Data Alignment

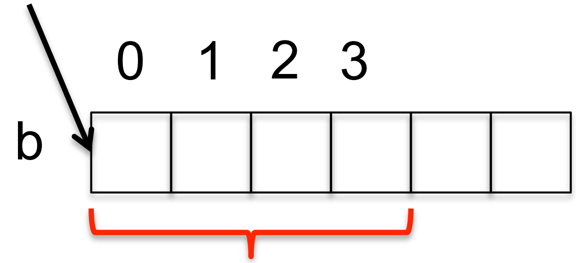
lwv

- Vector loads/stores load/store 128 consecutive bits to a vector register.
- Data addresses need to be 16-byte (128 bits) aligned to be loaded/ stored
  - Intel platforms support aligned and unaligned load/stores
  - IBM platforms do not support unaligned load/stores

128

```
void test1(float *a,float *b,float *c)
{
for (int i=0;i<LEN;i++){
  a[i] = b[i] + c[i];
}
```

Is &b[0] 16-byte aligned?



vector load loads b[0] … b[3]

Data Alignment

- To know if a pointer is 16-byte aligned, the last digit of the pointer address in hex must be 0.
- Note that if &b[0] is 16-byte aligned, and is a single precision array, then &b[4] is also 16-byte aligned

```
__attribute__ ((aligned(16))) float  B[1024];

int main(){
  printf("%p, %p\n", &B[0], &B[4]);
}
```

Output:
0x7fff1e9d8580, 0x7fff1e9d8590

Data Alignment

- In many cases, the compiler cannot statically know the alignment of the address in a pointer
- The compiler assumes that the base address of the pointer is 16-byte aligned and adds a run-time checks for it
  – if the runtime check is false, then it uses another code (which may be scalar)

Data Alignment

- Manual 16-byte alignment can be achieved by forcing the base address to be a multiple of 16.

```
__attribute__ ((aligned(16))) float b[N];
float* a = (float*) memalign(16,N*sizeof(float));
```

- When the pointer is passed to a function, the compiler should be aware of where the 16-byte aligned address of the array starts.

```
void func1(float *a, float *b,
float *c) {
  __assume_aligned(a, 16);
  __assume_aligned(b, 16);
  __assume_aligned(c, 16);
for int (i=0; i<LEN; i++) {
  a[i] = b[i] + c[i];
}
```

Data Alignment - Example

```
float A[N] __attribute__((aligned(16)));
float B[N] __attribute__((aligned(16)));
float C[N] __attribute__((aligned(16)));

void test(){
for (int i = 0; i < N; i++){
  C[i] = A[i] + B[i];
}}
```

Data Alignment - Example

```
float A[N] __attribute__((aligned(16)));
float B[N] __attribute__((aligned(16)));
float C[N] __attribute__((aligned(16)));
```

Aligned

```
void test1(){
__m128 rA, rB, rC;
 for (int i = 0; i < N; i+=4){
  rA = _mm_load_ps(&A[i]);
  rB = _mm_load_ps(&B[i]);
  rC = _mm_add_ps(rA,rB);
  _mm_store_ps(&C[i], rC);
}}
```

1

Unaligned

```
void test2(){
__m128 rA, rB, rC;
for (int i = 0; i < N; i+=4){
  rA = _mm_loadu_ps(&A[i]);
  rB = _mm_loadu_ps(&B[i]);
  rC = _mm_add_ps(rA,rB);
  _mm_storeu_ps(&C[i], rC);
}}
```

2

```
void test3(){
__m128 rA, rB, rC;
for (int i = 1; i < N-3; i+=4){
  rA = _mm_loadu_ps(&A[i]);
  rB = _mm_loadu_ps(&B[i]);
  rC = _mm_add_ps(rA,rB);
  _mm_storeu_ps(&C[i], rC);
}}
```

3

Nanosecond per iteration			
	Core 2 Duo	Intel i7	Power 7
1 Aligned	0.577	0.580	0.156
2 Aligned (unaligned ld)	0.689	0.581	0.241
3 Unaligned	2.176	0.629	0.243

Alignment in a struct

```
struct st{
  char A;
  int B[64];
  float C;
  int D[64];
};

int main(){
  st s1;
  printf("%p, %p, %p, %p\n", &s1.A, s1.B, &s1.C, s1.D);}
```

Output:
0x7fffe6765f00, 0x7fffe6765f04, 0x7fffe6766004, 0x7fffe6766008

- Arrays B and D are not 16-bytes aligned (see the address)

Alignment in a struct

```
struct st{
  char A;
  int B[64] __attribute__ ((aligned(16)));
  float C;
  int D[64] __attribute__ ((aligned(16)));
};

int main(){
  st s1;
  printf("%p, %p, %p, %p\n", &s1.A, s1.B, &s1.C, s1.D);}
```

Output:
0x7fff1e9d8580, 0x7fff1e9d8590, 0x7fff1e9d8690, 0x7fff1e9d86a0

- Arrays A and B are aligned to 16-byes (notice the 0 in the 4 least significant bits of the address)
- Compiler automatically does padding