



# RISC V Architecture

---

**Prof. H R Vanamala**

Department of Electronics and Communication Engg.

# RISC V ARCHITECTURE

---

## UNIT 4: Arithmetic for Computers

**Prof. H R Vanamala**

Department of Electronics and Communication Engineering

## Unit 4: Arithmetic for Computers - Going Faster: Subword Parallelism and Matrix Multiply

---



Declaration on **line 6** of the C code below uses the `__m256d` data Type – holds four doubleprecision floating-point values.

The intrinsic `_mm256_load_pd()` on **line 6** uses AVX instructions to load four double-precision floating-point numbers in parallel (`_pd`) from the matrix C into `c0`.

The address calculation `C+i+j*n` on **line 6** represents element `C[i+j*n]`. Symmetrically, the final step on **line 11** uses the intrinsic `_mm256_store_pd()` to store four double-precision floating-point numbers from `c0` into the matrix C.

Four elements in each iteration - **line 4** increments `i` by 4 instead of by 1 as on **line 3** of previous C code.

## Unit 4: Arithmetic for Computers - Going Faster: Subword Parallelism and Matrix Multiply

---



Inside the loops, on **line 9** we first load four elements of A again using `_mm256_load_pd()`.

To multiply these elements by one element of B, on **line 10** we use the intrinsic `_mm256_broadcast_sd()`, which makes four identical copies of the scalar double precision number—in this case an element of B—in one of the YMM registers.

Then use `_mm256_mul_pd()` on **line 9** to multiply the four doubleprecision results in parallel.

Finally, `_mm256_add_pd()` on **line 8** adds the four products to the four sums in c0.

## Unit 4: Arithmetic for Computers - Going Faster: Subword Parallelism and Matrix Multiply - X86 code

---



Resulting x86 code for the body of the inner loops produced by the compiler: Five AVX instructions—they all start with v and four of the five use pd for packed double precision—that correspond to the C intrinsics

,  
Similarities: both use 12 instructions, the integer instructions are nearly identical (but different registers), and the floating-point instruction

Differences: Just going from scalar double (sd) using XMM registers to packed double (pd) with YMM registers.

The one exception is **line 4**: Every element of A must be multiplied by one element of B.

One solution is to place four identical copies of the 64-bit B element side-by-side into the 256-bit YMM register - instruction vbroadcastsd is used

## Unit 4: Arithmetic for Computers - Going Faster: Subword Parallelism and Matrix Multiply

---



For matrices of dimensions of 32 by 32, the unoptimized DGEMM C code runs at 1.7 GigaFLOPS (FLoating point Operations Per Second) on one core of a 2.6 GHz Intel Core i7 (Sandy Bridge).

,

The optimized code C code performs at 6.4 GigaFLOPS.

The AVX version is 3.85 times as fast close to 4 times by using **subword parallelism**.

## Unit 4: Arithmetic for Computers - Going Faster: Subword Parallelism and Matrix Multiply

Optimized version of DGEMM using C intrinsics to generate AVX512 subword-parallel instructions for the x86.

```
1.  //include <x86intrin.h>
2.  void dgemm (size_t n, double* A, double* B, double* C)
3.  {
4.      for ( size_t i = 0; i < n; i+=4 )
5.          for ( size_t j = 0; j < n; j++ ) {
6.              __m256d c0 = _mm256_load_pd(C+i+j*n); /* c0 = C[i][j] */
7.              for( size_t k = 0; k < n; k++ )
8.                  c0 = _mm256_add_pd(c0, /* c0 += A[i][k]*B[k][j] */
9.                      _mm256_mul_pd(_mm256_load_pd(A+i+k*n),
10.                                     _mm256_broadcast_sd(B+k+j*n)));
11.              _mm256_store_pd(C+i+j*n, c0); /* C[i][j] = c0 */
12.          }
13. }
```

## Unit 4: Arithmetic for Computers - Going Faster: Subword Parallelism and Matrix Multiply

Assembly language produced by the compiler for the inner loop.

```
1. vmovapd (%r11),%ymm0           // Load 4 elements of C into %ymm0
2. mov     %rbx,%rcx              // register %rcx = %rbx
3. xor     %eax,%eax              // register %eax = 0
4. vbroadcastsd (%rax,%r8,1),%ymm1 // Make 4 copies of B element ←
5. add     $0x8,%rax              // register %rax = %rax + 8
6. vmulpd  (%rcx),%ymm1,%ymm1     // Parallel mul %ymm1,4 A elements
7. add     %r9,%rcx               // register %rcx = %rcx + %r9
8. cmp     %r10,%rax              // compare %r10 to %rax
9. vaddpd  %ymm1,%ymm0,%ymm0      // Parallel add %ymm1, %ymm0
10. jne     50 <dgemm+0x50>        // jump if not %r10 != %rax
11. add     $0x1,%esi             // register %esi = %esi + 1
12. vmovapd %ymm0,(%r11)          // Store %ymm0 into 4 C elements
```



## Unit 4: Arithmetic for Computers - Going Faster: Subword Parallelism and Matrix Multiply

---



- Primary difference from the previous program is that the original floating-point operations are now using YMM registers and the pd versions of the instructions for packed double precision instead of the sd version for scalar double precision.
- It is performing a single multiply-add instruction instead of separate multiply and add instruction.

## Unit 4: Arithmetic for Computers –Fallacies and Pitfalls

---



**Fallacy:** Just as a left shift instruction can replace an integer multiply by a power of 2, a right shift is the same as an integer division by a power of 2.

--- Shifting the bits right by  $n$  bits would seem to be the same as dividing by  $2^n$  which is true for unsigned integers. The problem is with signed integers –gives wrong result Ex:  $-5/4$

**Pitfall:** Floating-point addition is not associative larger range of Floating point Numbers

Ex:

$$\begin{aligned}c + (a + b) &= -1.5_{\text{ten}} \times 10^{38} + (1.5_{\text{ten}} \times 10^{38} + 1.0) \\&= -1.5_{\text{ten}} \times 10^{38} + (1.5_{\text{ten}} \times 10^{38}) \\&= 0.0\end{aligned}$$

$$\begin{aligned}(c + a) + b &= (-1.5_{\text{ten}} \times 10^{38} + 1.5_{\text{ten}} \times 10^{38}) + 1.0 \\&= (0.0_{\text{ten}}) + 1.0 \\&= 1.0\end{aligned}$$

**Fallacy:** Parallel execution strategies that work for integer data types also work for floating-point data types– holds good for two's complement integers, since integer addition is associative but since floating-point addition is not associative this does not hold –results in different output

**Fallacy:** Only theoretical mathematicians care about floating-point accuracy.  
Ex: The Pentium 40386 used a standard floating-point divide algorithm – cases with wrong result

## Unit 4: Arithmetic for Computers –Fallacies and Pitfalls

---



## Unit 4: Arithmetic for Computers –Fallacies and Pitfalls

---



## Unit 4: Arithmetic for Computers –Exercise Problems:

---





# THANK YOU

---

**Vanamala H R**

Department of Electronics and Communication