Example:

```
for (int i=0;i<100;++i)
{
        m_iData++;
}
```

Seldom are compilers smart enough to figure out that the above loop resolves into m_iData+=100 and optimizes it into a single operation. Most will happily load m_iData at runtime, increment it, and store it back into memory referenced by the "this" pointer. The first pass of the loop will run at full speed, but once it loops back, the m_iData value will incur a Load-Hit-Store from the write operation of the previous pass through the loop.

Since registers invoke no penalty, if the code was rewritten to look like this:

```
int iData = m_iData;
for (int i=0;i<100;++i) {
        iData++;
}
m_iData = iData;
```

The current incrementation waits the last iData++ store operation

VectorLoad-Hit-Store

The previous examples demonstrated how easy it is to cause Load-Hit-Store stalls with floating-point and integer transactions. The VMX register sets suffer from the same problem. It's common that some math operations could be done more efficiently in a VMX operation, but what if it involves non-vector data?

On the Xbox 360, the VMX register intrinsic __vector4 is mapped onto a structure. Run-time accessing of the elements of the structure should be discouraged for the reason below.

```
XMVECTOR Radius = CalcBounds();
pOut->fRadius = Radius.x;
```

The second line creates a Load-Hit-Store because the VMX register is used as a structure. As a result, the compiler has to write the contents of the entire register to local memory; then the first element is read with a floating-point register, and only then is the value written into pOut->fRadius.

Here is a way to write the same code without incurring the hidden Load-Hit-Store:

```
XMVECTOR Radius = CalcBounds();
__stvewx(&pout->fRadius,__vspltw(Radius,0),0);
```

VMX has the ability to write any specific entry as a single float. The **vspltw()** operation will copy the requested entry into a temp vector register and the **stvewx()** operation will handle the writing the float. Using the compiler's feature of accessing the value isn't recommended.

CANNOT UNDERSTAND FOR NOW

References

A Load-Hit-Store can happen in code, even when it looks like it shouldn't.

```
void foo(int &count)        → memory bound
{
        count = 0;
        for (int i=0;i<100;++i) {
                if (Test(i)) {
                        ++count;
                }
        }
}
```

That code generates a Load-Hit-Store. How?

The variable "count" is memory bound. All writes to it, and in many cases reads, go through memory. Anytime a variable is memory bound and in a tight loop, it can cause Load-Hit-Stores. A way of fixing this is similar to the previous code example.

```
void foo(int &output)
{
        int count = 0;        → to local register
        for (int i=0;i<100;++i) {
                if (Test(i)) {
                        ++count;
                }
        }
        output = count; // Write the result
}
```

Eliminate int → float conversions

Example:

```
for (int i=0;i<100;++i)
{
        m_iData++;
}
```

Seldom are compilers smart enough to figure out that the above loop resolves into m_iData+=100 and optimizes it into a single operation. Most will happily load m_iData at runtime, increment it, and store it back into memory referenced by the "this" pointer. The first pass of the loop will run at full speed, but once it loops back, the m_iData value will incur a Load-Hit-Store from the write operation of the previous pass through the loop.

Since registers invoke no penalty, if the code was rewritten to look like this:

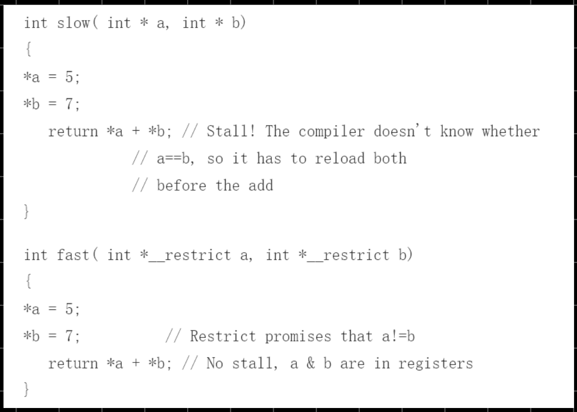
```
int iData = m_iData;
for (int i=0;i<100;++i) {
        iData++;
}
m_iData = iData;
```

```
for( DWORD i = 0; i<dwRingSegments; i++)
{
    FLOAT fAngle = (FLOAT)i * fAngleDelta;
```

→

```
FLOAT fi = 0.0f; // Added a copy of i as a float
for( DWORD i = 0; i<dwRingSegments; i++, fi+=1.0f ) // Inc fi
{
    FLOAT fAngle = fi * fAngleDelta; // NO int to float conversion
```

floating point compares has their own issues use _restrict or unique_ptr to avoid LHS



```
int slow( int * a, int * b)
{
*a = 5;
*b = 7;
   return *a + *b; // Stall! The compiler doesn't know whether
             // a==b, so it has to reload both
             // before the add
}

int fast( int *__restrict a, int *__restrict b)
{
*a = 5;
*b = 7;           // Restrict promises that a!=b
   return *a + *b; // No stall, a & b are in registers
}
```