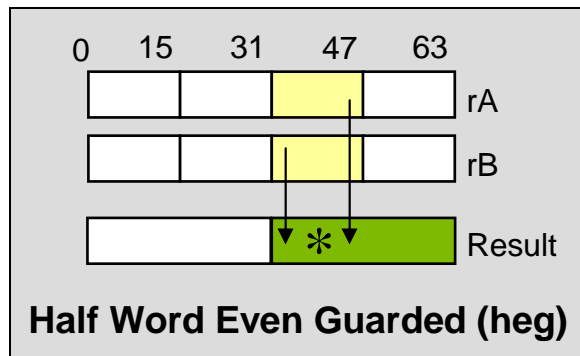
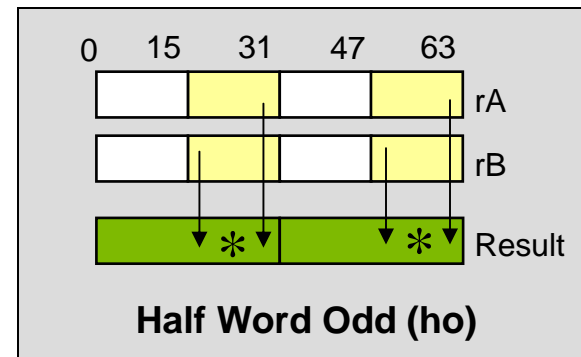
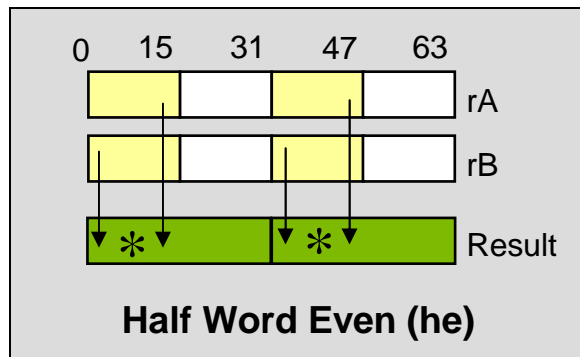


<b>Multiply</b>	
HO	Half word odd
HE	Half-word even
HOG	Half-word odd guarded
HEG	Half-word even guarded
WH	Word-High
WL	Word-Low
W	Word

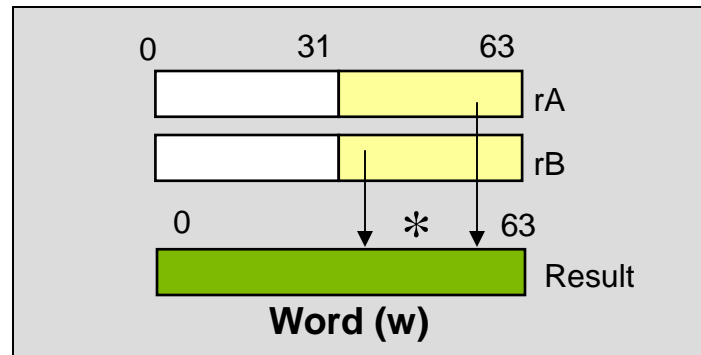
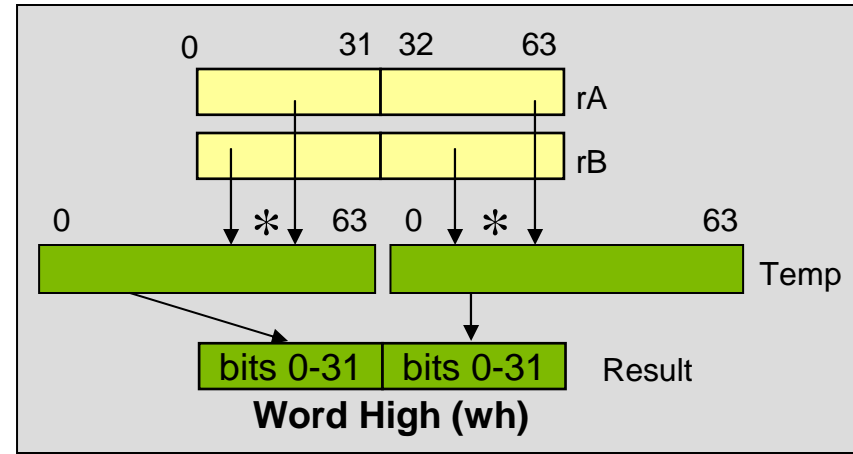
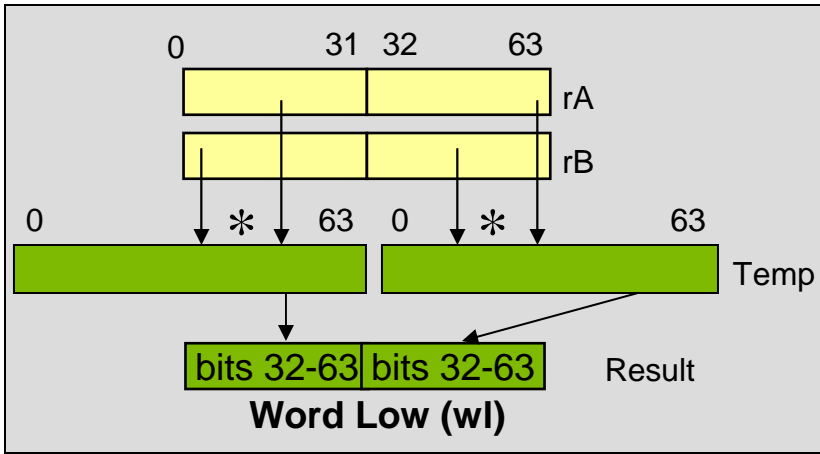
<b>Data Type</b>	
USI	Unsigned Saturate Integer
UMI	Unsigned Modulo Integer
SSI	Signed Saturate Integer
SMI	Signed Modulo Integer
SSF	Signed Saturate fractional
SMF	Signed Modulo fractional

<b>Accumulate</b>	
A	Update accumulator
AA	Update accumulator and added to accumulator
AN	Update accumulator and subtract from accumulator
AAW	Update accumulator and add to accumulator as words
ANW	Update accumulator and subtract from accumulator as words

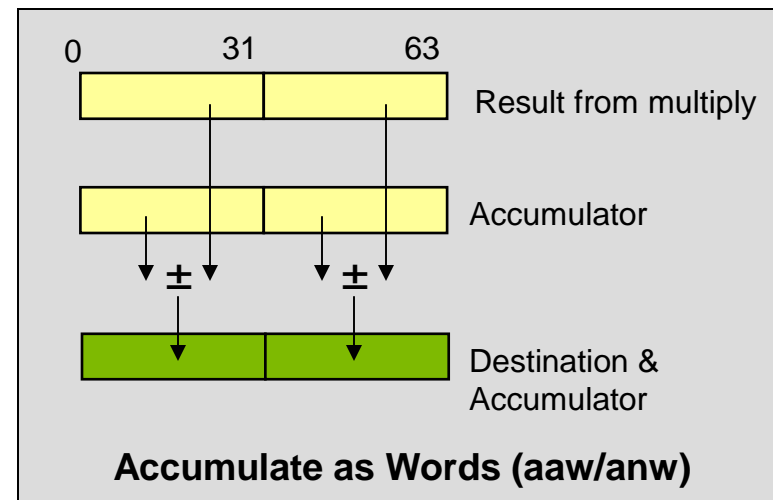
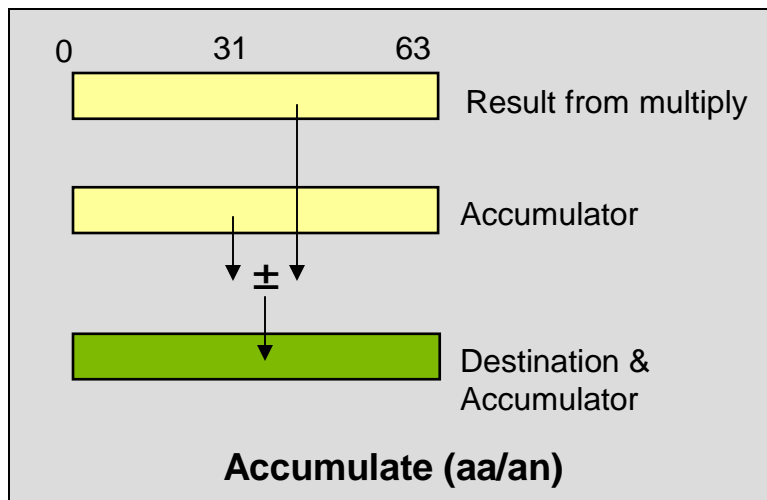
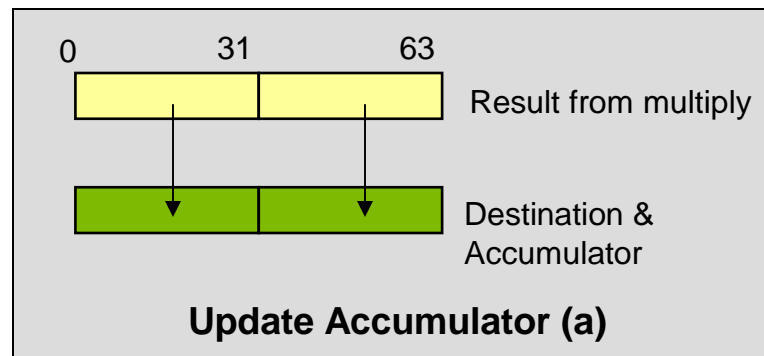
# Types of Half Word Multiplies



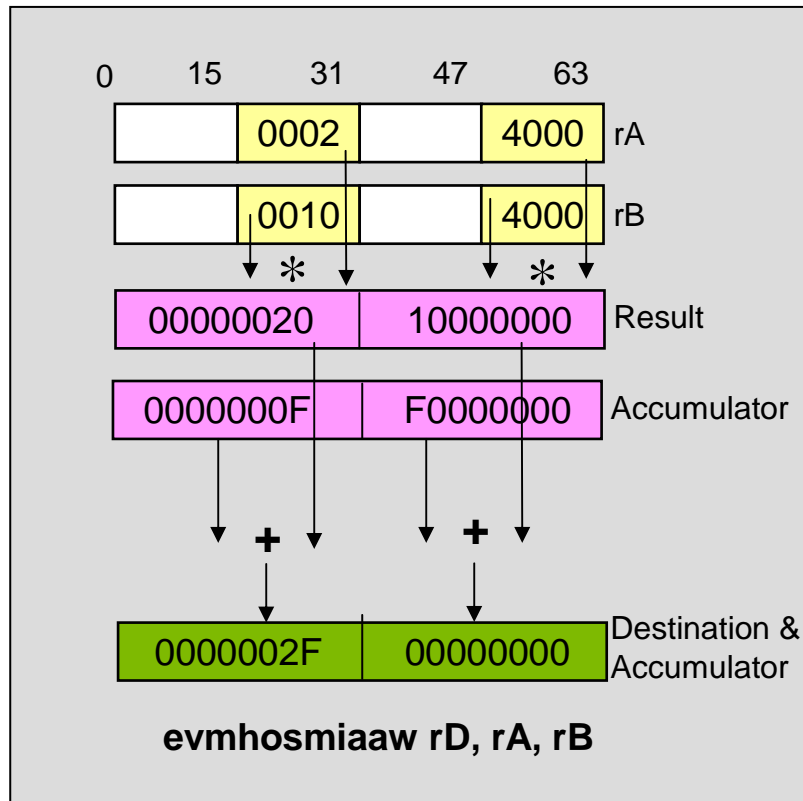
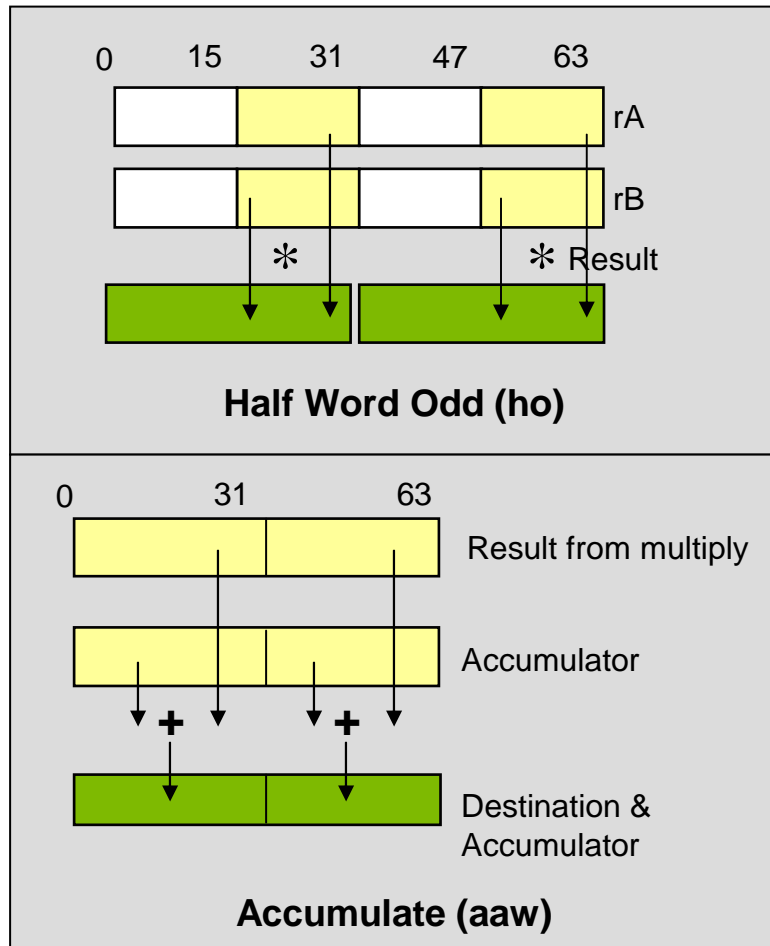
# Types of Word Multiplies



# Types of Accumulation



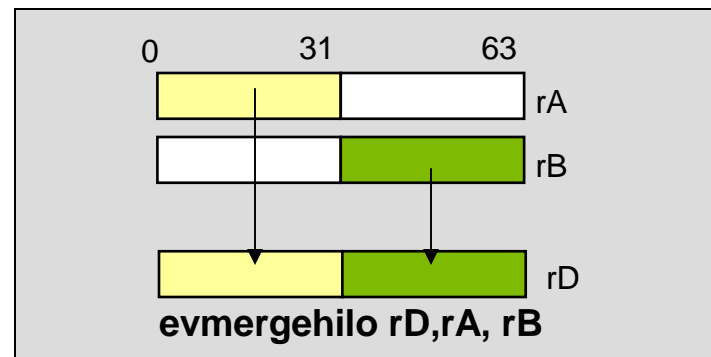
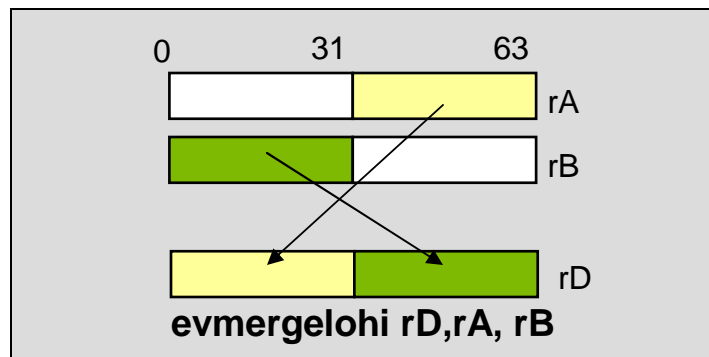
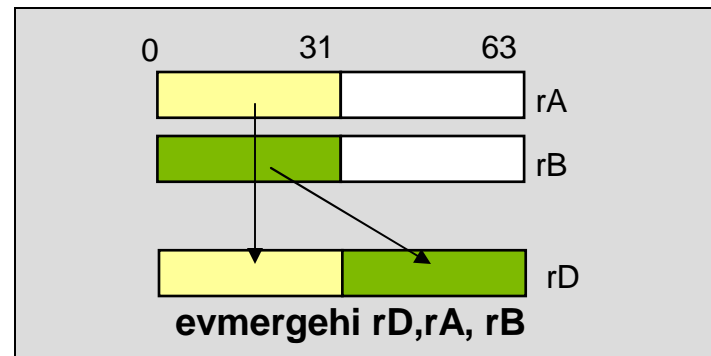
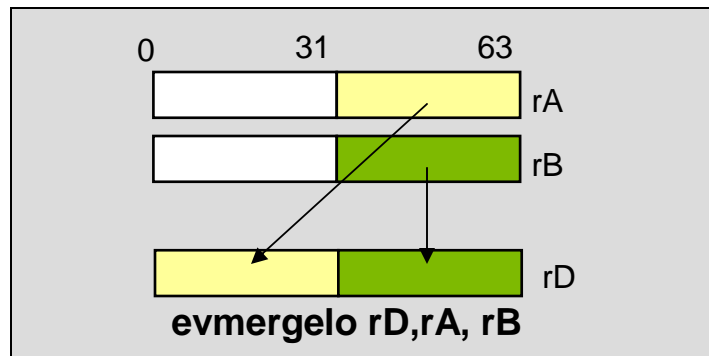
# evmhosmiaaw rD, rA, rB



# Merge



Useful for moving non-double-word aligned elements into upper half of register.



# Embedded Floating Point APU

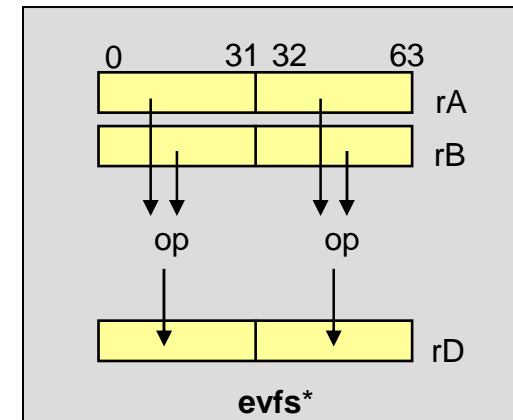
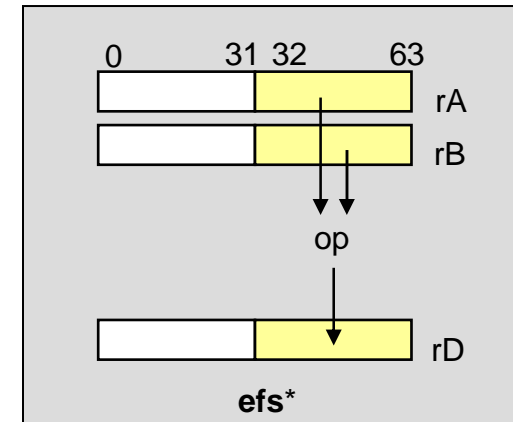


- » Provides single precision floating point capabilities
  - **IEEE® compliant with software exception handler**
  - **Uses GPRs instead of FPRs to save area and for fast float/integer/fractional conversions**
  - **Is a proper subset of the signal processing engine APU**
- » Uses IEEE single precision data format
- » “Normal” arithmetic handled according to IEEE 754, except near boundary conditions

# Floating Point



- » Support for both Scalar (efs\*) and Vector (evfs\*) operations:
- » Arithmetic: add, subtract, multiply, divide, negate, absolute value, negative absolute value
- » Comparisons: greater than, less than, equal to
- » Tests: greater than, less than, equal to
- » Converts: to/from integer, to/from fractional



Power Architecture™ '07  
DEVELOPER CONFERENCE

# Programming Interface Model

Power.ORG ™

The Power Architecture and Power.org word marks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org.

# Programming Interface Model



- » Provide high-level language access to SPE functionality through intrinsics
  - Typically maps to single instruction
- » Alleviate issues with writing assembly
  - Handling ABI calling and stack conventions
  - Handle register allocation
  - Handle code scheduling
  - Allows compiler to perform other optimizations
- » Let Compiler handle these tasks

# New Data Types/Initialization



```
» __ev64_opaque__ o; // can not be initialized
» __ev64_u16__ x = { 32, 65, 2, 2 };
» __ev64_s16__ y = { -32, 5, 7, -3 };
» __ev64_u32__ z = { 12, 0x2341 };
» __ev64_s32__ a = { -23, 01024 };
» __ev64_u64__ b = { 234123 };
» __ev64_s64__ c = { -2512341 };
» ev64 fs d = { -2.65, 6.2 };
```

# Mapping Instruction to C Level Intrinsics



## Arithmetic has one-to-one mapping:

```
evaddw rD, rA, rB maps:  
__ev64_opaque__ __ev_addw(__ev64_opaque__ a, __ev64_opaque__ b);
```

## Compares handled by predicates:

```
_Bool __ev_any_gts (__ev64_opaque__ a, __ev64_opaque__ b);  
  
__ev64_opaque__ __ev_select_ltu (a, b, c, d); // (a < b ? c : d)
```

## Load/Store handled by compiler and intrinsics:

```
__ev64_opaque__ *x;  
uint64_t y;  
x = &y;  
  
__ev64_opaque__ __ev_lddx (__ev64_opaque__ *p, int32_t index);
```

# Data Manipulation Intrinsics



## Create:

```
uint16_t a = 3, b = 4;  
__ev64_opaque__ x = __ev_create_u16(a, b, 13, 2);
```

## Get/Set:

```
uint32_t x = 4, y;  
__ev64_u32__ z = { 3, 1 };  
  
y = __ev_get_upper_u32 (z); // y = 3  
z = __ev_set_lower_u32(z, x); // z = {3, 4}
```

## Convert:

```
__ev64_u32__ z = {0x01020304, 0x05060708};  
uint64_t x = __ev_convert_u64(z); // x = 0x0102030405060708
```

Power Architecture™  
DEVELOPER CONFERENCE '07

# Programming Example

Power.ORG ™

The Power Architecture and Power.org word marks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org.

# Matrix Multiplication — C Code



```
int main(void) {
    volatile uint8_t i,j,k = 0;
    uint32_t starttime,endtime,ticdelay,loop_time,unrolled_code_time,asm_code_time[2];

    initTBM();

    starttime = tic();
    endtime = tic();
    ticdelay = endtime - starttime;

    starttime = tic();

    for(i = 0; i < M ; i++){
        for(k = 0; k < N ; k++){
            for(j = 0; j < P ; j++){
                C[i][k] = A[i][j]*B[j][k] + C[i][k];
            }
        }
    }

    endtime = tic();
    loop_time = endtime - starttime - ticdelay;

    . . .
}
```

# Matrix Multiplication — C Code



```
. . .  
  
starttime = tic();  
  
C[0][0] = A[0][0]*B[0][0] + A[0][1]*B[1][0];  
C[0][1] = A[0][0]*B[0][1] + A[0][1]*B[1][1];  
C[1][0] = A[1][0]*B[0][0] + A[1][1]*B[1][0];  
C[1][1] = A[1][0]*B[0][1] + A[1][1]*B[1][1];  
  
endtime = tic();  
unrolled_code_time = endtime - starttime - ticdelay;  
  
starttime = tic();  
matrix_mult(*A,*B,*C);  
endtime = tic();  
asm_code_time[0] = endtime - starttime - ticdelay;  
  
}
```

# Matrix Multiplication — SPE C Code



```
void matrix_mult(int16_t * A, int16_t * B, int32_t *C){

    asm(evlwvsplat  r8, 0(r3)); // load r8 = A[0][0] | A[0][1] | A[0][0] | A[0][1]
    asm(evlwhe     r9, 0(r4)); // load r9 = B[0][0] | 0x0000 | B[0][1] | 0x0000
    asm(evlwhou    r10, 4(r4)); // load r10 = 0x0000 | B[1][0] | 0x0000 | B[1][1]
    asm(evor       r9, r9, r10); // load r9 = B[0][0] | B[1][0] | B[0][1] | B[1][1]

    asm(evmhesmia  r10, r8,r9); // A[0][0]*B[0][0] ||A[0][0]*B[0][1]
    asm(evmhosmiaaw r10, r8,r9); // ACCH_HI + A[0][1]*B[1][0] || ACCH_LO + A[0][1]*B[1][1]
    asm(evstdw     r10, 0(r5)); // store to C[0][0] and C[0][1]

    asm(evlwvsplat r11, 4(r3)); // load r11 = A[1][0] | A[1][1] | A[1][0] | A[1][1]
    asm(evmhesmia  r12, r11,r9); // A[1][0]*B[0][0] ||A[1][0]*B[0][1]
    asm(evmhosmiaaw r12, r11,r9); // ACCH_HI + A[1][1]*B[1][0] || ACCH_LO + A[1][1]*B[1][1]
    asm(evstdw     r12, 8(r5)); // store to C[1][0] and C[1][1]

}
```

# FIR with 20 filter coefficients



» Output sequence  $y(0), y(1), \dots, (n-1)$  given by

» 
$$y(i) = h(0)x(i) + h(1)x(i-1) + \dots + h(19)x(i-19)$$

» Input sequence  $x(0), x(1), \dots, x(n-1)$

» Initial conditions  $x(-19), x(-18), \dots, x(-1)$

- Often assumed to be zero

» Coefficient sequence  $h(0), h(1), \dots, h(19)$

# FIR 20 filter coefficients — C Code



```
void fir(uint32_t *y, uint32_t n, uint32_t *x, uint32_t *h)
{
    uint32_t i, j;

    for(i = 19; i < n+19; i++) {
        for (j = 0; j < 20; j++) {
            y[i] += h[j] * x[i-j];
        }
    }
}
```

## Assumptions:

Pointers are aligned to double word addresses.

Value n is a multiple of 2.

Initial conditions followed by input sequence in memory.

# FIR 20 Filter Taps — SPE C Code

Compute two outputs at a time



```
void fir(int16_t *y, uint32_t n, int16_t *x, int16_t *h)
{
    uint32_t i, j;
    uint64_t zero =0;
    __ev64_opaque__ y0, z;
    __ev64_opaque__ tap[20];

    for(i = 0; i < 20; i++ )
        tap[i] = __ev_create_s32 ((int32_t) h[i], (int32_t) h[i]);

    for(i = 19; i < n+19; i+=2 ){
        __ev_set_acc_u64( zero ); //clear accumulator
        for(j = 0; j < 20; j++)
        {
            z = __ev_create_s32((int32_t) x[i-j],(int32_t) x[i+1-j]);
            y0 = __ev_mhossiaaw(tap[j], z);
        }
        y[i] = __ev_get_upper_u32(y0);
        y[i+1] = __ev_get_lower_u32(y0);
    }
}
```

# FIR 20 Filter Taps — Assembly Code



- » Compute two outputs  $y(i)$  and  $y(i+1)$  at a time
- » Splat coefficients  $h(0), h(1), \dots, h(19)$  into registers
- » Issue 20 back to back multiply accumulate instructions
- » Pair MAC instructions with loads and merges to achieve close to 2 IPC.
- » 29 registers
  - 20 for filter taps
  - 2 for input and output pointers
  - 2 for counter and N (number of elements)
  - 1 for output values

# FIR 20 Filter Taps — Assembly Code



```
Loop code:                                // 3 cycle load latency

    evlwhou    x_01, 0(xptr)                // x(0) | x(1)
    evlwhou    x_23, 4(xptr)                // x(2) | x(3)
    evlwhou    x_45, 8(xptr)                // x(4) | x(5)
    evlwhou    x_67, 12(xptr)               // x(6) | x(7)

    evmhossfa  y_N, x_01, h0                // x(0)h(0) | x(1)h(0)

    evmergelohi x_12, x_01, x_23            // x(1) | x(2)
    evhossfaaw y_N, x_23, h2                // x(2)h(2)+x(0)h(0) | x(3)h(2)+x(1)h(0)

    evmergelohi x_34, x_23, x_45            // x(2) | x(3)
    evmhossfaaw y_N, x_12, h1                // x(1)h(1) + x(2)h(2) + x(0)h(0) | ...

    evlwhou    x_89, 16(xptr)
    evmhossfaaw y_N, x_45, h4

    ...

    evstwhe    y_N, 0(yptr)                // store y(0) and y(1)
```

Power Architecture™  
DEVELOPER CONFERENCE '07

# Application Examples

Power.ORG ™

The Power Architecture and Power.org word marks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org.

# Kalman Filters Overview



The basic state-dependent Kalman filter equation is:

$$\hat{x}_{n+1} = \hat{A}_n \cdot \left( \hat{x}_n + K_n (y_n - \hat{C}_n \hat{x}_n) \right) + \hat{B}_n u_n$$

$$\hat{y}_n = \hat{C}_n \hat{x}_n$$

In the equation above, K is the filter gain and is calculated by:

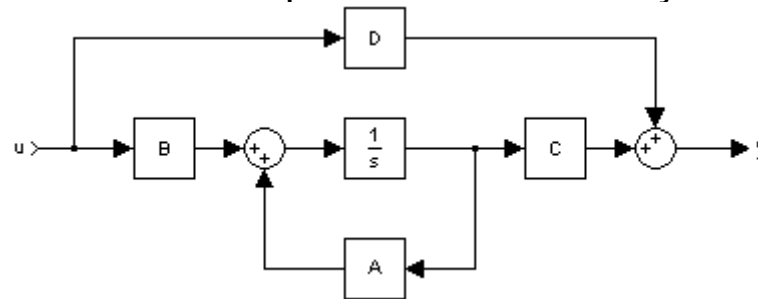
$$K_n = P_n \hat{C}_n^T \left( \hat{C}_n P_n \hat{C}_n^T + R_n \right)^{-1}$$

In this equation, R is the covariance of the measurement noise:

P is the solution of the Riccati equation and will be calculated off-line:

$$P_n = \hat{A}_n \left[ P_n - P_n \hat{C}_n^T \left( R + \hat{C}_n P_n \hat{C}_n^T \right)^{-1} \hat{C}_n P_n \right] \hat{A}_n^T + Q$$

The other variables come from the state-space model of the system (see diagram below):



# Basic Kalman Filter



- State prediction (4x1 matrix)
  - $x_{k+1} = A_k x_k + B_k U_k$
- Measurement prediction (2x1 matrix)
  - $y_{k+1} = C_k x_{k+1}$
- Covariance prediction (4x4 matrix)
  - $P_{k+1} = A_k P_k A_k' + Q$
- Residual Covariance (2x2 matrix)
  - $S_{k+1} = C_k P_{k+1} C_k' + R$
- Measurement Residual (2x1 matrix)
  - $e_{k+1} = y_{k+1} - y_{k+1}$
- Kalman Gain (2x4 matrix)
  - $K_{k+1} = P_{k+1} C_k' \text{inv}(S_{k+1})$
- state estimate
  - $x_{k+1} = x_{k+1} + K_{k+1} e_{k+1}$
- update state covariance (4x4 matrix)
  - $P_{k+1} = P_{k+1} - K_{k+1} S_{k+1} K_{k+1}'$

Basic Kalman Filter Implementation	
<b>Test</b>	
Rolled Loop in C *	36068
Unrolled Loop in C *	7120
SPE-optimized assembler	1540
<b>Ratio</b>	
C loop vs asm	23.421
C Unrolled vs asm	4.6

\* No compiler optimization switches were enabled

- » In blue: floating point variables and operations.
- » In orange: result converted to float.

# Inversion Algorithms



## » UD Algorithm

- Floating point implementation

Implementation	2*2	3*3	4*4	5*5	6*6	7*7	8*8	9*9	10*10
Rolled Loop in C *	1984	4092	7011	11109	16634	22786	30409	40163	52584
SPE-optimized assembler	81	171	380	590	961	1383	1915	2548	3304
<b>Ratio</b>									
C loop vs. asm	24.49	23.93	18.45	18.83	17.31	16.48	15.88	15.76	15.92

\* No compiler optimization switches were enabled

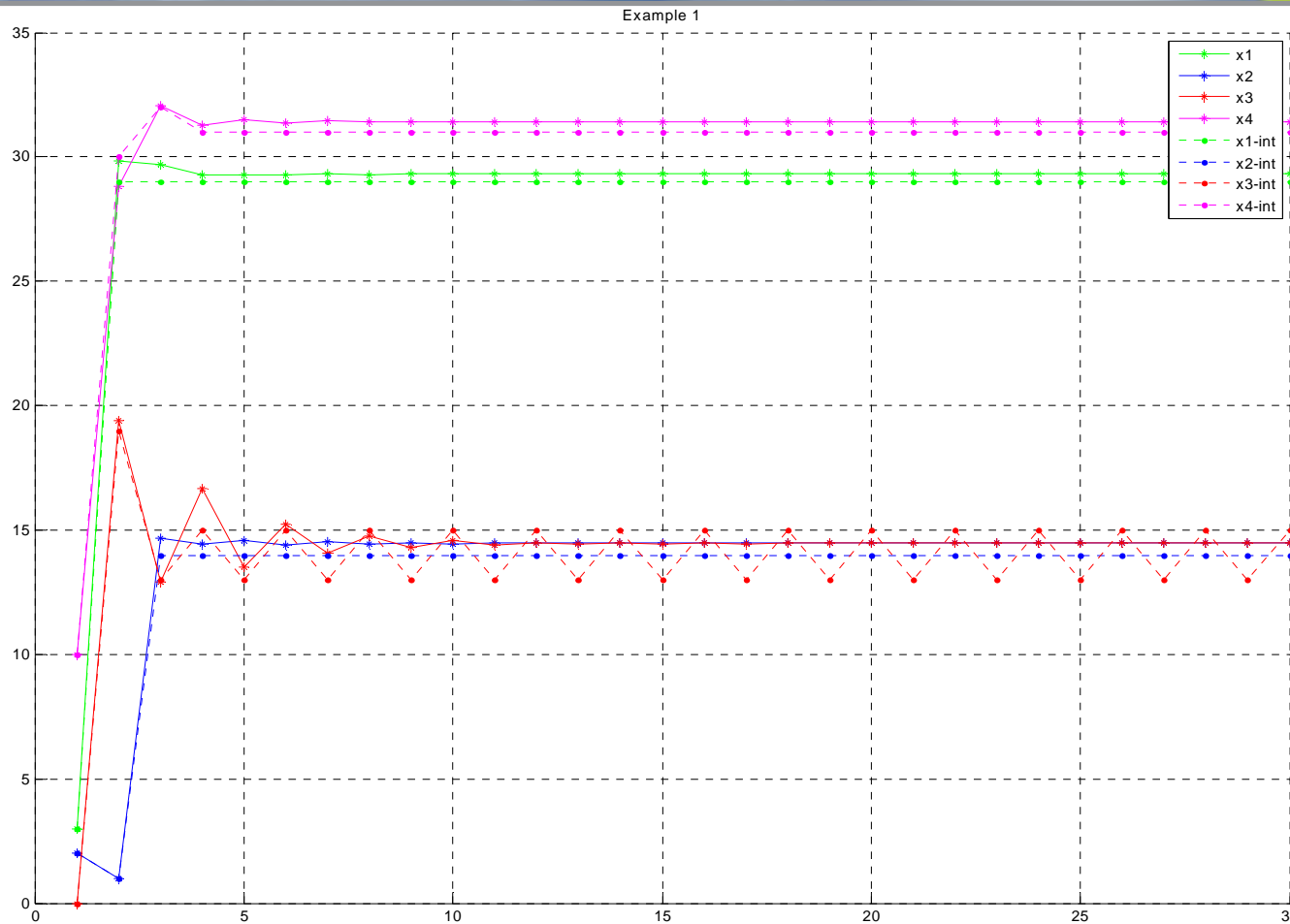
# Floating point vs integer multiplication



<b>Floating Point Implementation</b>		<b>2*2</b>	<b>4*4</b>	<b>6*6</b>	<b>8*8</b>
Test					
1	Rolled Loop in C *	1059	7693	26575	59193
2	Unrolled Loop in C *	170	1219	3976	9303
3	SPE-optimized assembler	<b>41</b>	<b>202</b>	<b>651</b>	<b>2100</b>
<b>16 bit Integers Implementation</b>		<b>2*2</b>	<b>4*4</b>	<b>6*6</b>	<b>8*8</b>
Test					
1	Rolled Loop in C *	1043	7565	26143	58169
2	Unrolled Loop in C *	161	1185	3913	9185
3	SPE-optimized assembler	<b>37</b>	<b>130</b>	<b>323</b>	<b>633</b>

\* No compiler optimization switches were enabled

# Accuracy Results – Example 1



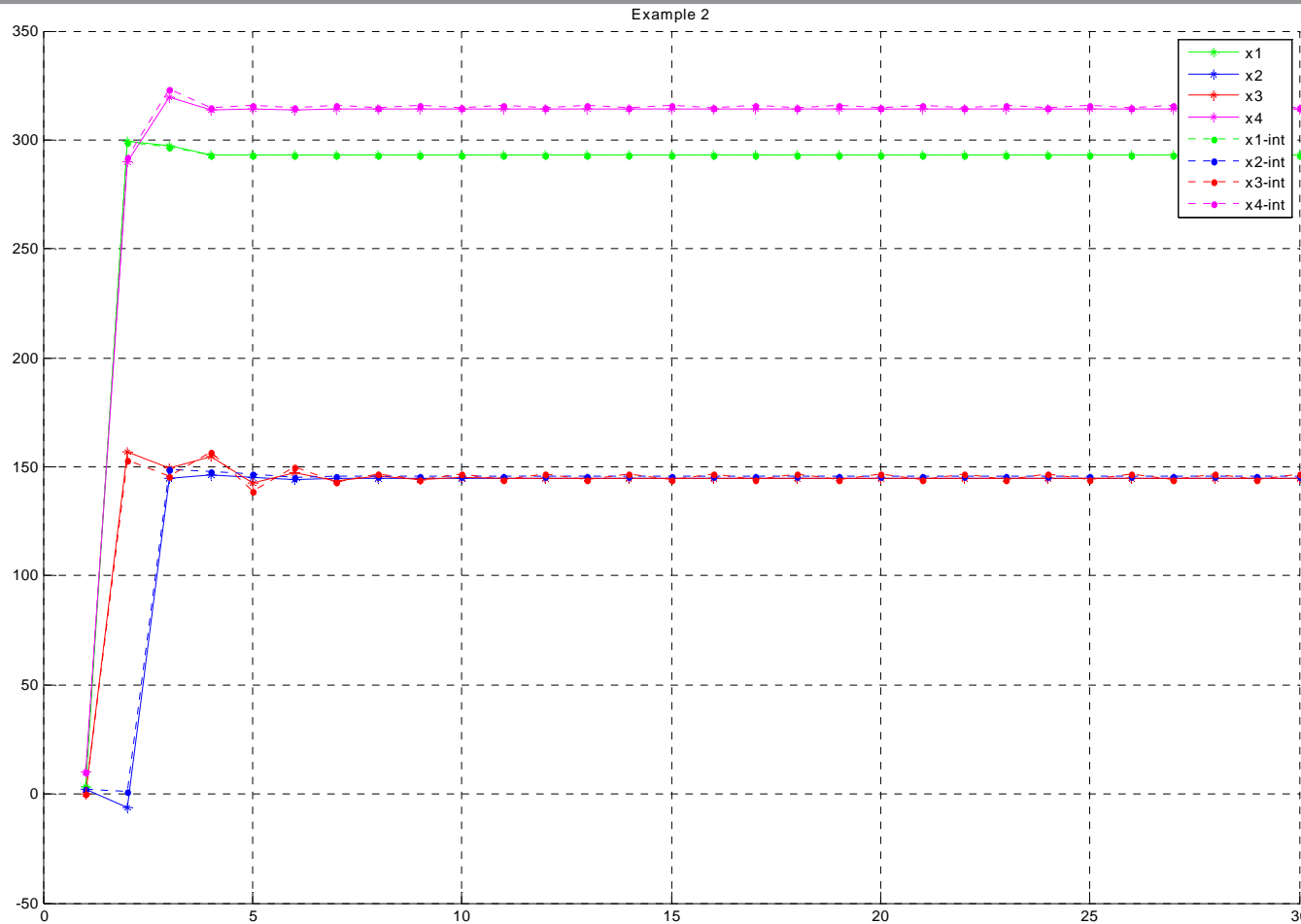
x1 = 1%

x2 = 3.3%

x3 = 10.1%  
to 3.7%

x4 = 1.3%

# Accuracy Results – Example 2



x1 = 0.3%  
x2 = 0.86%  
x3 = 1.1%  
x4 = 0.28%

**Power Architecture™**  
DEVELOPER CONFERENCE

**'07**

Power.ORG ™