Assembly

Contents

Registers

Registers are the fastest kind of memory available in the machine. x86-64 has 14 general-purpose registers and several special-purpose registers. This table gives all the basic registers, with special-purpose registers highlighted in yellow. You'll notice different naming conventions, a side effect of the long history of the x86 architecture (the 8086 was first released in 1978).

Full register (bits 0–63)	32-bit (bits 0 – 31)	16-bit (bits 0 – 15)	8-bit low (bits 0 – 7)	8-bit high (bits 8 – 15)	Use in calling convention	Callee-saved?
General-purpose registers:						
%rax	%eax	%ax	%al	%ah	Return value (accumulator)	No
%rbx	%ebx	%bx	%bl	%bh	–	Yes
%rcx	%ecx	%cx	%cl	%ch	4th function argument	No
%rdx	%edx	%dx	%dl	%dh	3rd function argument	No
%rsi	%esi	%si	%sil	–	2nd function argument	No
%rdi	%edi	%di	%dil	–	1st function argument	No
%r8	%r8d	%r8w	%r8b	–	5th function argument	No
%r9	%r9d	%r9w	%r9b	–	6th function argument	No
%r10	%r10d	%r10w	%r10b	–	–	No
%r11	%r11d	%r11w	%r11b	–	–	No
%r12	%r12d	%r12w	%r12b	–	–	Yes
%r13	%r13d	%r13w	%r13b	–	–	Yes
%r14	%r14d	%r14w	%r14b	–	–	Yes
%r15	%r15d	%r15w	%r15b	–	–	Yes
Special-purpose registers:						
%rsp	%esp	%sp	%spl	–	Stack pointer	Yes
%rbp	%ebp	%bp	%bpl	–	Base pointer (general-purpose in some compiler modes)	Yes
%rip	%eip	%ip	–	–	Instruction pointer (Program counter; called $pc in GDB)	*
%rflags	%eflags	%flags	–	–	Flags and condition codes	No

Note that unlike *primary* memory (which is what we think of when we discuss memory in a C/C++ program), registers have no addresses! There is no address value that, if cast to a pointer and dereferenced, would return the contents of the `%rax` register. Registers live in a separate world from the memory whose contents are partially prescribed by the C abstract machine.

The %rbp register has a special purpose: it points to the bottom of the current function's stack frame, and local variables are often accessed relative to its value. However, when optimization is on, the compiler may determine that all local variables can be stored in registers. This frees up %rbp for use as another general-purpose register.

The relationship between different register bit widths is a little weird.

1. Modifying a 32-bit register name sets the *upper* 32 bits of the register to zero. Thus, after `movl $-1, %eax`, the `%rax` register has value 0x0000'0000'FFFF'FFFF. The same is true after `movq $-1, %rax; addl $0, %eax`! (The `movq` sets `%rax` to 0xFFFF'FFFF'FFFF'FFFF; the `addl` sets its upper 32 bits to zero.)
2. Modifying a 16- or 8-bit register name *leaves all other bits of the register unchanged.*

There are special instructions for loading signed and unsigned 8-, 16-, and 32-bit quantities into registers, recognizable by instruction suffixes. For instance, `movzbl` moves an 8-bit quantity (a **b**yte) into a 32-bit register (a **l**ongword) with **z**ero extension; `movslq` moves a 32-bit quantity (**l**ongword) into a 64-bit register (**q**uadword) with **s**ign extension. There's no need for `movzlq` (why?).

Instruction format

The basic kinds of assembly instructions are:

1. **Arithmetic.** These instructions perform computation on values, typically values stored in registers. Most have zero or one *source operands* and one *source/destination operand*. The source operand is listed first in the *instruction*, but the source/destination operand comes first in the *computation* (this matters for non-commutative operators like subtraction). For example, the instruction `addq %rax, %rbx` performs the computation `%rbx := %rbx + %rax`.
2. **Data movement.** These instructions move data between registers and memory. Almost all have one source operand and one destination operand; the source operand comes first.
3. **Control flow.** Normally the CPU executes instructions in sequence. Control flow instructions change the instruction pointer in other ways. There are unconditional branches (the instruction pointer is set to a new value), conditional branches (the instruction pointer is set to a new value if a condition is true), and function call and return instructions.

(We use the "AT&T syntax" for x86-64 assembly. For the "Intel syntax," which you can find in online documentation from Intel, see the Aside in CS:APP3e §3.3, p177, or Wikipedia, or other online resources. AT&T syntax is distinguished by several features, but especially by the use of percent signs for registers. Sadly, the Intel syntax puts destination registers *before* source registers.)

Some instructions appear to combine arithmetic and data movement. For example, given the C code `int* ip; ... ++(*ip);` the compiler might generate `incl (%rax)` rather than `movl (%rax), %ebx; incl %ebx; movl %ebx, (%rax)`. However, the processor actually divides these complex instructions into tiny, simpler, invisible instructions called microcode, because the simpler instructions can be made to execute faster. The complex `incl` instruction actually runs in three phases: data movement, then arithmetic, then data movement. This matters when we introduce parallelism.

Directives

Assembly generated by a compiler contains instructions as well as *labels* and *directives*. Labels look like `labelname:` or `labelnumber:`; directives look like `.directivename arguments`. Labels are markers in the generated assembly, used to compute addresses. We usually see them used in control flow instructions, as in `jmp L3` ("jump to L3"). Directives are instructions to the assembler; for instance, the `.globl L` instruction says "label `L` is globally visible in the executable", `.align` sets the alignment of the following data, `.long` puts a number in the output, and `.text` and `.data` define the current segment.

We also frequently look at assembly that is *disassembled* from executable instructions by GDB, `objdump -d`, or `objdump -S`. This output looks different from compiler-generated assembly: in disassembled instructions, there are no intermediate labels or directives. This is because the labels and directives disappear during the process of generating executable instructions.

For instance, here is some compiler-generated assembly:

```
    .globl  _Z1fiii
    .type   _Z1fiii, @function
_Z1fiii:
.LFB0:
    cmpl    %edx, %esi
    je  .L3
    movl    %esi, %eax
    ret
.L3:
    movl    %edi, %eax
    ret
.LFE0:
    .size   _Z1fiii, .-_Z1fiii
```

And a disassembly of the same function, from an object file:

```
0000000000000000 <_Z1fiii>:
   0:   39 d6                   cmp    %edx,%esi
   2:   74 03                   je     7 <_Z1fiii+0x7>
   4:   89 f0                   mov    %esi,%eax
   6:   c3                      retq
   7:   89 f8                   mov    %edi,%eax
   9:   c3                      retq
```

Everything but the instructions is removed, and the helpful `.L3` label has been replaced with an actual address. The function appears to be located at address 0. This is just a placeholder; the final address is assigned by the linking process, when a final executable is created.

Finally, here is some disassembly from an executable:

```
0000000000400517 <_Z1fiii>:
  400517:   39 d6                   cmp    %edx,%esi
  400519:   74 03                   je     40051e <_Z1fiii+0x7>
  40051b:   89 f0                   mov    %esi,%eax
  40051d:   c3                      retq
  40051e:   89 f8                   mov    %edi,%eax
  400520:   c3                      retq
```

The instructions are the same, but the addresses are different. (Other compiler flags would generate different addresses.)

Address modes

Most instruction operands use the following syntax for values. (See also CS:APP3e Figure 3.3 in §3.4.1, p181.)

Type	Example syntax	Value used
Register	`%rbp`	Contents of `%rbp`
Immediate	`$0x4`	0x4
Memory	`0x4`	Value stored at address 0x4
	`symbol_name`	Value stored in global `symbol_name` (the compiler resolves the symbol name to an address when creating the executable)
	`symbol_name(%rip)`	`%rip`-relative addressing for global
	`symbol_name+4(%rip)`	Simple arithmetic on symbols are allowed (the compiler resolves the arithmetic when creating the executable)
	`(%rax)`	Value stored at address in `%rax`
	`0x4(%rax)`	Value stored at address `%rax + 4`
	`(%rax,%rbx)`	Value stored at address `%rax + %rbx`
	`(%rax,%rbx,4)`	Value stored at address `%rax + %rbx*4`
	`0x18(%rax,%rbx,4)`	Value stored at address `%rax + 0x18 + %rbx*4`

The full form of a memory operand is `offset(base,index,scale)`, which refers to the address `offset + base + index*scale`. In `0x18(%rax,%rbx,4)`, `%rax` is the base, `0x18` the offset, `%rbx` the index, and `4` the scale. The offset (if used) must be a constant and the base and index (if used) must be registers; the scale must be either 1, 2, 4, or 8. The default offset, base, and index are 0, and the default scale is 1.

`symbol_name`s are found only in compiler-generated assembly; disassembly uses raw addresses (`0x601030`) or `%rip`-relative offsets (`0x200bf2(%rip)`).

Jumps and function call instructions use different syntax 🤷: `*`, rather than `()`, represents indirection.

Type	Example syntax	Address used
Register	`*%rax`	Contents of `%rax`
Immediate	`.L3`	Address of `.L3` (compiler-generated assembly)
	`400410` or `0x400410`	Given address
Memory	`*0x200b96(%rip)`	Value stored at address `%rip + 0x200b96`
	`*(%r12,%rbp,8)`	Other address modes accepted

Address arithmetic

The `base(offset,index,scale)` form compactly expresses many array-style address computations. It's typically used with a `mov`-type instruction to dereference memory. However, the compiler can use that form to compute addresses, thanks to the `lea` (Load Effective Address) instruction.

For instance, in `movl 0x18(%rax,%rbx,4), %ecx`, the address `%rax + 0x18 + %rbx*4` is computed, then immediately dereferenced: the 4-byte value located there is loaded into `%ecx`. In `leaq 0x18(%rax,%rbx,4), %rcx`, the same address is computed, but it is *not* dereferenced. Instead, the *computed address* is moved into register `%rcx`.

Thanks to `lea`, the compiler will also prefer the `base(offset,index,scale)` form over `add` and `mov` for certain arithmetic computations on integers. For example, this instruction:

```
leaq (%rax,%rbx,2), %rcx
```

performs the function `%rcx := %rax + 2 * %rbx`, but in one instruction, rather than three (`movq %rax, %rcx; addq %rbx, %rcx; addq %rbx, %rcx`).

`%rip`-relative addressing

x86-64 code often refers to globals using **%rip-relative** addressing: a global variable named `a` is referenced as `a(%rip)` rather than `a`.

This style of reference supports *position-independent code* (PIC), a security feature. It specifically supports *position-independent executables* (PIEs), which are programs that work independently of where their code is loaded into memory.

To run a conventional program, the operating system loads the program's instructions into memory *at a fixed address that's the same every time*, then starts executing the program at its first instruction. This works great, and runs the program in a predictable execution environment (the addresses of functions and global variables are the same every time). Unfortunately, the very predictability of this environment makes the program easier to attack.

In a position-independent executable, the operating system loads the program at *varying* locations: every time it runs, the program's functions and global variables have different addresses. This makes the program harder to attack (though not impossible).

Program startup performance matters, so the operating system doesn't recompile the program with different addresses each time. Instead, the compiler does most of the work in advance by using *relative addressing*.

When the operating system loads a PIE, it picks a starting point and loads all instructions and globals relative to that starting point. The PIE's instructions never refer to global variables using direct addressing: you'll never see `movl global_int, %eax`. Globals are referenced *relatively* instead, using deltas relative to the next `%rip`: `movl global_int(%rip), %eax`. These relative addresses work great independent of starting point! For instance, consider an instruction located at (starting-point + 0x80) that loads a variable `g` located at (starting-point + 0x1000) into `%rax`. In a non-PIE, the instruction might be written `movq 0x400080, %rax` (in compiler output, `movq g, %rax`); but this relies on `g` having a fixed address. In a PIE, the instruction might be written `movq 0xf79(%rip), %rax` (in compiler output, `movq g(%rip), %rax`), which works out beautifully no matter the starting point.

At starting point…	The `mov` instruction is at…	The next instruction is at…	And `g` is at…	So the delta (`g` − next `%rip`) is…
0x400000	0x400080	0x400087	0x401000	0xF79
0x404000	0x404080	0x404087	0x405000	0xF79
0x4003F0	0x400470	0x400477	0x4013F0	0xF79

Arithmetic instructions

The operations of many x86-64 arithmetic instructions are easy enough to guess from their names. Then there are some arithmetic instructions, particularly those associated with Streaming SIMD Extensions and its follow-ons, that are hard to guess (`phminposuw`?).

The basic arithmetic instructions on 64-bit quantities ("quadwords") are:

Instruction	Operation	Type	Expansion
`addq SRC, DST`	Addition	Normal	`DST := DST + SRC`
`subq SRC, DST`	Subtraction	Normal	`DST := DST - SRC`
`incq DST`	Increment	Normal	`DST := DST + 1`
`decq DST`	Decrement	Normal	`DST := DST - 1`
`imulq SRC, DST`	Signed multiplication	Normal	`DST := DST * SRC`
`negq DST`	Negation	Normal	`DST := -DST` (`DST := ~DST + 1`)
`andq SRC, DST`	Bitwise and	Normal	`DST := DST & SRC`
`orq SRC, DST`	Bitwise or	Normal	`DST := DST | SRC`
`xorq SRC, DST`	Bitwise exclusive or	Normal	`DST := DST ^ SRC`
`notq DST`	Complement	Normal	`DST := ~DST`
`sal SRC, DST` (`shl SRC, DST`)	Left shift	Normal	`DST := DST << SRC`
`sar SRC, DST`	Signed right shift	Normal	`DST:= DST>>SRC`, shifting in sign bit
`shr SRC, DST`	Unsigned right shift	Normal	`DST:= DST>>SRC`, shifting in zeros
`cmpq SRC, DST`	Subtraction for flags	Flags-only	`DST - SRC`; see below
`testq SRC, DST`	Bitwise and for flags	Flags-only	`DST & SRC`; see below

There are also compact multiplication and division instructions that modify multiple registers at once and take fixed registers for some arguments. These instructions treat the combination of `%rax` and `%rdx` as a single 128-bit value where the most significant bits (bits 64–127) of the value are stored in `%rdx` and the least significant bits (0–63) are stored in `%rax`. The division instructions compute both a quotient and a remainder. (In the below, `TMP` is a 128-bit number.)

Instruction	Operation	Type	Expansion
`imulq SRC`	Signed multiplication	Mul/div	`TMP := %rax * SRC; %rdx := TMP>>64; %rax := TMP`
`mulq SRC`	Unsigned multiplication	Mul/div	`TMP := %rax * SRC; %rdx := TMP>>64; %rax := TMP`
`idivq SRC`	Signed division	Mul/div	`TMP := (%rdx<<64) | %rax; %rax := TMP / SRC; %rdx := TMP % SRC`
`divq SRC`	Unsigned division	Mul/div	`TMP := (%rdx<<64) | %rax; %rax := TMP / SRC; %rdx := TMP % SRC`

Calling convention

A **calling convention** governs how functions on a particular architecture and operating system interact. This includes rules about includes how function arguments are placed, where return values go, what registers functions may use, how they may allocate local variables, and so forth. Calling conventions ensure that functions compiled by different compilers can interoperate, and they ensure that operating systems can run code from different programming languages and compilers. Some aspects of a calling convention are derived from the instruction set itself, but some are conventional, meaning decided upon by people (for instance, at a convention).

Calling conventions constrain both *callers* and *callees*. A caller is a function that calls another function; a callee is a function that was called. The currently-executing function is a callee, but not a caller.

For concreteness, we learn the x86-64 calling conventions for Linux. These conventions are shared by many OSes, including MacOS (but not Windows), and are officially called the "System V AMD64 ABI."

The official specification: AMD64 ABI

Argument passing and stack frames

One set of calling convention rules governs how function arguments and return values are passed. On x86-64 Linux, the first six function arguments are passed in registers `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, and `%r9`, respectively. The seventh and subsequent arguments are passed on the stack, about which more below. The return value is passed in register `%rax`.

The full rules more complex than this. You can read them in the AMD64 ABI, section 3.2.3, but they're quite detailed. Some highlights:

1. A structure argument that fits in a single machine word (64 bits/8 bytes) is passed in a single register.
   Example: `struct small { char a1, a2; }`
2. A structure that fits in two to four machine words (16–32 bytes) is passed in sequential registers, as if it were multiple arguments.
   Example: `struct medium { long a1, a2; }`
3. A structure that's larger than four machine words is always passed on the stack.
   Example: `struct large { long a, b, c, d, e, f, g; }`

4. Floating point arguments are generally passed in special registers, the “SSE registers,” that we don’t discuss further.
5. Return values of up to one machine word are passed in `%rax`. Many return values that fit in two machine words—for instance, a pair of `long`s—are passed in `%rax` (for the first 8 bytes) and `%rdx` (for the second 8 bytes). For return values that fit only in three or more machine words, the *caller* reserves space for the return value, and passes the *address* of that space as the first argument of the function. The callee will fill in that space when it returns.

Writing small programs to demonstrate these rules is a pleasant exercise; for example:

```
struct small { char a1, a2; };
int f(small s) {
    return s.a1 + 2 * s.a2;
}
```

compiles to:

```
movl %edi, %eax           # copy argument to %eax
movsbl %dil, %edi         # %edi := sign-extension of lowest byte of argument
(s.a1)
movsbl %ah, %eax          # %eax := sign-extension of 2nd byte of argument
(s.a2)
movsbl %al, %eax
leal (%rdi,%rax,2), %eax  # %eax := %edi + 2 * %eax
ret
```

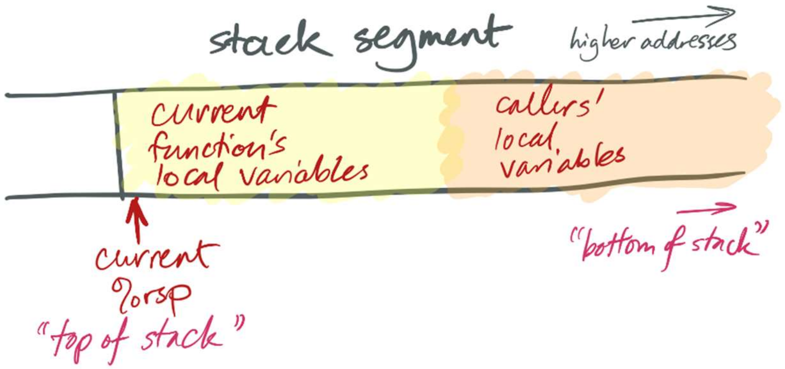
Stack

Recall that the stack is a segment of memory used to store objects with automatic lifetime. Typical stack addresses on x86-64 look like `0x7ffd'9f10'4f58`—that is, close to 2^{47}.

The stack is named after a data structure, which was sort of named after pancakes. Stack data structures support at least three operations: **push** adds a new element to the “top” of the stack; **pop** removes the top element, showing whatever was underneath; and **top** accesses the top element. Note what’s missing: the data structure does not allow access to elements other than the top. (Which is sort of how stacks of pancakes work.) This restriction can speed up stack implementations.

Like a stack data structure, the stack memory segment is only accessed from the top. The currently running function accesses *its* local variables; the function’s caller, grand-caller, great-grand-caller, and so forth are dormant until the currently running function returns.

x86-64 stacks look like this:



The x86-64 `%rsp` register is a special-purpose register that defines the current “stack pointer.” This holds the address of the current top of the stack. On x86-64, as on many architectures, stacks grow *down*: a “push” operation adds space for more automatic-lifetime objects by moving the stack pointer left, to a numerically-smaller address, and a “pop” operation recycles space by moving the stack pointer right, to a numerically-larger address. This means that, considered numerically, the “top” of the stack has a smaller address than the “bottom.”

This is built in to the architecture by the operation of instructions like `pushq`, `popq`, `call`, and `ret`. A `push` instruction pushes a value onto the stack. This both modifies the stack pointer (making it smaller) and modifies the stack segment (by moving data there). For instance, the instruction `pushq X` means:

```
subq $8, %rsp
movq X, (%rsp)
```

And `popq X` undoes the effect of `pushq X`. It means:

```
movq (%rsp), X
addq $8, %rsp
```

`X` can be a register or a memory reference.

The portion of the stack reserved for a function is called that function’s **stack frame**. Stack frames are aligned: x86-64 requires that each stack frame be a multiple of 16 bytes, and when a `callq` instruction begins execution, the `%rsp` register must be 16-byte aligned. This means that every function’s entry `%rsp` address will be 8 bytes off a multiple of 16.

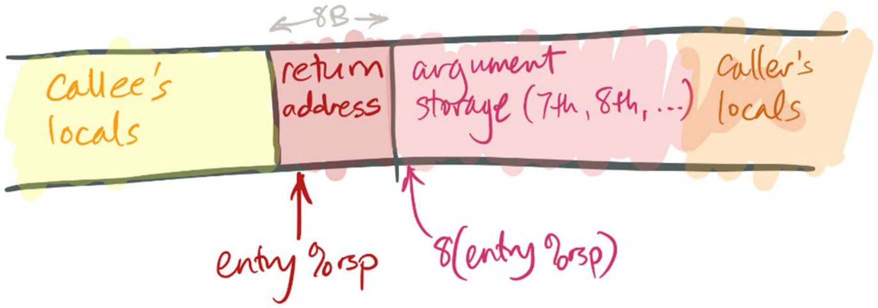
Return address and entry and exit sequence

The steps required to call a function are sometimes called the *entry sequence* and the steps required to return are called the *exit sequence*. Both caller and callee have responsibilities in each sequence.

To prepare for a function call, the caller performs the following tasks in its entry sequence.

1. The caller stores the first six arguments in the corresponding registers.
2. If the callee takes more than six arguments, or if some of its arguments are large, the caller must store the surplus arguments on its stack frame. It stores these in increasing order, so that the 7th argument has a smaller address than the 8th argument, and so forth. The 7th argument must be stored at `(%rsp)` (that is, the top of the stack) when the caller executes its `callq` instruction.
3. The caller saves any caller-saved registers (see below).
4. The caller executes `callq FUNCTION`. This has an effect like `pushq $NEXT_INSTRUCTION; jmp FUNCTION` (or, equivalently, `subq $8, %rsp; movq $NEXT_INSTRUCTION, (%rsp); jmp FUNCTION`), where `NEXT_INSTRUCTION` is the address of the instruction immediately following `callq`.

This leaves a stack like this:



To return from a function:

1. The callee places its return value in `%rax`.
2. The callee restores the stack pointer to its value at entry (“entry `%rsp`”), if necessary.
3. The callee executes the `retq` instruction. This has an effect like `popq %rip`, which removes the return address from the stack and jumps to that address.
4. The caller then cleans up any space it prepared for arguments and restores caller-saved registers if necessary.

Particularly simple callees don't need to do much more than return, but most callees will perform more tasks, such as allocating space for local variables and calling functions themselves.

Callee-saved registers and caller-saved registers

The calling convention gives callers and callees certain guarantees and responsibilities about the values of registers across function calls. Function implementations may expect these guarantees to hold, and must work to fulfill their responsibilities.

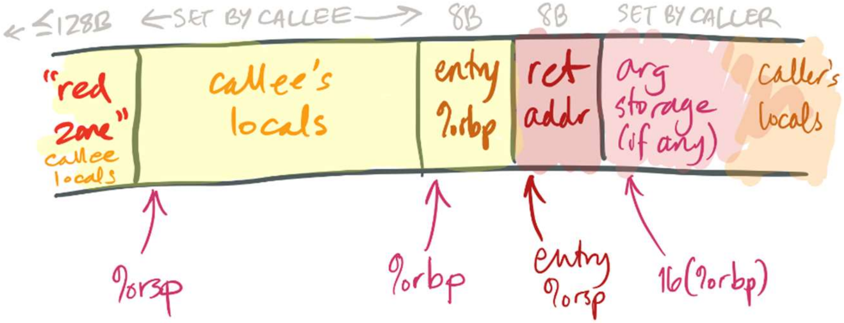
The most important responsibility is that certain registers' values *must be preserved across function calls*. A callee may use these registers, but if it changes them, it must restore them to their original values before returning. These registers are called **callee-saved registers**. All other registers are **caller-saved**.

Callers can simply use callee-saved registers across function calls; in this sense they behave like C++ local variables. Caller-saved registers behave differently: if a caller wants to preserve the value of a caller-saved register across a function call, the caller must explicitly save it before the `callq` and restore it when the function resumes.

On x86-64 Linux, `%rbp`, `%rbx`, `%r12`, `%r13`, `%r14`, and `%r15` are callee-saved, as (sort of) are `%rsp` and `%rip`. The other registers are caller-saved.

Base pointer (frame pointer)

The `%rbp` register is called the *base pointer* (and sometimes the *frame pointer*). For simple functions, an optimizing compiler generally treats this like any other callee-saved general-purpose register. However, for more complex functions, `%rbp` is used in a specific pattern that facilitates debugging. It works like this:



1. The first instruction executed on function entry is `pushq %rbp`. This saves the caller's value for `%rbp` into the callee's stack. (Since `%rbp` is callee-saved, the callee must save it.)
2. The second instruction is `movq %rsp, %rbp`. This saves the current stack pointer in `%rbp` (so `%rbp` = entry `%rsp` - 8).
   This adjusted value of `%rbp` is the callee's "frame pointer." The callee will not change this value until it returns. The frame pointer provides a stable reference point for local variables and caller arguments. (Complex functions may need a stable reference point because they reserve varying amounts of space for calling different functions.)
   Note, also, that the value stored at `(%rbp)` is the *caller's* `%rbp`, and the value stored at `8(%rbp)` is the return address. This information can be used to trace backwards through callers' stack frames by functions such as debuggers.
3. The function ends with `movq %rbp, %rsp; popq %rbp; retq`, or, equivalently, `leave; retq`. This sequence restores the caller's `%rbp` and entry `%rsp` before returning.

Stack size and red zone

Functions execute fast because allocating space within a function is simply a matter of decrementing `%rsp`. This is much cheaper than a call to `malloc` or `new`! But making this work takes a lot of machinery. We'll see this in more detail later; but in brief: The operating system knows that `%rsp` points to the stack, so if a function accesses nonexistent memory near `%rsp`, the OS assumes it's for the stack and transparently allocates new memory there.

So how can a program "run out of stack"? The operating system puts a limit on each function's stack, and if `%rsp` gets too low, the program segmentation faults.

The diagram above also shows a nice feature of the x86-64 architecture, namely the **red zone**. This is a small area *above* the stack pointer (that is, at lower addresses than `%rsp`) that can be used by the currently-running function for local variables. The red zone is nice because it can be used without mucking around with the stack pointer; for small functions `push` and `pop` instructions end up taking time.

Branches

The processor typically executes instructions in sequence, incrementing `%rip` each time. Deviations from sequential instruction execution, such as function calls, are called **control flow transfers**.

Function calls aren't the only kind of control flow transfer. A *branch* instruction jumps to a new instruction without saving a return address on the stack.

Branches come in two flavors, unconditional and conditional. The `jmp` or `j` instruction executes an unconditional branch (like a `goto`). All other branch instructions are conditional: they only branch if some condition holds. That condition is represented by **condition flags** that are set as a side effect of every arithmetic operation.

Arithmetic instructions change part of the `%rflags` register as a side effect of their operation. The most often used flags are:

- **ZF** (zero flag): set iff the result was zero.
- **SF** (sign flag): set iff the most significant bit (the sign bit) of the result was one (i.e., the result was negative if considered as a signed integer).
- **CF** (carry flag): set iff the result overflowed when considered as unsigned (i.e., the result was greater than 2^W-1).
- **OF** (overflow flag): set iff the result overflowed when considered as signed (i.e., the result was greater than 2^{W-1}-1 or less than $–2^{W-1}$).

Flags are most often accessed via conditional jump or conditional move instructions. The conditional branch instructions are:

Instruction	Mnemonic	C example	Flags
j (jmp)	Jump	break;	(Unconditional)
je (jz)	Jump if equal (zero)	if (x == y)	ZF
jne (jnz)	Jump if not equal (nonzero)	if (x != y)	!ZF
jg (jnle)	Jump if greater	if (x > y), signed	!ZF && !(SF ^ OF)
jge (jnl)	Jump if greater or equal	if (x >= y), signed	!(SF ^ OF)
jl (jnge)	Jump if less	if (x < y), signed	SF ^ OF
jle (jng)	Jump if less or equal	if (x <= y), signed	(SF ^ OF) || ZF
ja (jnbe)	Jump if above	if (x > y), unsigned	!CF && !ZF
jae (jnb)	Jump if above or equal	if (x >= y), unsigned	!CF
jb (jnae)	Jump if below	if (x < y), unsigned	CF
jbe (jna)	Jump if below or equal	if (x <= y), unsigned	CF || ZF
js	Jump if sign bit	if (x < 0), signed	SF
jns	Jump if not sign bit	if (x >= 0), signed	!SF
jc	Jump if carry bit	N/A	CF
jnc	Jump if not carry bit	N/A	!CF
jo	Jump if overflow bit	N/A	OF
jno	Jump if not overflow bit	N/A	!OF

The `test` and `cmp` instructions are frequently seen before a conditional branch. These operations perform arithmetic but throw away the result, except for condition codes. `test` performs binary-and, `cmp` performs subtraction.

`cmp` is hard to grasp: remember that `subq %rax, %rbx` performs `%rbx := %rbx - %rax`—the source/destination operand is on the left. So `cmpq %rax, %rbx` evaluates `%rbx - %rax`. The sequence `cmpq %rax, %rbx; jg L` will jump to label `L` if and only if `%rbx` is greater than `%rax` (signed).

The weird-looking instruction `testq %rax, %rax`, or more generally `testq REG, SAMEREG`, is used to load the condition flags appropriately for a single register. For example, the bitwise-and of `%rax` and `%rax` is zero if and only if `%rax` is zero, so `testq %rax, %rax; je L` jumps to `L` if and only if `%rax` is zero.

You will occasionally see instructions named `setFLAG` that load the binary value of a condition (0 or 1) into an 8-bit register. For example, `setz %al` sets `%al` to 1 if the zero flag ZF is on, and 0 otherwise. There are `set` constructions for all conditions. `set` instructions are often followed by zero-extending instructions: `setz %al; movzbl %al, %eax` sets *all of* `%rax` to 1 if the zero flag ZF is on, and 0 otherwise.

Data-movement and control-flow instructions do not modify flags. Oddly, for example, `lea` does not modify flags (it counts as data movement), though `add` does (it counts as arithmetic).

Sidebar: C++ data structures

C++ compilers and data structure implementations have been designed to avoid the so-called *abstraction penalty*, which is when convenient data structures compile to more and more-expensive instructions than simple, raw memory accesses. When this works, it works quite well; for example, this:

```
long f(std::vector<int>& v) {
    long sum = 0;
    for (auto& i : v) {
        sum += i;
    }
    return sum;
}
```

compiles to this, a very tight loop similar to the C version:

```
        movq    (%rdi), %rax
        movq    8(%rdi), %rcx
        cmpq    %rcx, %rax
        je      .L4
        movq    %rax, %rdx
        addq    $4, %rax
        subq    %rax, %rcx
        andq    $-4, %rcx
        addq    %rax, %rcx
        movl    $0, %eax
.L3:
        movslq  (%rdx), %rsi
        addq    %rsi, %rax
        addq    $4, %rdx
        cmpq    %rcx, %rdx
        jne     .L3
        rep ret
.L4:
        movl    $0, %eax
        ret
```

We can also use this output to infer some aspects of `std::vector`’s implementation. It looks like:

- The first element of a `std::vector` structure is a pointer to the first element of the vector;
- The elements are stored in memory in a simple array;
- The second element of a `std::vector` structure is a pointer to *one-past-the-end* of the elements of the vector (i.e., if the vector is empty, the first and second elements of the structure have the same value).

Compiler optimizations

Argument elision

A compiler may decide to elide (or remove) certain operations setting up function call arguments, if it can decide that the registers containing these arguments will hold the correct value before the function call takes place. Let's see an example of a function disassembled function `f` in `f31.s`:

```
subq    $8, %rsp
call    _Z1gi@PLT
addq    $8, %rsp
addl    $1, %eax
ret
```

This function calls another function `g`, adds 1 to `g`'s return value, and returns that value.

It is possible that the function has the following definition in C++:

```
int f() {
    return 1 + g();
}
```

However, the actual definition of `f` in `f31.cc` is:

```
int f(x) {
    return 1 + g(x);
}
```

The compiler realizes that the argument to function `g`, which is passed via register `%rdi`, already has the right value when `g` is called, so it doesn't bother doing anything about it. This is one example of numerous optimizations a compiler can perform to reduce the size of generated code.

Inlining

A compiler may also copy the body of function to its call site, instead of doing an explicit function call, when it decides that the overhead of performing a function call outweights the overhead of doing this copy. For example, if we have a function `g` defined as `g(x) = 2 + x`, and `f` is defined as `f(x) = 1 + g(x)`, then the compiler may actually generate `f(x)` as simply `3 + x`, without inserting any `call` instructions. In assembly terms, function `g` will look like

```
leal    2(%rdi), %eax
ret
```

and `f` will simply be

```
leal    3(%rdi), %eax
ret
```

Tail call elimination

Let's look at another example in `f32.s`:

```
addl    $1, %edi
jmp _Z1gi@PLT
```

This function doesn't even contain a `ret` instruction! What is going on? Let's take a look at the actual definition of `f`, in `f32.cc`:

```
int f(int x) {
    return g(x + 1);
}
```

Note that the call to function `g` is the last operation in function `f`, and the return value of `f` is just the return value of the invocation of `g`. In this case the compiler can perform a *tail call* *elimination*: instead of calling g explicitly, it can simply jump to g and have g return to the same address that f would have returned to.

A tail call elimination may occur if a function (caller) ends with another function call (callee) and performs no cleanup once the callee returns. In this case the caller and simply jump to the callee, instead of doing an explicit call.

Loop unrolling

Before we jump into loop unrolling, let's take a small excursion into an aspect of calling conventions called caller/callee-saved registers. This will help us under the sample program in f33.s better.

Calling conventions: caller/callee-saved registers

Let's look at the function definition in f33.s:

```
    pushq   %r12
    pushq   %rbp
    pushq   %rbx
    testl   %edi, %edi
    je  .L4
    movl    %edi, %r12d
    movl    $0, %ebx
    movl    $0, %ebp
.L3:
    movl    %ebx, %edi
    call    _Z1gj@PLT
    addl    %eax, %ebp
    addl    $1, %ebx
    cmpl    %ebx, %r12d
    jne .L3
.L1:
    movl    %ebp, %eax
    popq    %rbx
    popq    %rbp
    popq    %r12
    ret
.L4:
    movl    %edi, %ebp
    jmp .L1
```

From the assembly we can tell that the backwards jump to .L3 is likely a loop. The loop index is in %ebx and the loop bound is in %r12d. Note that upon entry to the function we first moved the value %rdi to %r12d. This is necessary because in the loop f calls g, and %rdi is used to pass arguments to g, so we must move its value to a different register to used it as the loop bound (this case %r12). But there is more to this: the compiler also needs to ensure that this register's value is preserved across function calls. Calling conventions dictate that certain registers always exhibit this property, and they are called **callee-saved registers**. If a register is callee-saved, then the caller doesn't have to save its value before entering a function call.

We note that upon entry to the function, f saved a bunch of registers by pushing them to the stack: %r12, %rbp, %rbx. It is because all these registers are callee-saved registers, and f uses them during the function call. In general, the following registers in x86_64 are callee-saved:

%rbx, %r12-%r15, %rbp, %rsp (%rip)

All the other registers are **caller-saved**, which means the callee doesn't have to preserve their values. If the caller wants to reuse values in these registers across function calls, it will have to explicitly save and restore these registers. In general, the following registers in x86_64 are caller-saved:

%rax, %rcx, %rdx, %r8-%r11

Now let's get back to loop unrolling. Let us a look at the program in f34.s:

```
    testl   %edi, %edi
    je  .L7
    leal    -1(%rdi), %eax
    cmpl    $7, %eax
    jbe .L8
    pxor    %xmm0, %xmm0
    movl    %edi, %edx
    xorl    %eax, %eax
    movdqa  .LC0(%rip), %xmm1
    shrl    $2, %edx
    movdqa  .LC1(%rip), %xmm2
.L5:
    addl    $1, %eax
    paddd   %xmm1, %xmm0
    paddd   %xmm2, %xmm1
    cmpl    %edx, %eax
    jb  .L5
    movdqa  %xmm0, %xmm1
    movl    %edi, %edx
    andl    $-4, %edx
    psrldq  $8, %xmm1
    paddd   %xmm1, %xmm0
    movdqa  %xmm0, %xmm1
    cmpl    %edx, %edi
    psrldq  $4, %xmm1
    paddd   %xmm1, %xmm0
    movd    %xmm0, %eax
    je  .L10
.L3:
    leal    1(%rdx), %ecx
    addl    %edx, %eax
    cmpl    %ecx, %edi
    je  .L1
    addl    %ecx, %eax
    leal    2(%rdx), %ecx
    cmpl    %ecx, %edi
    je  .L1
    addl    %ecx, %eax
    leal    3(%rdx), %ecx
    cmpl    %ecx, %edi
    je  .L1
    addl    %ecx, %eax
    leal    4(%rdx), %ecx
    cmpl    %ecx, %edi
    je  .L1
    addl    %ecx, %eax
    leal    5(%rdx), %ecx
    cmpl    %ecx, %edi
    je  .L1
    addl    %ecx, %eax
    leal    6(%rdx), %ecx
    cmpl    %ecx, %edi
    je  .L1
    addl    %ecx, %eax
    addl    $7, %edx
    leal    (%rax,%rdx), %ecx
    cmpl    %edx, %edi
    cmovne  %ecx, %eax
    ret
.L7:
    xorl    %eax, %eax
```

```
.L1:
    rep ret
.L10:
    rep ret
.L8:
    xorl    %edx, %edx
    xorl    %eax, %eax
    jmp .L3
```

Wow this looks long and repetitive! Especially the section under label .L3! If we take a look at the original function definition in f34.cc, we will find that it's almost the same as f33.cc, except that in f34.cc we know the definition of g as well. With knowledge of what g does the compiler's optimizer decides that unrolling the loop into 7-increment batches results in faster code.

Code like this can become difficult to understand, especially when the compiler begins to use more advanced registers reserved for vector operations. We can fine-tune the optimizer to disable certain optimizations. For example, we can use the -mno-sse -fno-unroll-loops compiler options to disable the use of SSE registers and loop unrolling. The resulting code, in f35.s, for the same function definitions in f34.cc, becomes much easier to understand:

```
    testl   %edi, %edi
    je  .L4
    xorl    %edx, %edx
    xorl    %eax, %eax
.L3:
    addl    %edx, %eax
    addl    $1, %edx
    cmpl    %edx, %edi
    jne .L3
    rep ret
.L4:
    xorl    %eax, %eax
    ret
```

Note that the compiler still performed inlining to eliminate function g.

Optimizing recursive functions

Let's look at the following recursive function in f36.cc:

```
int f(int x) {
    if (x > 0) {
        return x * f(x - 1);
    } else {
        return 0;
    }
}
```

At the first glance it may seem that the function returns factorial of x. But it actually returns 0. Despite it doing a series of multiplications, in the end it always multiplies the whole result with 0, which produces 0.

When we compile this function to assembly without much optimization, we see the expensive computation occurring:

```
    movl    $0, %eax
    testl   %edi, %edi
    jg  .L8
    rep ret
.L8:
    pushq   %rbx
    movl    %edi, %ebx
    leal    -1(%rdi), %edi
    call    _Z1fi
    imull   %ebx, %eax
    popq    %rbx
    ret
```

In f37.cc there is an actual factorial function:

```
int f(int x) {
    if (x > 0) {
        return x * f(x - 1);
    } else {
        return 1;
    }
}
```

If we compile this function using level-2 optimization (-O2), we get the following assembly:

```
    testl   %edi, %edi
    movl    $1, %eax
    jle .L4
.L3:
    imull   %edi, %eax
    subl    $1, %edi
    jne .L3
    rep ret
.L4:
    rep ret
```

There is no call instructions again! The compiler has transformed the recursive function into a loop.

If we revisit our "fake" factorial function that always returns 0, and compile it with -O2, we see yet more evidence of compiler's deep understanding of our program:

```
xorl    %eax, %eax
ret
```

Optimizing arithmetic operations

The assembly code in f39.s looks like this:

```
leal    (%rdi,%rdi,2), %eax
leal    (%rdi,%rax,4), %eax
ret
```

It looks like some rather complex address computations! The first leal instruction basically loads %eax with value 3*%rdi (or %rdi + 2*%rdi). The second leal multiplies the previous result by another 4, and adds another %rdi to it. So what it actually does is 3*%rdi*4 + %rdi, or simply 13*%rdi. This is also revealed in the function name in f39.s.

The compiler choose to use leal instructions instead of an explicit multiply because the two leal instructions actually take less space.

關於 GNU Inline Assembly

DEC 10TH, 2015 10:16 PM | COMMENTS

以前稍微接觸過 GNU Inline Assembly，對於那些奇怪的符號總是覺得匪夷所思。這次找時間把他整理一下。雖然釐清了一些觀念，不過卻產生更多的疑惑，也許以後有機會看到範例會慢慢有感覺吧。

目錄

前言

我自己對於 GNU Inline Assembly 的看法。

- 編譯器 夠聰明，所以暫存器分配可以安心交給編譯器處理。也就是說語法上面要處理這塊。

- 暫存器、變數有些資訊仍然要讓編譯器知道，讓編譯器產生 object binary 遵守這樣的規則，如
  - 這個 operand 是一個暫存器
  - 這個 operand 是一塊記憶體
  - 這個 operand 是浮點常數
  - ...
- 不想讓編譯器幫你安排暫存器，而是在 Inline Assembly 指定暫存器的話，就要明確的列出來。讓編譯器知道這些暫存器有被改過資料，進而針對這些暫存器做適當的處理。

測試環境

我使用 ARMv7 為主的 Banana Pi 開發版加上 Lubuntu 14.04 作為測試環境。

```
$ lsb_release -a
No LSB modules are available.
Distributor ID:	Ubuntu
Description:	Ubuntu 14.04.3 LTS
Release:	14.04
Codename: trusty

$ dmesg
...
[	0.000000] Linux version 3.4.90 (bananapi@lemaker) (gcc version 4.6.3 (Ubuntu/Linaro 4.6.3-1ubuntu5) ) #2 SMP PREEMPT Tue Aug 5 14:11:40 CST 2014
[	0.000000] CPU: ARMv7 Processor [410fc074] revision 4 (ARMv7), cr=10c5387d
...

$ gcc -v
...
Target: arm-linux-gnueabihf
...
gcc version 4.8.4 (Ubuntu/Linaro 4.8.4-2ubuntu1~14.04)
```

語法

inline assembler 關鍵字是 `asm`，不過`__asm__`也可以使用(註)。

根據目前(Dec/2015)的 gcc 手冊，inline assembler 有分為 `basic` 和 `extended` 兩種。雖然我使用的平台是 gcc 4.8.4，而且 gcc 4.8.5 手冊(官方網站上沒有 4.8.4 手冊)並沒有提到這個部份。但是目前語法上測試的確沒有問題，但是有些說明上面卻很難驗證是否可以套用到 4.8.5 上(例如最佳化的說明、需要注意常犯的錯誤)，請自行斟酌。

以下是整理自最新的手冊說明，請自行斟酌您使用的 gcc 版本是否有符合。

Basic inline assembler

```
 [ volatile ] asm("Assembler Template");
```

以下是整理自最新(Dec/2015)的手冊說明節錄，請自行斟酌您使用的 gcc 版本是否有符合。

- basic inline assembler 預設就是 volatile
- 基本上編譯器只是把引號內的東西抄錄，所以只要組譯器支援的語法，就可以寫入 Assembler Template 內
- 和 extended inline assembler 的差異
  - extended inline assembler 只允許在函數內使用
  - 有 `naked` 屬性的函數必須使用 basic inline assembler(見註解)
  - basic inline assembler 就是把 template 內的字串作為組合語言組譯。而`%`字元在 extended inline assembler 有特別意義，然而有些組合語言如 x86 中`%`是暫存器語法的一部份。以至於`%`字元要在 extended inline assembler 中改為`%%`才是真正的意思，舉個例子`%eax`->`%%eax`
- 有要使用 C 語言的資料，使用 extended inline assembler 比較妥當
- GCC 最佳化時是有可能把你的 inline assembler 幹掉或是和你想的不一樣，請注意
- 你不可以從一個 `asm(..)`裏面跳到另外一個 `asm(..)`的 label

最簡單的廢話範例如下

```
asm("nop"); /*  啥事都不要做  */
```

在沒有使用 C 語言的資料下，就和一般的組合語言沒有差太多。 更複雜一點的例子可以看 rtenv 裏面的使用方式：

```
size_t strlen(const char *s) __attribute__ ((naked));
size_t strlen(const char *s)
{
	asm(
		"	sub	r3, r0, #1			\n"
		"strlen_loop:				\n"
		"	ldrb r2, [r3, #1]!		\n"
		"	cmp	r2, #0				\n"
		"   bne	strlen_loop		\n"
		"	sub	r0, r3, r0			\n"
		"	bx	lr					\n"
		:::
	);
}
```

要注意`__attribute__ ((naked));`是有意義的。這是為何這段範例沒有直接指名用到 C 語言函式變數名稱的關鍵點。有興趣請看這邊，請直接找字串 `naked`。

Extended inline assembler

```
 asm [volatile] ( AssemblerTemplate
                   : OutputOperands      // optional
                 [ : InputOperands       // optional
                 [ : Clobbers ]          // optional
                 ])
```

Assembler Template 基本上就是你要寫的組語加上 Inline Assembler 專用的符號。要注意的是，在編譯的過程中，你寫的 inline assembler 可能由於最佳化考慮不會被組譯。如果你確認你 inline assembler 一定要被組譯，請加上 `volatile` keyword。

Assembler 專用的符號節錄如下：

符號	說明
%%	單一%字元
%{	單一{字元
%}	單一}字元
|{	單一|字元
%=	只知道並驗證過會產生唯一的數字。用途部份看不懂，英文真是奧妙的東西啊。

AssemblerTemplate

由於前言提到的三項個人猜測，造成 inline assembler 要使用 C 語言變數時語法會出現很多令人眼花撩亂的符號。

由於編譯器提供協助分配暫存器和記憶體，也就是說需要有對應的語法指定目前指令的 operand 是什麼。GCC 有兩種方式指定，分別是

- 編號指定，從零開始編號
- Symbolic name 指定: GCC 3.1 以後支援出處

分別給個範例讓各位感受一下

編號指定，從零開始編號

這邊`%0`, `%1` 就是編號。後面 operand 可以看到就是指定變數、以及變數的限制。這邊簡單解釋一下`=`表示這是一個輸出、而 `r` 表示變數要放在暫存器中、`m` 表示變數是放在記憶體中。有興趣比對編譯出來的 binary 反組譯時的組合語言請看這邊。不過編號和指令中的 operand 似乎很隨意，我沒有看到特殊規範。只能交叉比對 assembler template 和 input/output operands 才能看出端倪。我猜更複雜的情況你還要比對反組譯出來的結果。

```
#include <stdio.h>

int main(void)
{
	int var1 = 12;
	int var2 = 10;

	asm("mov %0, %1 \n	\
		add %1, %0, $1" : "=r"(var1), "=r"(var2) : "r"(var2), "r"(var1):);
	printf("var1 = %d, var2 = %d\n", var1, var2);

	asm("ldr r5, %0 \n":				: "m"(var1): "r5");
	asm("str r4, %0"	: "=m"(var2):				: "r4");
	return 0;
}
```

Symbolic name 指定

編號的缺點就是可讀性比較差，所以 gcc 3.1 出現使用 symbolic name 的方式。至於那一個比較好，看你自己習慣。

直接把上面的範例更改一下。GCC 4.8.5 手冊上面說 symbolic name 隨便取，甚至和變數同名稱都可以，只要單一 **asm(...)**內的 **symbolic name** 不要重複就好。有興趣比對編譯出來的 binary 反組譯時的組合語言請看這邊。

```
#include <stdio.h>

int main(void)
{
        int var1 = 12;
        int var2 = 10;

        asm("mov %[my_var1], %[my_var2] \n    \
                add %[my_var3], %[my_var4], $1" :
                        [my_var1] "=r" (var1), [my_var3] "=r" (var2) :
                        [my_var2] "r"    (var2), [my_var4] "r"    (var1) :);
        printf("var1 = %d, var2 = %d\n", var1, var2);

        asm("ldr r5, %[my_var1] \n":: [my_var1] "m"(var1): "r5");
        asm("str r4, %[my_var1]": [my_var1] "=m" (var2):: "r4");
        return 0;
}
```

接下來來看每個欄位吧。

Output operands

```
[ [asmSymbolicName] ] constraint (cvariablename)
```

`[asmSymbolicName]` 是 GCC 3.1 以後支援語法，如前所述，不用 Symbolic Name 就用編號方式對應 assembler template operand。

指定結果要存在 C 語言中的那個變數。要注意的除了要設定對的資訊（constraints，下面會節錄）以外，operand 的 prefix 一定要是`=`或`+`這兩個 constraint。

隨便舉幾個範例

- `=r(var1)`：變數請寫入並放在暫存器中
- `=m(var1)`：變數請寫入並存到記憶體中

Input operands

```
[ [asmSymbolicName] ] constraint (cvariablename)
```

`[asmSymbolicName]` 是 GCC 3.1 以後支援語法，如前所述，不用 Symbolic Name 就用編號方式對應 assembler template operand。

指定要從在 C 語言中的那個變數取出資料。主要是要設定對的資訊（constraints，下面會節錄）。

- `r(var1)`：變數請放在暫存器中
- `m(var1)`：變數是在記憶體中

Clobbered registers list

先講結論，在 `asm("語法")`中明確地指定暫存器名稱的話，要在這邊列出。

現在我會習慣查單字。`Clobbered` 查英文單字會發現就是把東西用力地砸毀。所以翻譯成中文就是「砸爛的暫存器列表」。什麼是爛掉的暫存器？就是本節前面的結論囉。

另外從 Dec/2015 的 gcc 手冊還有找到下面語法，一樣請注意版本問題

符號	說明
“cc”	和狀態有關的 flag 暫存器會被修改
“memory”	這段組合語言會讀寫列出 operand 以外的記憶體內容，因此編譯器會視情況備份暫

Constraints

```
<Constraints>           ::= <Constraint Modifier> <Other Constraints> | <Other Constraints>
<Other Constraints> ::= <Simple Constraints> | <Machine Constraints>

; /*  以上 BNF 是我整理的，terminal symbol 請自行看手冊  */
```

節錄整理我~~看得懂~~感興趣的部份。

Simple Constraints

符號	說明
空白	會被忽略，排版用
m	operand 存放在記憶體中
r	operand 將被放在暫存器中
i	operand 是一個整數常數，該常數包含下面的情形(symbolic name)： `#define MAX_LINE (32)`
n	operand 是一個整數常數，只允許填入數字
E	operand 是一個浮點數常數，不清楚和`F`的差異
F	operand 是一個浮點數常數，不清楚和`E`的差異
g	operand 存在暫存器(r)或是記憶體內(m)，或是這是一個整數常數
X	不用檢查 operand

你可以使用組合技如`"rim"`，如果這樣寫的話，意思是要編譯器幫你挑一個最適合的方式處理對應於 assembler template 內的 operand。

Constraint Modifier

符號	說明
=	表示這是一個 write only 的 operand，必須為 contraint 開始字元。
+	表示這個 operand 在指令中是同時被讀寫的，必須為 contraint 開始字元。
&	該 operand 為 earlyclobber。earlyclobber 就是在 instruction 讀取該 operand 前，該 operand 會被寫入。雖然如此，到底是多久前？是和 data hazard 有關嘛？還是跟資料一致性有關？或者是和編譯器 最佳化造成非預期結果有關？ 真是一團謎完全搞不懂做啥用，也不清楚使用時機。這邊有範例， 一樣搞不懂為什麼要有+, &的 modifier
%	該 operand 可以讓編譯器 決定這個 operand 是否和後面的 operand 交換(commutative)， 完全搞不懂做啥用

ARM 專用的 Constraint

我參考的是 gcc 4.8.5 手冊(因為和測試環境的 gcc 版本最接近)，可能有版本的問題，這些我都沒有做實驗測試，請自行斟酌。

符號	說明(一般模式)
w	VFP 浮點運算
G	浮點運算的 0.0
I	8 bit 正整數
K	I contraint 的 invert (一的補數)， Wen: 不知道為什麼要扯到 I constraint?
L	I contraint 的負數 (二的補數)， Wen: 不知道為什麼要扯到 I constraint?
M	0 ~ 32 的正整數
Q	要參考的記憶體位址存放在一個暫存器內
R	operand 是一個 const pool 內的東西， 不要問我 const pool 是啥，估狗到都和 Java 有關
S	operand 目前檔案中.text 內的一個 symbol
Uv	VFP load/store 指令可存取的記憶體
Uy	iWMMXt load/store 指令可存取的記憶體
Uq	ARMv4 ldrsb 指令可存取的記憶體

完整列表在這邊，要注意的是 2015 年 12 月的手冊又多了一些新的 contstraint。請自行參考。

- gcc 4.8.5: Constraints for Particular Machines
  - 請自行參考你的硬體平台
- Dec/2015 手冊: 6.44.4.4 Constraints for Particular Machines

參考資料

- 中文
  - (BIG5)用 Open Source 工具開發軟體: 新軟體開發關念: Chapter 4. GNU Compiler Collection
    - 題外話，寫這位文件的作者個人非常佩服，但是網路上似乎關於這位作者只有這份文件。真是神祕的人物

    - Nano 雞排: Inline Assembly
- 英文
    - gcc: How to Use Inline Assembly Language in C Code
        - 手冊上寫是給 `gcc 6.0`，我目前從 GGG release 網站上看到最新版本是 5.3，怪。
        - 為什麼列出這個，因為我原本找的 gcc-4.8.5 對於 assembler template 說明沒有特別列出 gcc 支援的 inline assembler 符號。另外這份的文件結構的確比 4.8.5 清楚。
    - gcc-4.8.5: Assembler Instructions with C Expression Operands
    - gcc-4.8.5: Constraints for asm Operands
    - ARM GCC Inline Assembler Cookbook
        - 相當推荐，不論是給的說明和範例，更厲害的是關於 contraint 部份寫的比手冊清楚，一樣我沒測過就是了。
    - A Tiny Guide to GCC Inline Assembly
    - GCC-Inline-Assembly-HOWTO
    - OSDev: Inline Assembly
    - Introduction to GCC inline assembler
        - 似乎有點古老。

附錄

- C 語言標準有提到編譯器可以使用 `asm` keyword，而且沒有定義語法。有興趣可以找 `C11`、`C99`、`C89` 的標準，直接搜尋 `asm` 就可以看到了。

- `naked` 使用 basic inline assembler 和 extended inline assembler 比較

下面兩個函數，`strcmp1` 沒有任何 extended inline assembler 而 `strcmp2` 硬塞了一個下去：

```
int strcmp1(const char *a, const char *b) __attribute__ ((naked));
int strcmp1(const char *a, const char *b)
{
	asm(
		"strcmp_lop1:                            \n"
		"	ldrb	r2, [r0],#1	\n"
		"	ldrb	r3, [r1],#1	\n"
		"	cmp	r2, #1		\n"
		"	it	hi		\n"
		"	cmphi	r2, r3		\n"
		"	beq	strcmp_lop1	\n"
		"	sub	r0, r2, r3	\n"
		"	bx	lr		\n"
		:::
	);
}

int strcmp2(const char *a, const char *b) __attribute__ ((naked));
int strcmp2(const char *a, const char *b)
{
		int i;
	asm(
		"strcmp_lop2:                            \n"
		"	ldrb	r2, [r0],#1	\n"
		"	ldrb	r3, [r1],#1	\n"
		"	cmp	r2, #1		\n"
		"	it	hi		\n"
		"	cmphi	r2, r3		\n"
		"	mov	%1, $1 \n"
		"	beq	strcmp_lop2	\n"
		"	sub	r0, r2, r3	\n"
		"	bx	lr		\n"
		:"=r"(i)::
	);
}
```

我們可以比較一下下面兩個函數最後編譯出來的指令，`strcmp2` 顯然和我們預期的差很多。

```
000083f4 <strcmp1>:


int strcmp1(const char *a, const char *b) __attribute__ ((naked));
int strcmp1(const char *a, const char *b)
{
	asm(
	83f4:	f810 2b01 	ldrb.w r2, [r0], #1
	83f8:	f811 3b01 	ldrb.w r3, [r1], #1
	83fc:	2a01      	cmp	r2, #1
	83fe:	bf88      	it	hi
	8400:	429a      	cmphi	r2, r3
	8402:	d0f7      	beq.n	83f4 <strcmp1>
	8404:	eba2 0003 	sub.w	r0, r2, r3
	8408:	4770      	bx	lr
		"	beq	strcmp_lop1	\n"
		"	sub	r0, r2, r3	\n"
		"	bx	lr		\n"
		:::
	);
}
	840a:	4618      	mov	r0, r3

0000840c <strcmp2>:
		"	beq	strcmp_lop2	\n"
		"	sub	r0, r2, r3	\n"
		"	bx	lr		\n"
		:"=r"(i)::
	);
}
```

- 範例一的反組譯節錄

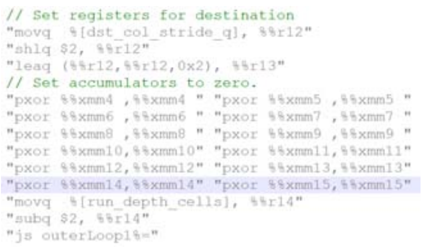
```
$ objdump -d -S asm
...
000083f4 <main>:
#include <stdio.h>

int main(void)
{
...
	int var1 = 12;
	83fa:	230c      	movs r3, #12
	83fc:	603b      	str	r3, [r7, #0]

	int var2 = 10;
	83fe:	230a      	movs r3, #10
	8400:	607b      	str	r3, [r7, #4]

	asm("mov %0, %1 \n	\
		add %1, %0, $1" : "=r"(var1), "=r"(var2) : "r"(var2), "r"(var1):);
	8402:	687b      	ldr	r3, [r7, #4]
	8404:	683a      	ldr	r2, [r7, #0]
	8406:	461a      	mov	r2, r3
	8408:	f102 0301 	add.w	r3, r2, #1
	840c:	603a      	str	r2, [r7, #0]
	840e:	607b      	str	r3, [r7, #4]
...
	asm("ldr r5, %0 \n":			: "m"(var1): "r5");
	8424:	683d      	ldr	r5, [r7, #0]

	asm("str r4, %0"	: "=m"(var2):			: "r4");
```

```
    8426:   607c            str     r4, [r7, #4]
...
}
...
```

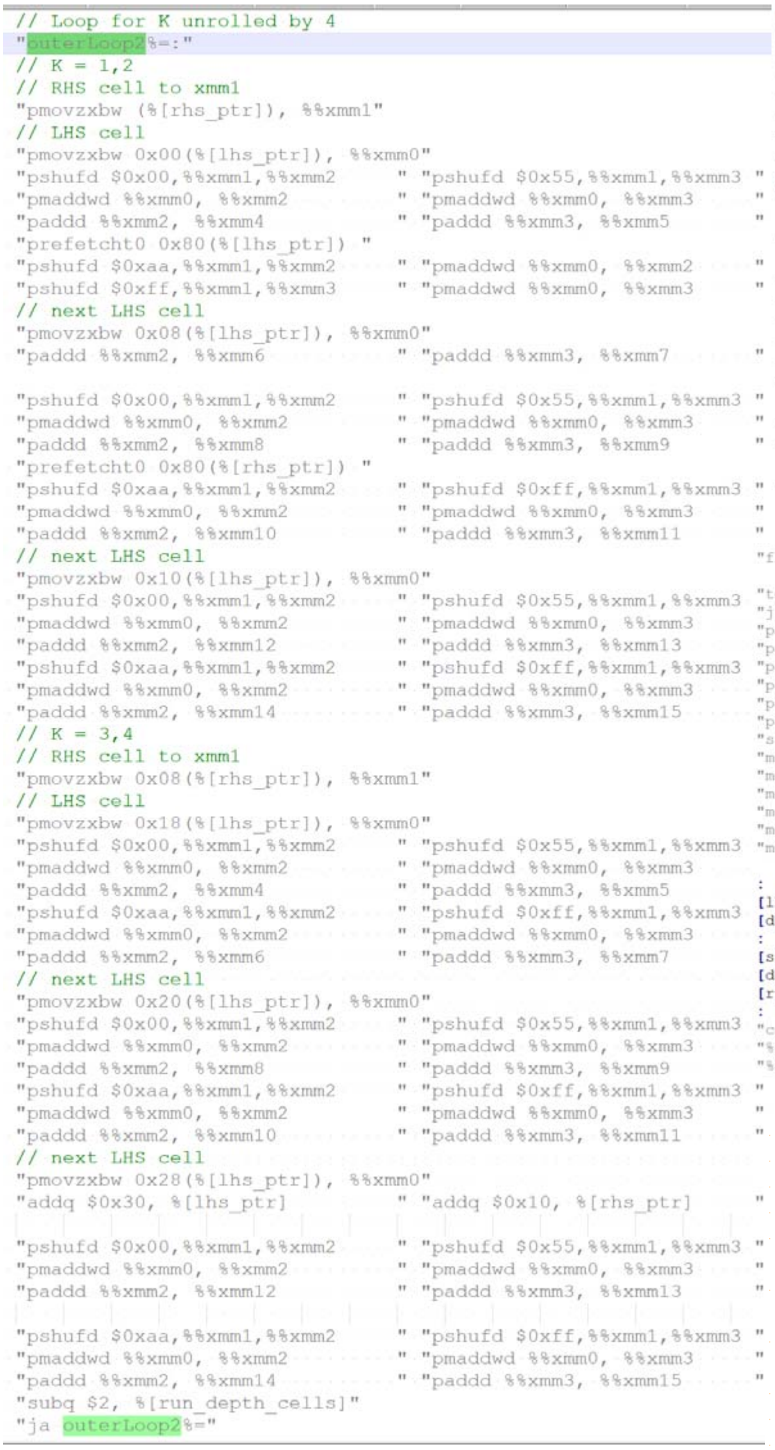
- 範例二的反組譯節錄

```
$ objdump -d -S asm
...
000083f4 <main>:
#include <stdio.h>

int main(void)
{
...
    int var1 = 12;
    83fa:   230c            movs r3, #12
    83fc:   603b            str     r3, [r7, #0]

    int var2 = 10;
    83fe:   230a            movs r3, #10
    8400:   607b            str     r3, [r7, #4]

    asm("mov %[my_var1], %[my_var2] \n     \
         add %[my_var3], %[my_var4], $1" :
              [my_var1] "=r" (var1), [my_var3] "=r" (var2) :
              [my_var2] "r"    (var2), [my_var4] "r"    (var1) :);
    8402:   687b            ldr     r3, [r7, #4]
    8404:   683a            ldr     r2, [r7, #0]
    8406:   461a            mov     r2, r3
    8408:   f102 0301       add.w   r3, r2, #1
    840c:   603a            str     r2, [r7, #0]
    840e:   607b            str     r3, [r7, #4]
...
    asm("ldr r5, %[my_var1] \n":: [my_var1] "m"(var1): "r5");
    8424:   683d            ldr     r5, [r7, #0]

    asm("str r4, %[my_var1]": [my_var1] "=m" (var2):: "r4");
    8426:   607c            str     r4, [r7, #4]
...
}
```

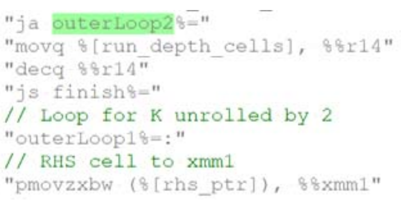
```
// Set registers for destination
"movq    %[dst_col_stride_q], %%r12" "shlq $2, %%r12" "leaq (%%r12,%%r12,0x2), %%r13"
// Set accumulators to zero.
"pxor %%xmm4 ,%%xmm4 " "pxor %%xmm5 ,%%xmm5 " "pxor %%xmm6 ,%%xmm6 " "pxor %%xmm7 ,%%xmm7 "
"pxor %%xmm8 ,%%xmm8 " "pxor %%xmm9 ,%%xmm9 " "pxor %%xmm10,%%xmm10" "pxor %%xmm11,%%xmm11"
"pxor %%xmm12,%%xmm12" "pxor %%xmm13,%%xmm13" "pxor %%xmm14,%%xmm14" "pxor %%xmm15,%%xmm15"
"movq    %[run_depth_cells], %%r14"
"subq $2, %%r14"
"js outerLoop1%="
// Loop for K unrolled by 4
"outerLoop2%=:"
// K = 1,2 // RHS cell to xmm1
"pmovzxbw (%[rhs_ptr]), %%xmm1"
// LHS cell
"pmovzxbw 0x00(%[lhs_ptr]), %%xmm0"
"pshufd $0x00, %%xmm1, %%xmm2          " "pshufd $0x55, %%xmm1, %%xmm3 "
"pmaddwd %%xmm0, %%xmm2                 " "pmaddwd %%xmm0, %%xmm3          "
"paddd %%xmm2, %%xmm4                   " "paddd      %%xmm3, %%xmm5          "
"prefetcht0 0x80(%[lhs_ptr]) "
"pshufd $0xaa, %%xmm1, %%xmm2          " "pmaddwd %%xmm0, %%xmm2          "
"pshufd $0xff, %%xmm1, %%xmm3          " "pmaddwd %%xmm0, %%xmm3          "
// next LHS cell
"pmovzxbw 0x08(%[lhs_ptr]), %%xmm0"
"paddd %%xmm2, %%xmm6                   " "paddd %%xmm3, %%xmm7             "

"pshufd $0x00, %%xmm1, %%xmm2          " "pshufd $0x55, %%xmm1, %%xmm3 "
"pmaddwd %%xmm0, %%xmm2                 " "pmaddwd %%xmm0, %%xmm3          "
"paddd %%xmm2, %%xmm8                   " "paddd %%xmm3, %%xmm9             "
"prefetcht0 0x80(%[rhs_ptr]) "
"pshufd $0xaa, %%xmm1, %%xmm2          " "pshufd $0xff, %%xmm1, %%xmm3 "
"pmaddwd %%xmm0, %%xmm2                 " "pmaddwd %%xmm0, %%xmm3          "
"paddd %%xmm2, %%xmm10                  " "paddd %%xmm3, %%xmm11            "
// next LHS cell
"pmovzxbw 0x10(%[lhs_ptr]), %%xmm0"
"pshufd $0x00, %%xmm1, %%xmm2          " "pshufd $0x55, %%xmm1, %%xmm3 "
"pmaddwd %%xmm0, %%xmm2                 " "pmaddwd %%xmm0, %%xmm3          "
"paddd %%xmm2, %%xmm12                  " "paddd %%xmm3, %%xmm13            "
"pshufd $0xaa, %%xmm1, %%xmm2          " "pshufd $0xff, %%xmm1, %%xmm3 "
"pmaddwd %%xmm0, %%xmm2                 " "pmaddwd %%xmm0, %%xmm3          "
"paddd %%xmm2, %%xmm14                  " "paddd %%xmm3, %%xmm15            "
// K = 3,4 // RHS cell to xmm1
"pmovzxbw 0x08(%[rhs_ptr]), %%xmm1"
// LHS cell
"pmovzxbw 0x18(%[lhs_ptr]), %%xmm0"
"pshufd $0x00, %%xmm1, %%xmm2          " "pshufd $0x55, %%xmm1, %%xmm3 "
"pmaddwd %%xmm0, %%xmm2                 " "pmaddwd %%xmm0, %%xmm3          "
"paddd %%xmm2, %%xmm4                   " "paddd %%xmm3, %%xmm5             "
"pshufd $0xaa, %%xmm1, %%xmm2          " "pshufd $0xff, %%xmm1, %%xmm3 "
"pmaddwd %%xmm0, %%xmm2                 " "pmaddwd %%xmm0, %%xmm3          "
"paddd %%xmm2, %%xmm6                   " "paddd %%xmm3, %%xmm7             "
// next LHS cell
"pmovzxbw 0x20(%[lhs_ptr]), %%xmm0"
"pshufd $0x00, %%xmm1, %%xmm2          " "pshufd $0x55, %%xmm1, %%xmm3 "
"pmaddwd %%xmm0, %%xmm2                 " "pmaddwd %%xmm0, %%xmm3          "
"paddd %%xmm2, %%xmm8                   " "paddd %%xmm3, %%xmm9             "
"pshufd $0xaa, %%xmm1, %%xmm2          " "pshufd $0xff, %%xmm1, %%xmm3 "
"pmaddwd %%xmm0, %%xmm2                 " "pmaddwd %%xmm0, %%xmm3          "
"paddd %%xmm2, %%xmm10                  " "paddd %%xmm3, %%xmm11            "
// next LHS cell
"pmovzxbw 0x28(%[lhs_ptr]), %%xmm0"
"addq $0x30, %[lhs_ptr]                  " "addq $0x10, %[rhs_ptr]          "

"pshufd $0x00, %%xmm1, %%xmm2          " "pshufd $0x55, %%xmm1, %%xmm3 "
"pmaddwd %%xmm0, %%xmm2                 " "pmaddwd %%xmm0, %%xmm3          "
"paddd %%xmm2, %%xmm12                  " "paddd %%xmm3, %%xmm13            "

"pshufd $0xaa, %%xmm1, %%xmm2          " "pshufd $0xff, %%xmm1, %%xmm3 "
"pmaddwd %%xmm0, %%xmm2                 " "pmaddwd %%xmm0, %%xmm3          "
"paddd %%xmm2, %%xmm14                  " "paddd %%xmm3, %%xmm15            "
"subq $2, %[run_depth_cells]"
"ja outerLoop2%="
"movq %[run_depth_cells], %%r14"
"decq %%r14"
"js finish%="
// Loop for K unrolled by 2
"outerLoop1%=:"
// RHS cell to xmm1
"pmovzxbw (%[rhs_ptr]), %%xmm1"
// LHS cell
"pmovzxbw 0x00(%[lhs_ptr]), %%xmm0"
"pshufd $0x00, %%xmm1, %%xmm2          " "pshufd $0x55, %%xmm1, %%xmm3          "
"pmaddwd %%xmm0, %%xmm2                 " "pmaddwd %%xmm0, %%xmm3                 "
"paddd %%xmm2, %%xmm4                   " "paddd %%xmm3, %%xmm5                   "
"pshufd $0xaa, %%xmm1, %%xmm2          " "pshufd $0xff, %%xmm1, %%xmm3          "
"pmaddwd %%xmm0, %%xmm2                 " "pmaddwd %%xmm0, %%xmm3                 "
"paddd %%xmm2, %%xmm6                   " "paddd %%xmm3, %%xmm7                   "
// next LHS cell
"pmovzxbw 0x08(%[lhs_ptr]), %%xmm0"
"pshufd $0x00, %%xmm1, %%xmm2          " "pshufd $0x55, %%xmm1, %%xmm3          "
"pmaddwd %%xmm0, %%xmm2                 " "pmaddwd %%xmm0, %%xmm3                 "
"paddd %%xmm2, %%xmm8                   " "paddd %%xmm3, %%xmm9                   "
"pshufd $0xaa, %%xmm1, %%xmm2          " "pshufd $0xff, %%xmm1, %%xmm3          "
"pmaddwd %%xmm0, %%xmm2                 " "pmaddwd %%xmm0, %%xmm3                 "
"paddd %%xmm2, %%xmm10                  " "paddd %%xmm3, %%xmm11                  "
// next LHS cell
"pmovzxbw 0x10(%[lhs_ptr]), %%xmm0"

"addq $0x18, %[lhs_ptr]                  " "addq $0x08, %[rhs_ptr]                  "
"pshufd $0x00, %%xmm1, %%xmm2          " "pshufd $0x55, %%xmm1, %%xmm3          "
"pmaddwd %%xmm0, %%xmm2                 " "pmaddwd %%xmm0, %%xmm3                 "
"paddd %%xmm2, %%xmm12                  " "paddd %%xmm3, %%xmm13                  "
"pshufd $0xaa, %%xmm1, %%xmm2          " "pshufd $0xff, %%xmm1, %%xmm3          "
"pmaddwd %%xmm0, %%xmm2                 " "pmaddwd %%xmm0, %%xmm3                 "
"paddd %%xmm2, %%xmm14                  " "paddd %%xmm3, %%xmm15                  "
"decq %[run_depth_cells]"
"jnz outerLoop1%="
"finish%=:"
"test %[start_depth], %[start_depth]"
"jz storeDst%="
"paddd    0x00(%[dst_ptr])                , %%xmm4 "    "paddd 0x10(%[dst_ptr])                   , %%xmm8 "
"paddd    0x20(%[dst_ptr])                , %%xmm12"    "paddd 0x00(%[dst_ptr], %%r12, 1) , %%xmm5 "
"paddd    0x10(%[dst_ptr], %%r12,1), %%xmm9 "      "paddd 0x20(%[dst_ptr], %%r12, 1) , %%xmm13"
"paddd    0x00(%[dst_ptr], %%r12,2), %%xmm6 "      "paddd 0x10(%[dst_ptr], %%r12, 2) , %%xmm10"
"paddd    0x20(%[dst_ptr], %%r12,2), %%xmm14"      "paddd 0x00(%[dst_ptr], %%r13, 1) , %%xmm7 "
"paddd    0x10(%[dst_ptr], %%r13,1), %%xmm11"      "paddd 0x20(%[dst_ptr], %%r13, 1) , %%xmm15"
"storeDst%=:"
"movdqu    %%xmm4 ,0x00(%[dst_ptr])                "    "movdqu %%xmm8 ,0x10(%[dst_ptr])                  "
"movdqu    %%xmm12,0x20(%[dst_ptr])                "    "movdqu %%xmm5 ,0x00(%[dst_ptr], %%r12,1)"
"movdqu    %%xmm9 ,0x10(%[dst_ptr], %%r12,1)"    "movdqu %%xmm13,0x20(%[dst_ptr], %%r12,1)"
"movdqu    %%xmm6 ,0x00(%[dst_ptr], %%r12,2)"    "movdqu %%xmm10,0x10(%[dst_ptr], %%r12,2)"
"movdqu    %%xmm14,0x20(%[dst_ptr], %%r12,2)"    "movdqu %%xmm7 ,0x00(%[dst_ptr], %%r13,1)"
"movdqu    %%xmm11,0x10(%[dst_ptr], %%r13,1)"    "movdqu %%xmm15,0x20(%[dst_ptr], %%r13,1)"

:    // outputs
[lhs_ptr] "+r"(lhs_ptr), [rhs_ptr] "+r"(rhs_ptr),
[dst_ptr] "+r"(dst_ptr)
:    // inputs
[start_depth] "r"(start_depth),
[dst_col_stride_q] "r"(dst_col_stride_q),
[run_depth_cells] "r"(run_depth_cells)
:    // clobbers
"cc", "memory", "%xmm0", "%xmm1", "%xmm3", "%xmm2", "%xmm4", "%xmm5",
"%xmm6", "%xmm7", "%xmm8", "%xmm9", "%xmm10", "%r12", "%r13", "%r14",
"%xmm11", "%xmm12", "%xmm13", "%xmm14", "%xmm15");
```



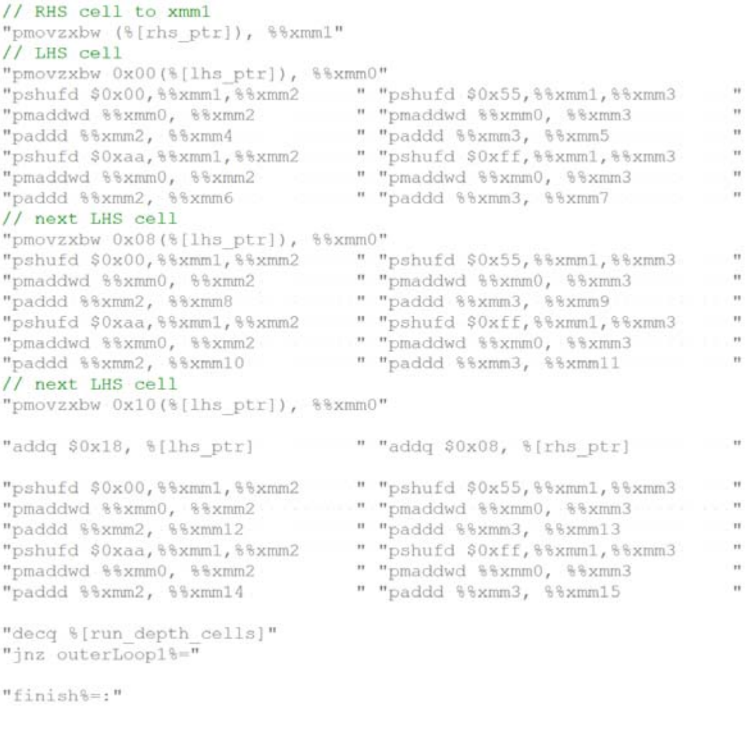
```
// Set registers for destination
"movq  %[dst_col_stride_q], %%r12"
"shlq $2, %%r12"
"leaq (%%r12,%%r12,0x2), %%r13"
// Set accumulators to zero.
"pxor %%xmm4 ,%%xmm4 " "pxor %%xmm5 ,%%xmm5 "
"pxor %%xmm6 ,%%xmm6 " "pxor %%xmm7 ,%%xmm7 "
"pxor %%xmm8 ,%%xmm8 " "pxor %%xmm9 ,%%xmm9 "
"pxor %%xmm10,%%xmm10" "pxor %%xmm11,%%xmm11"
"pxor %%xmm12,%%xmm12" "pxor %%xmm13,%%xmm13"
"pxor %%xmm14,%%xmm14" "pxor %%xmm15,%%xmm15"
"movq  %[run_depth_cells], %%r14"
"subq $2, %%r14"
"js outerLoop1%="
```



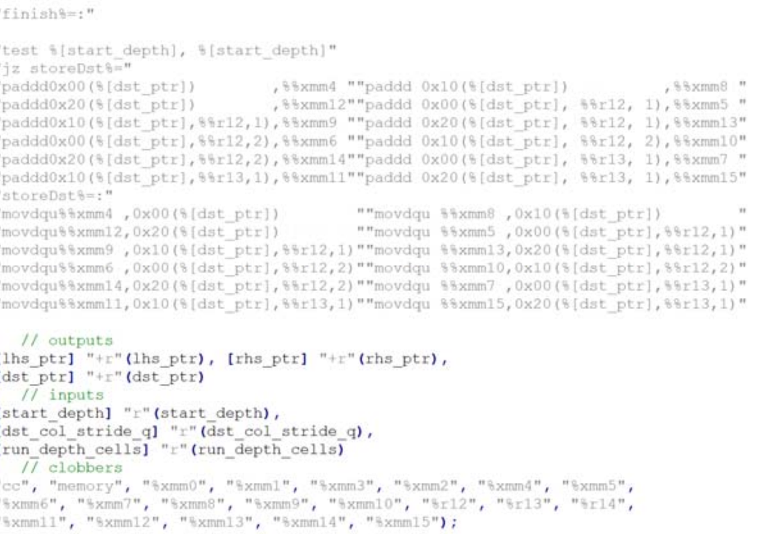
```
// Loop for K unrolled by 4
"outerLoop2%=:"
// K = 1,2
// RHS cell to xmm1
"pmovzxbw (%[rhs_ptr]), %%xmm1"
// LHS cell
"pmovzxbw 0x00(%[lhs_ptr]), %%xmm0"
"pshufd $0x00,%%xmm1,%%xmm2       " "pshufd $0x55,%%xmm1,%%xmm3 "
"pmaddwd %%xmm0, %%xmm2           " "pmaddwd %%xmm0, %%xmm3     "
"paddd %%xmm2, %%xmm4             " "paddd %%xmm3, %%xmm5       "
"prefetcht0 0x80(%[lhs_ptr]) "
"pshufd $0xaa,%%xmm1,%%xmm2       " "pmaddwd %%xmm0, %%xmm2     "
"pshufd $0xff,%%xmm1,%%xmm3       " "pmaddwd %%xmm0, %%xmm3     "
// next LHS cell
"pmovzxbw 0x08(%[lhs_ptr]), %%xmm0"
"paddd %%xmm2, %%xmm6             " "paddd %%xmm3, %%xmm7       "

"pshufd $0x00,%%xmm1,%%xmm2       " "pshufd $0x55,%%xmm1,%%xmm3 "
"pmaddwd %%xmm0, %%xmm2           " "pmaddwd %%xmm0, %%xmm3     "
"paddd %%xmm2, %%xmm8             " "paddd %%xmm3, %%xmm9       "
"prefetcht0 0x80(%[rhs_ptr]) "
"pshufd $0xaa,%%xmm1,%%xmm2       " "pshufd $0xff,%%xmm1,%%xmm3 "
"pmaddwd %%xmm0, %%xmm2           " "pmaddwd %%xmm0, %%xmm3     "
"paddd %%xmm2, %%xmm10            " "paddd %%xmm3, %%xmm11      "
// next LHS cell
"pmovzxbw 0x10(%[lhs_ptr]), %%xmm0"
"pshufd $0x00,%%xmm1,%%xmm2       " "pshufd $0x55,%%xmm1,%%xmm3 "
"pmaddwd %%xmm0, %%xmm2           " "pmaddwd %%xmm0, %%xmm3     "
"paddd %%xmm2, %%xmm12            " "paddd %%xmm3, %%xmm13      "
"pshufd $0xaa,%%xmm1,%%xmm2       " "pshufd $0xff,%%xmm1,%%xmm3 "
"pmaddwd %%xmm0, %%xmm2           " "pmaddwd %%xmm0, %%xmm3     "
"paddd %%xmm2, %%xmm14            " "paddd %%xmm3, %%xmm15      "
// K = 3,4
// RHS cell to xmm1
"pmovzxbw 0x08(%[rhs_ptr]), %%xmm1"
// LHS cell
"pmovzxbw 0x18(%[lhs_ptr]), %%xmm0"
"pshufd $0x00,%%xmm1,%%xmm2       " "pshufd $0x55,%%xmm1,%%xmm3 "
"pmaddwd %%xmm0, %%xmm2           " "pmaddwd %%xmm0, %%xmm3     "
"paddd %%xmm2, %%xmm4             " "paddd %%xmm3, %%xmm5       "
"pshufd $0xaa,%%xmm1,%%xmm2       " "pshufd $0xff,%%xmm1,%%xmm3 "
"pmaddwd %%xmm0, %%xmm2           " "pmaddwd %%xmm0, %%xmm3     "
"paddd %%xmm2, %%xmm6             " "paddd %%xmm3, %%xmm7       "
// next LHS cell
"pmovzxbw 0x20(%[lhs_ptr]), %%xmm0"
"pshufd $0x00,%%xmm1,%%xmm2       " "pshufd $0x55,%%xmm1,%%xmm3 "
"pmaddwd %%xmm0, %%xmm2           " "pmaddwd %%xmm0, %%xmm3     "
"paddd %%xmm2, %%xmm8             " "paddd %%xmm3, %%xmm9       "
"pshufd $0xaa,%%xmm1,%%xmm2       " "pshufd $0xff,%%xmm1,%%xmm3 "
"pmaddwd %%xmm0, %%xmm2           " "pmaddwd %%xmm0, %%xmm3     "
"paddd %%xmm2, %%xmm10            " "paddd %%xmm3, %%xmm11      "
// next LHS cell
"pmovzxbw 0x28(%[lhs_ptr]), %%xmm0"
"addq $0x30, %[lhs_ptr]           " "addq $0x10, %[rhs_ptr]     "

"pshufd $0x00,%%xmm1,%%xmm2       " "pshufd $0x55,%%xmm1,%%xmm3 "
"pmaddwd %%xmm0, %%xmm2           " "pmaddwd %%xmm0, %%xmm3     "
"paddd %%xmm2, %%xmm12            " "paddd %%xmm3, %%xmm13      "

"pshufd $0xaa,%%xmm1,%%xmm2       " "pshufd $0xff,%%xmm1,%%xmm3 "
"pmaddwd %%xmm0, %%xmm2           " "pmaddwd %%xmm0, %%xmm3     "
"paddd %%xmm2, %%xmm14            " "paddd %%xmm3, %%xmm15      "
"subq $2, %[run_depth_cells]"
"ja outerLoop2%="
```



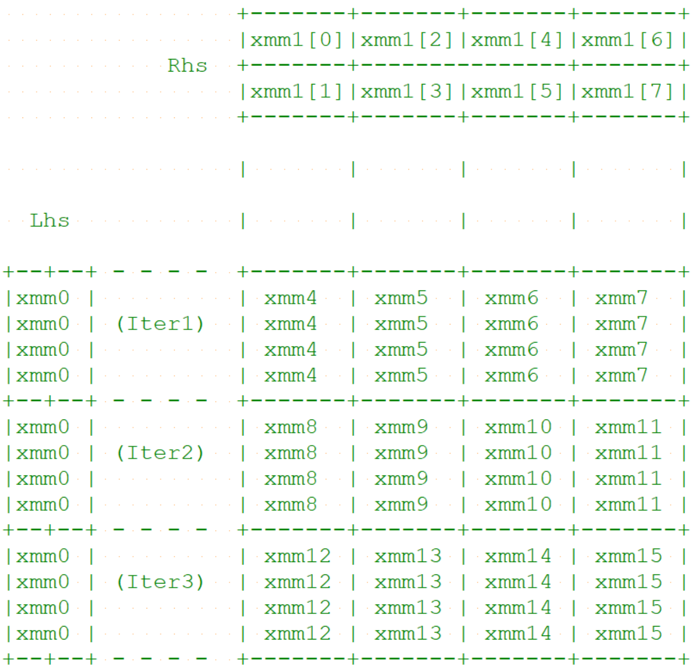
```
"ja outerLoop2%="
"movq %[run_depth_cells], %%r14"
"decq %%r14"
"js finish%="
// Loop for K unrolled by 2
"outerLoop1%=:"
// RHS cell to xmm1
"pmovzxbw (%[rhs_ptr]), %%xmm1"
```



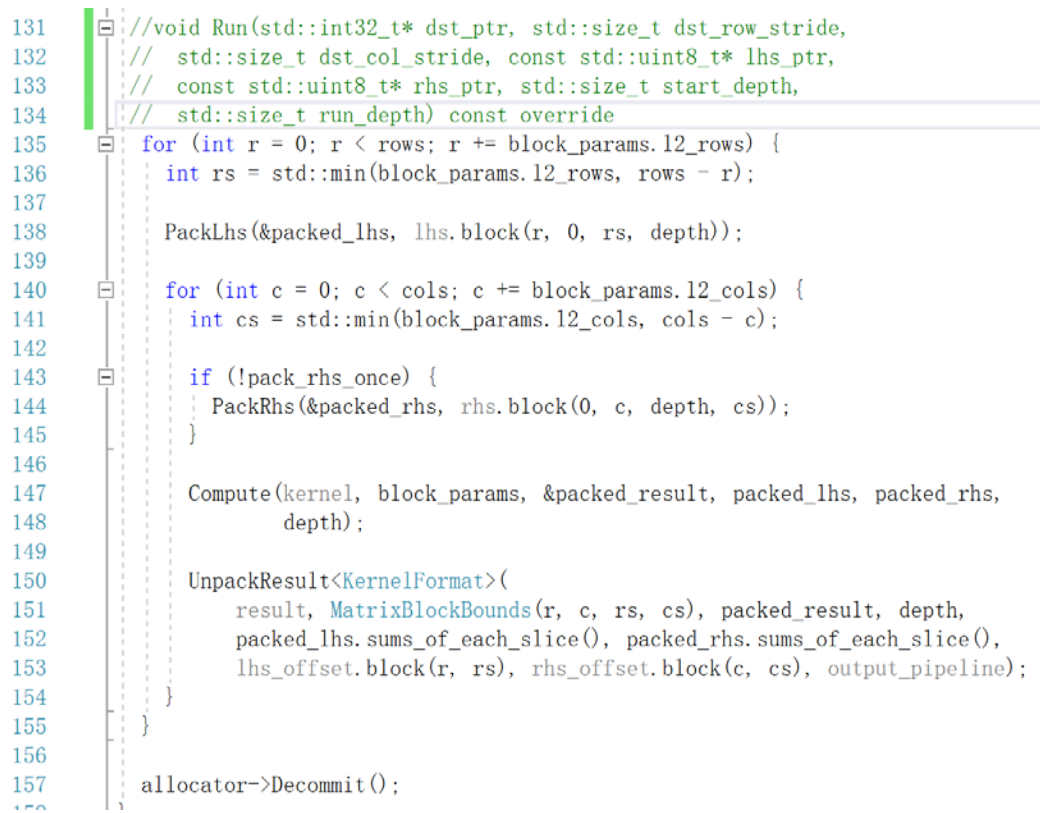
```
// RHS cell to xmm1
"pmovzxbw (%[rhs_ptr]), %%xmm1"
// LHS cell
"pmovzxbw 0x00(%[lhs_ptr]), %%xmm0"
"pshufd $0x00,%%xmm1,%%xmm2        " "pshufd $0x55,%%xmm1,%%xmm3       "
"pmaddwd %%xmm0, %%xmm2            " "pmaddwd %%xmm0, %%xmm3           "
"paddd %%xmm2, %%xmm4              " "paddd %%xmm3, %%xmm5             "
"pshufd $0xaa,%%xmm1,%%xmm2        " "pshufd $0xff,%%xmm1,%%xmm3       "
"pmaddwd %%xmm0, %%xmm2            " "pmaddwd %%xmm0, %%xmm3           "
"paddd %%xmm2, %%xmm6              " "paddd %%xmm3, %%xmm7             "
// next LHS cell
"pmovzxbw 0x08(%[lhs_ptr]), %%xmm0"
"pshufd $0x00,%%xmm1,%%xmm2        " "pshufd $0x55,%%xmm1,%%xmm3       "
"pmaddwd %%xmm0, %%xmm2            " "pmaddwd %%xmm0, %%xmm3           "
"paddd %%xmm2, %%xmm8              " "paddd %%xmm3, %%xmm9             "
"pshufd $0xaa,%%xmm1,%%xmm2        " "pshufd $0xff,%%xmm1,%%xmm3       "
"pmaddwd %%xmm0, %%xmm2            " "pmaddwd %%xmm0, %%xmm3           "
"paddd %%xmm2, %%xmm10             " "paddd %%xmm3, %%xmm11            "
// next LHS cell
"pmovzxbw 0x10(%[lhs_ptr]), %%xmm0"

"addq $0x18, %[lhs_ptr]            " "addq $0x08, %[rhs_ptr]           "

"pshufd $0x00,%%xmm1,%%xmm2        " "pshufd $0x55,%%xmm1,%%xmm3       "
"pmaddwd %%xmm0, %%xmm2            " "pmaddwd %%xmm0, %%xmm3           "
"paddd %%xmm2, %%xmm12             " "paddd %%xmm3, %%xmm13            "
"pshufd $0xaa,%%xmm1,%%xmm2        " "pshufd $0xff,%%xmm1,%%xmm3       "
"pmaddwd %%xmm0, %%xmm2            " "pmaddwd %%xmm0, %%xmm3           "
"paddd %%xmm2, %%xmm14             " "paddd %%xmm3, %%xmm15            "

"decq %[run_depth_cells]"
"jnz outerLoop1%="

"finish%=:"
```



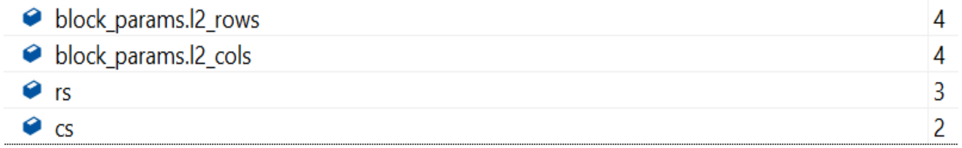
```
"finish%=:"

"test %[start_depth], %[start_depth]"
"jz storeDst%="
"paddd0x00(%[dst_ptr])          ,%%xmm4 ""paddd 0x10(%[dst_ptr])            ,%%xmm8 "
"paddd0x20(%[dst_ptr])          ,%%xmm12""paddd 0x00(%[dst_ptr], %%r12, 1),%%xmm5 "
"paddd0x10(%[dst_ptr],%%r12,1),%%xmm9 ""paddd 0x20(%[dst_ptr], %%r12, 1),%%xmm13"
"paddd0x00(%[dst_ptr],%%r12,2),%%xmm6 ""paddd 0x10(%[dst_ptr], %%r12, 2),%%xmm10"
"paddd0x20(%[dst_ptr],%%r12,2),%%xmm14""paddd 0x00(%[dst_ptr], %%r13, 1),%%xmm7 "
"paddd0x10(%[dst_ptr],%%r13,1),%%xmm11""paddd 0x20(%[dst_ptr], %%r13, 1),%%xmm15"
"storeDst%=:"
"movdqu%%xmm4 ,0x00(%[dst_ptr])         ""movdqu %%xmm8 ,0x10(%[dst_ptr])          "
"movdqu%%xmm12,0x20(%[dst_ptr])         ""movdqu %%xmm5 ,0x00(%[dst_ptr],%%r12,1)"
"movdqu%%xmm9 ,0x10(%[dst_ptr],%%r12,1)""movdqu %%xmm13,0x20(%[dst_ptr],%%r12,1)"
"movdqu%%xmm6 ,0x00(%[dst_ptr],%%r12,2)""movdqu %%xmm10,0x10(%[dst_ptr],%%r12,2)"
"movdqu%%xmm14,0x20(%[dst_ptr],%%r12,2)""movdqu %%xmm7 ,0x00(%[dst_ptr],%%r13,1)"
"movdqu%%xmm11,0x10(%[dst_ptr],%%r13,1)""movdqu %%xmm15,0x20(%[dst_ptr],%%r13,1)"

:  // outputs
[lhs_ptr] "+r"(lhs_ptr), [rhs_ptr] "+r"(rhs_ptr),
[dst_ptr] "+r"(dst_ptr)
:  // inputs
[start_depth] "r"(start_depth),
[dst_col_stride_q] "r"(dst_col_stride_q),
[run_depth_cells] "r"(run_depth_cells)
:  // clobbers
"cc", "memory", "%xmm0", "%xmm1", "%xmm3", "%xmm2", "%xmm4", "%xmm5",
"%xmm6", "%xmm7", "%xmm8", "%xmm9", "%xmm10", "%r12", "%r13", "%r14",
"%xmm11", "%xmm12", "%xmm13", "%xmm14", "%xmm15");
```



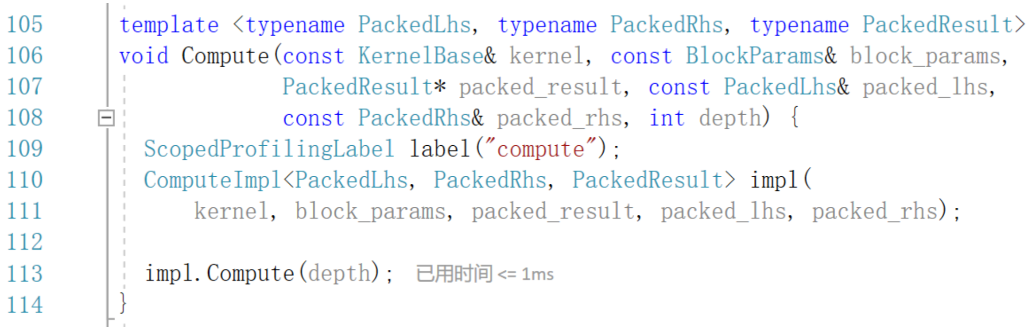
```
                            +--------+--------+--------+--------+
                            |xmm1[0] |xmm1[2] |xmm1[4] |xmm1[6] |
                       Rhs  +--------+--------+--------+--------+
                            |xmm1[1] |xmm1[3] |xmm1[5] |xmm1[7] |
                            +--------+--------+--------+--------+

                            |        |        |        |        |

    Lhs                     |        |        |        |        |

+--+--+ - - - - -           +--------+--------+--------+--------+
|xmm0 |                     | xmm4   | xmm5   | xmm6   | xmm7   |
|xmm0 |  (Iter1)            | xmm4   | xmm5   | xmm6   | xmm7   |
|xmm0 |                     | xmm4   | xmm5   | xmm6   | xmm7   |
|xmm0 |                     | xmm4   | xmm5   | xmm6   | xmm7   |
+--+--+ - - - - -           +--------+--------+--------+--------+
|xmm0 |                     | xmm8   | xmm9   | xmm10  | xmm11  |
|xmm0 |  (Iter2)            | xmm8   | xmm9   | xmm10  | xmm11  |
|xmm0 |                     | xmm8   | xmm9   | xmm10  | xmm11  |
|xmm0 |                     | xmm8   | xmm9   | xmm10  | xmm11  |
+--+--+ - - - - -           +--------+--------+--------+--------+
|xmm0 |                     | xmm12  | xmm13  | xmm14  | xmm15  |
|xmm0 |  (Iter3)            | xmm12  | xmm13  | xmm14  | xmm15  |
|xmm0 |                     | xmm12  | xmm13  | xmm14  | xmm15  |
|xmm0 |                     | xmm12  | xmm13  | xmm14  | xmm15  |
+--+--+ - - - - -           +--------+--------+--------+--------+
```



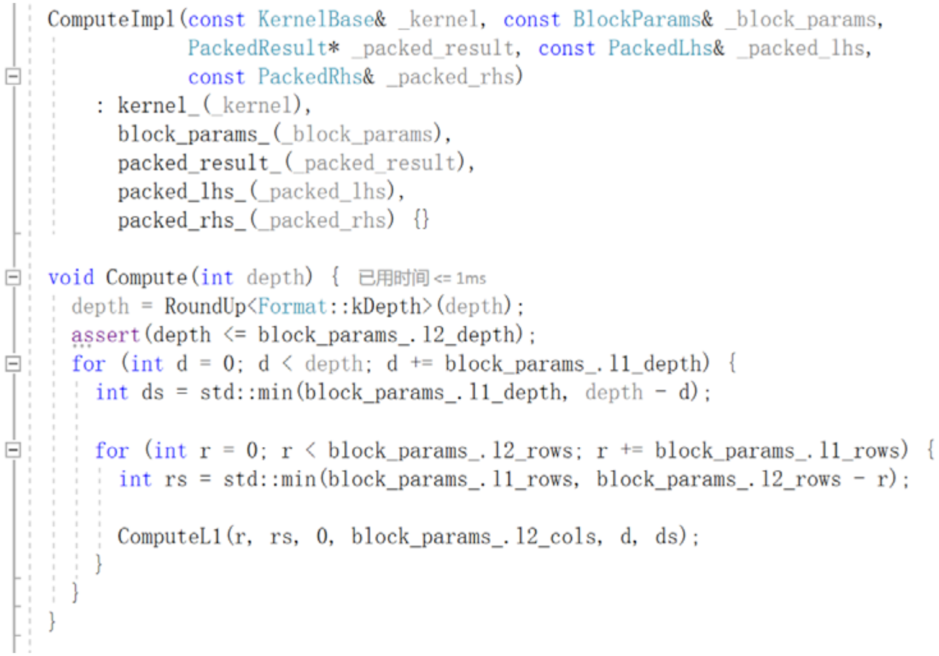
```
//void Run(std::int32_t* dst_ptr, std::size_t dst_row_stride,
//  std::size_t dst_col_stride, const std::uint8_t* lhs_ptr,
//  const std::uint8_t* rhs_ptr, std::size_t start_depth,
//  std::size_t run_depth) const override
for (int r = 0; r < rows; r += block_params.l2_rows) {
  int rs = std::min(block_params.l2_rows, rows - r);

  PackLhs(&packed_lhs, lhs.block(r, 0, rs, depth));

  for (int c = 0; c < cols; c += block_params.l2_cols) {
    int cs = std::min(block_params.l2_cols, cols - c);

    if (!pack_rhs_once) {
      PackRhs(&packed_rhs, rhs.block(0, c, depth, cs));
    }

    Compute(kernel, block_params, &packed_result, packed_lhs, packed_rhs,
            depth);

    UnpackResult<KernelFormat>(
        result, MatrixBlockBounds(r, c, rs, cs), packed_result, depth,
        packed_lhs.sums_of_each_slice(), packed_rhs.sums_of_each_slice(),
        lhs_offset.block(r, rs), rhs_offset.block(c, cs), output_pipeline);
  }
}

allocator->Decommit();
```



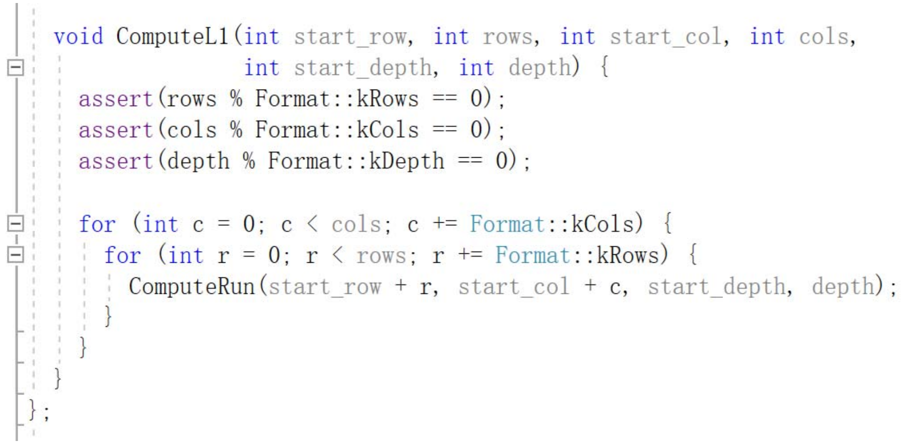
block_params.l2_rows	4
block_params.l2_cols	4
rs	3
cs	2



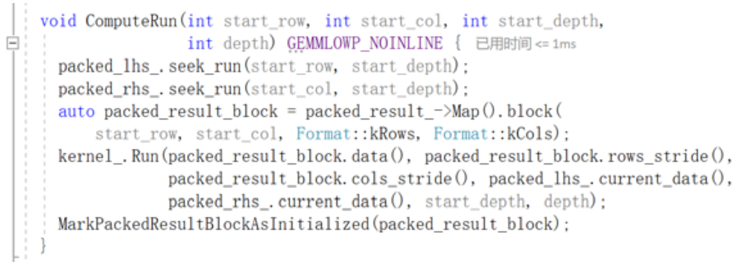
```
template <typename PackedLhs, typename PackedRhs, typename PackedResult>
void Compute(const KernelBase& kernel, const BlockParams& block_params,
             PackedResult* packed_result, const PackedLhs& packed_lhs,
             const PackedRhs& packed_rhs, int depth) {
  ScopedProfilingLabel label("compute");
  ComputeImpl<PackedLhs, PackedRhs, PackedResult> impl(
      kernel, block_params, packed_result, packed_lhs, packed_rhs);

  impl.Compute(depth);  已用时间 <= 1ms
}
```



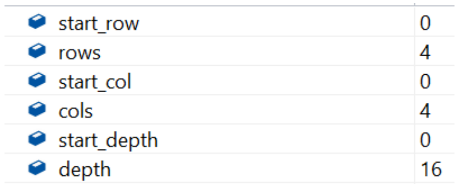
```
ComputeImpl(const KernelBase& _kernel, const BlockParams& _block_params,
            PackedResult* _packed_result, const PackedLhs& _packed_lhs,
            const PackedRhs& _packed_rhs)
    : kernel_(_kernel),
      block_params_(_block_params),
      packed_result_(_packed_result),
      packed_lhs_(_packed_lhs),
      packed_rhs_(_packed_rhs) {}

void Compute(int depth) {  已用时间 <= 1ms
  depth = RoundUp<Format::kDepth>(depth);
  assert(depth <= block_params_.l2_depth);
  for (int d = 0; d < depth; d += block_params_.l1_depth) {
    int ds = std::min(block_params_.l1_depth, depth - d);

    for (int r = 0; r < block_params_.l2_rows; r += block_params_.l1_rows) {
      int rs = std::min(block_params_.l1_rows, block_params_.l2_rows - r);

      ComputeL1(r, rs, 0, block_params_.l2_cols, d, ds);
    }
  }
}
```



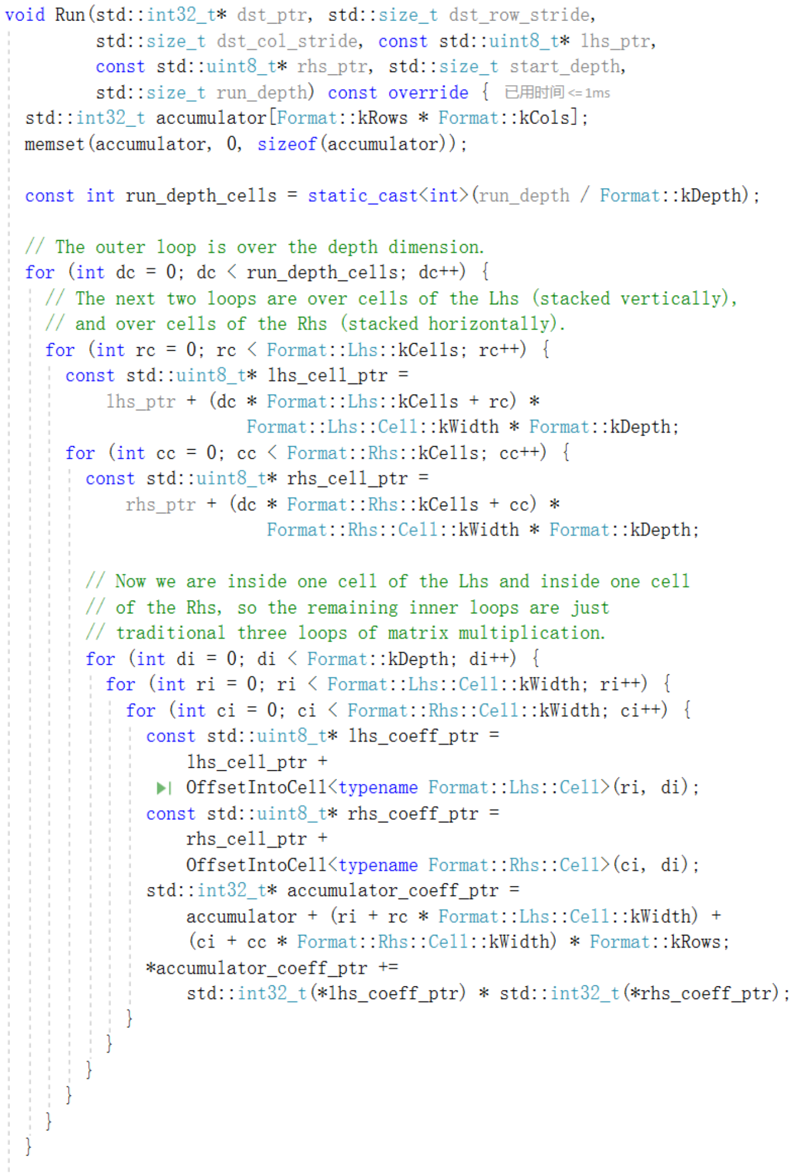
```
void ComputeL1(int start_row, int rows, int start_col, int cols,
               int start_depth, int depth) {
  assert(rows % Format::kRows == 0);
  assert(cols % Format::kCols == 0);
  assert(depth % Format::kDepth == 0);

  for (int c = 0; c < cols; c += Format::kCols) {
    for (int r = 0; r < rows; r += Format::kRows) {
      ComputeRun(start_row + r, start_col + c, start_depth, depth);
    }
  }
}
};
```



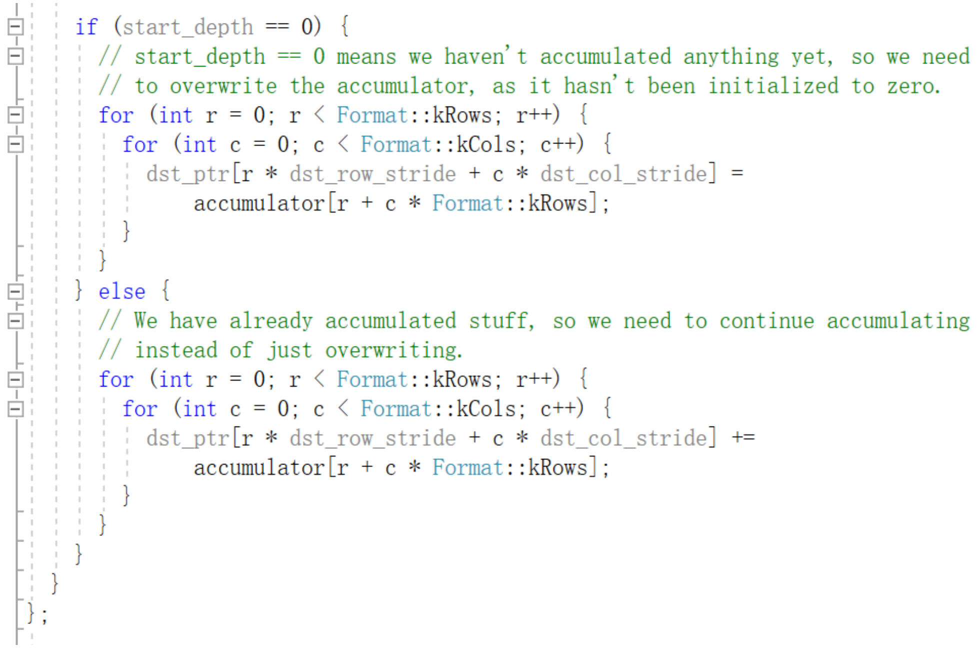
```
void ComputeRun(int start_row, int start_col, int start_depth,
                int depth) GEMMLOWP_NOINLINE {  已用时间 <= 1ms
  packed_lhs_.seek_run(start_row, start_depth);
  packed_rhs_.seek_run(start_col, start_depth);
  auto packed_result_block = packed_result_->Map().block(
      start_row, start_col, Format::kRows, Format::kCols);
  kernel_.Run(packed_result_block.data(), packed_result_block.rows_stride(),
              packed_result_block.cols_stride(), packed_lhs_.current_data(),
              packed_rhs_.current_data(), start_depth, depth);
  MarkPackedResultBlockAsInitialized(packed_result_block);
}
```



start_row	0
rows	4
start_col	0
cols	4
start_depth	0
depth	16



```
void Run(std::int32_t* dst_ptr, std::size_t dst_row_stride,
         std::size_t dst_col_stride, const std::uint8_t* lhs_ptr,
         const std::uint8_t* rhs_ptr, std::size_t start_depth,
         std::size_t run_depth) const override {  已用时间 <= 1ms
  std::int32_t accumulator[Format::kRows * Format::kCols];
  memset(accumulator, 0, sizeof(accumulator));

  const int run_depth_cells = static_cast<int>(run_depth / Format::kDepth);

  // The outer loop is over the depth dimension.
  for (int dc = 0; dc < run_depth_cells; dc++) {
    // The next two loops are over cells of the Lhs (stacked vertically),
    // and over cells of the Rhs (stacked horizontally).
    for (int rc = 0; rc < Format::Lhs::kCells; rc++) {
      const std::uint8_t* lhs_cell_ptr =
          lhs_ptr + (dc * Format::Lhs::kCells + rc) *
                        Format::Lhs::Cell::kWidth * Format::kDepth;
      for (int cc = 0; cc < Format::Rhs::kCells; cc++) {
        const std::uint8_t* rhs_cell_ptr =
            rhs_ptr + (dc * Format::Rhs::kCells + cc) *
                          Format::Rhs::Cell::kWidth * Format::kDepth;

        // Now we are inside one cell of the Lhs and inside one cell
        // of the Rhs, so the remaining inner loops are just
        // traditional three loops of matrix multiplication.
        for (int di = 0; di < Format::kDepth; di++) {
          for (int ri = 0; ri < Format::Lhs::Cell::kWidth; ri++) {
            for (int ci = 0; ci < Format::Rhs::Cell::kWidth; ci++) {
              const std::uint8_t* lhs_coeff_ptr =
                  lhs_cell_ptr +
                  OffsetIntoCell<typename Format::Lhs::Cell>(ri, di);
              const std::uint8_t* rhs_coeff_ptr =
                  rhs_cell_ptr +
                  OffsetIntoCell<typename Format::Rhs::Cell>(ci, di);
              std::int32_t* accumulator_coeff_ptr =
                  accumulator + (ri + rc * Format::Lhs::Cell::kWidth) +
                  (ci + cc * Format::Rhs::Cell::kWidth) * Format::kRows;
              *accumulator_coeff_ptr +=
                  std::int32_t(*lhs_coeff_ptr) * std::int32_t(*rhs_coeff_ptr);
            }
          }
        }
      }
    }
  }
```



```
    if (start_depth == 0) {
      // start_depth == 0 means we haven't accumulated anything yet, so we need
      // to overwrite the accumulator, as it hasn't been initialized to zero.
      for (int r = 0; r < Format::kRows; r++) {
        for (int c = 0; c < Format::kCols; c++) {
          dst_ptr[r * dst_row_stride + c * dst_col_stride] =
              accumulator[r + c * Format::kRows];
        }
      }
    } else {
      // We have already accumulated stuff, so we need to continue accumulating
      // instead of just overwriting.
      for (int r = 0; r < Format::kRows; r++) {
        for (int c = 0; c < Format::kCols; c++) {
          dst_ptr[r * dst_row_stride + c * dst_col_stride] +=
              accumulator[r + c * Format::kRows];
        }
      }
    }
  }
};
```

```
;-------------------------------------------------
;####### Ch02_01.asm
;-------------------------------------------------
; extern "C" int IntegerAddSub_(int a, int b, int c, int d);

        .code
IntegerAddSub_ proc

; Calculate a + b + c - d
        mov eax,ecx                     ;eax = a
        add eax,edx                     ;eax = a + b
        add eax,r8d                     ;eax = a + b + c
        sub eax,r9d                     ;eax = a + b + c - d

        ret                             ;return result to caller
IntegerAddSub_ endp
        end
;-------------------------------------------------
;####### Ch02_02.asm
;-------------------------------------------------
; extern "C" unsigned int IntegerLogical_(unsigned int a, unsigned int b, unsigned int c, unsigned int d);

        extern g_Val1:dword             ;external doubleword (32-bit) value

        .code
IntegerLogical_ proc

; Calculate (((a & b) | c ) ^ d) + g_Val1
        and ecx,edx                     ;ecx = a & b
        or ecx,r8d                      ;ecx = (a & b) | c
        xor ecx,r9d                     ;ecx = ((a & b) | c) ^ d
        add ecx,[g_Val1]                ;ecx = (((a & b) | c) ^ d) + g_Val1

        mov eax,ecx                     ;eax = final result
        ret                             ;return to caller
IntegerLogical_ endp
        end
;-------------------------------------------------
;####### Ch02_03.asm
;-------------------------------------------------
; extern "C" int IntegerShift_(unsigned int a, unsigned int count, unsigned int* a_shl, unsigned int* a_shr);
; Returns:      0 = error (count >= 32), 1 = success

        .code
IntegerShift_ proc
        xor eax,eax                     ;set return code in case of error
        cmp edx,31                      ;compare count against 31
        ja InvalidCount                 ;jump if count > 31

        xchg ecx,edx                    ;exchange contents of ecx & edx
        mov eax,edx                     ;eax = a
        shl eax,cl                      ;eax = a << count;
        mov [r8],eax                    ;save result

        shr edx,cl                      ;edx = a >> count
        mov [r9],edx                    ;save result

        mov eax,1                       ;set success return code

InvalidCount:
        ret                             ;return to caller

IntegerShift_ endp
        end
;-------------------------------------------------
```

```
;####### Ch02_04.asm
;-------------------------------------------------
; extern "C" int IntegerMulDiv_(int a, int b, int* prod, int* quo, int* rem);
; Returns:      0 = error (divisor equals zero), 1 = success

        .code
IntegerMulDiv_ proc

; Make sure the divisor is not zero
        mov eax,edx                     ;eax = b
        or eax,eax                      ;logical OR sets status flags
        jz InvalidDivisor               ;jump if b is zero

; Calculate product and save result
        imul eax,ecx                    ;eax = a * b
        mov [r8],eax                    ;save product

; Calculate quotient and remainder, save results
        mov r10d,edx                    ;r10d = b
        mov eax,ecx                     ;eax = a
        cdq                             ;edx:eax contains 64-bit dividend
        idiv r10d                       ;eax = quotient, edx = remainder

        mov [r9],eax                    ;save quotient
        mov rax,[rsp+40]                ;rax = 'rem'
        mov [rax],edx                   ;save remainder
        mov eax,1                       ;set success return code

InvalidDivisor:
        ret                             ;return to caller

IntegerMulDiv_ endp
        end
;-------------------------------------------------
;####### Ch02_05.asm
;-------------------------------------------------
; extern "C" int64_t IntegerMul_(int8_t a, int16_t b, int32_t c, int64_t d, int8_t e, int16_t f, int32_t g, int64_t h);

        .code
IntegerMul_ proc

; Calculate a * b * c * d
        movsx rax,cl                    ;rax = sign_extend(a)
        movsx rdx,dx                    ;rdx = sign_extend(b)
        imul rax,rdx                    ;rax = a * b
        movsxd rcx,r8d                  ;rcx = sign_extend(c)
        imul rcx,r9                     ;rcx = c * d
        imul rax,rcx                    ;rax = a * b * c * d

; Calculate e * f * g * h
        movsx rcx,byte ptr [rsp+40]     ;rcx = sign_extend(e)
        movsx rdx,word ptr [rsp+48]     ;rdx = sign_extend(f)
        imul rcx,rdx                    ;rcx = e * f
        movsxd rdx,dword ptr [rsp+56]   ;rdx = sign_extend(g)
        imul rdx,qword ptr [rsp+64]     ;rdx = g * h
        imul rcx,rdx                    ;rcx = e * f * g * h

; Compute the final product
        imul rax,rcx                    ;rax = final product

        ret
IntegerMul_ endp

; extern "C" int UnsignedIntegerDiv_(uint8_t a, uint16_t b, uint32_t c, uint64_t d, uint8_t e, uint16_t f, uint32_t g, uint64_t h, uint64_t* quo, uint64_t* rem);
```

```
UnsignedIntegerDiv_ proc

; Calculate a + b + c + d
        movzx rax,cl                    ;rax = zero_extend(a)
        movzx rdx,dx                    ;rdx = zero_extend(b)
        add rax,rdx                     ;rax = a + b
        mov r8d,r8d                     ;r8 = zero_extend(c)
        add r8,r9                       ;r8 = c + d
        add rax,r8                      ;rax = a + b + c + d
        xor rdx,rdx                     ;rdx:rax = a + b + c + d

; Calculate e + f + g + h
        movzx r8,byte ptr [rsp+40]      ;r8 = zero_extend(e)
        movzx r9,word ptr [rsp+48]      ;r9 = zero_extend(f)
        add r8,r9                       ;r8 = e + f
        mov r10d,[rsp+56]               ;r10 = zero_extend(g)
        add r10,[rsp+64]                ;r10 = g + h;
        add r8,r10                      ;r8 = e + f + g + h
        jnz DivOK                       ;jump if divisor is not zero

        xor eax,eax                     ;set error return code
        jmp done

; Calculate (a + b + c + d) / (e + f + g + h)
DivOK:  div r8                          ;unsigned divide rdx:rax / r8
        mov rcx,[rsp+72]
        mov [rcx],rax                   ;save quotient
        mov rcx,[rsp+80]
        mov [rcx],rdx                   ;save remainder

        mov eax,1                       ;set success return code

Done:   ret
UnsignedIntegerDiv_ endp
        end
;-------------------------------------------------
;####### Ch02_06.asm
;-------------------------------------------------
; Simple lookup table (.const section data is read only)
            .const
FibVals     dword 0, 1, 1, 2, 3, 5, 8, 13
            dword 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597

NumFibVals_ dword ($ - FibVals) / sizeof dword
            public NumFibVals_

; Data section (data is read/write)
            .data
FibValsSum_ dword ?         ;value to demo RIP-relative addressing
            public FibValsSum_

; extern "C" int MemoryAddressing_(int i, int* v1, int* v2, int* v3, int* v4);
; Returns:      0 = error (invalid table index), 1 = success

           .code
MemoryAddressing_ proc

; Make sure 'i' is valid
        cmp ecx,0
        jl InvalidIndex                 ;jump if i < 0
        cmp ecx,[NumFibVals_]
        jge InvalidIndex                ;jump if i >= NumFibVals_

; Sign extend i for use in address calculations
        movsxd rcx,ecx                  ;sign extend i
        mov [rsp+8],rcx                 ;save copy of i (in rcx home area)
; Example #1 - base register
```

```
        mov r11,offset FibVals          ;r11 = FibVals
        shl rcx,2                       ;rcx = i * 4
        add r11,rcx                     ;r11 = FibVals + i * 4
        mov eax,[r11]                   ;eax = FibVals[i]
        mov [rdx],eax                   ;save to v1

; Example #2 - base register + index register
        mov r11,offset FibVals          ;r11 = FibVals
        mov rcx,[rsp+8]                 ;rcx = i
        shl rcx,2                       ;rcx = i * 4
        mov eax,[r11+rcx]               ;eax = FibVals[i]
        mov [r8],eax                    ;save to v2

; Example #3 - base register + index register * scale factor
        mov r11,offset FibVals          ;r11 = FibVals
        mov rcx,[rsp+8]                 ;rcx = i
        mov eax,[r11+rcx*4]             ;eax = FibVals[i]
        mov [r9],eax                    ;save to v3

; Example #4 - base register + index register * scale factor + disp
        mov r11,offset FibVals-42       ;r11 = FibVals - 42
        mov rcx,[rsp+8]                 ;rcx = i
        mov eax,[r11+rcx*4+42]          ;eax = FibVals[i]
        mov r10,[rsp+40]                ;r10 = ptr to v4
        mov [r10],eax                   ;save to v4

; Example #5 - RIP relative
        add [FibValsSum_],eax           ;update sum

        mov eax,1                       ;set success return code
        ret

InvalidIndex:
        xor eax,eax                     ;set error return code
        ret

MemoryAddressing_ endp
        end
;-------------------------------------------------
;####### Ch02_07.asm
;-------------------------------------------------
; extern "C" int SignedMinA_(int a, int b, int c);
; Returns:      min(a, b, c)
        .code
SignedMinA_ proc
        mov eax,ecx
        cmp eax,edx                     ;compare a and b
        jle @F                          ;jump if a <= b
        mov eax,edx                     ;eax = b

@@:     cmp eax,r8d                     ;compare min(a, b) and c
        jle @F
        mov eax,r8d                     ;eax = min(a, b, c)
@@:     ret
SignedMinA_ endp

; extern "C" int SignedMaxA_(int a, int b, int c);
; Returns:      max(a, b, c)
SignedMaxA_ proc
        mov eax,ecx
        cmp eax,edx                     ;compare a and b
        jge @F                          ;jump if a >= b
        mov eax,edx                     ;eax = b

@@:     cmp eax,r8d                     ;compare max(a, b) and c
        jge @F
        mov eax,r8d                     ;eax = max(a, b, c)
```

```
@@:     ret
SignedMaxA_ endp

; extern "C" int SignedMinB_(int a, int b, int c);
; Returns:      min(a, b, c)
SignedMinB_ proc
        cmp ecx,edx
        cmovg ecx,edx                   ;ecx = min(a, b)
        cmp ecx,r8d
        cmovg ecx,r8d                   ;ecx = min(a, b, c)
        mov eax,ecx
        ret
SignedMinB_ endp

; extern "C" int SignedMaxB_(int a, int b, int c);
; Returns:      max(a, b, c)
SignedMaxB_ proc
        cmp ecx,edx
        cmovl ecx,edx                   ;ecx = max(a, b)
        cmp ecx,r8d
        cmovl ecx,r8d                   ;ecx = max(a, b, c)
        mov eax,ecx
        ret
SignedMaxB_ endp
        end
;-------------------------------------------------
;####### Ch03_01.asm
;-------------------------------------------------
; extern "C" int CalcArraySum_(const int* x, int n)
; Returns:      Sum of elements in array x

        .code
CalcArraySum_ proc

; Initialize sum to zero
        xor eax,eax                     ;sum = 0

; Make sure 'n' is greater than zero
        cmp edx,0
        jle InvalidCount                ;jump if n <= 0

; Sum the elements of the array
@@:     add eax,[rcx]                   ;add next element to total (sum += *x)
        add rcx,4                       ;set pointer to next element (x++)
        dec edx                         ;adjust counter (n -= 1)
        jnz @B                          ;repeat if not done

InvalidCount:
        ret

CalcArraySum_ endp
        end
;-------------------------------------------------
;####### Ch03_02.asm
;-------------------------------------------------
; extern "C" long long CalcArrayValues_(long long* y, const int* x, int a, short b, int n);
; Calculation:  y[i] = x[i] * a + b
; Returns:      Sum of the elements in array y.

        .code
CalcArrayValues_ proc frame

; Function prolog
        push rsi                        ;save volatile register rsi
        .pushreg rsi
        push rdi                        ;save volatile register rdi
        .pushreg rdi
        .endprolog

; Initialize sum to zero and make sure 'n' is valid
        xor rax,rax                     ;sum = 0
        mov r11d,[rsp+56]               ;r11d = n
        cmp r11d,0
        jle InvalidCount                ;jump if n <= 0

; Initialize source and destination pointers
        mov rsi,rdx                     ;rsi = ptr to array x
        mov rdi,rcx                     ;rdi = ptr to array y

; Load expression constants and array index
        movsxd r8,r8d                   ;r8 = a (sign extended)
        movsx r9,r9w                    ;r9 = b (sign extended)
        xor edx,edx                     ;edx = array index i

; Repeat until done
@@:     movsxd rcx,dword ptr [rsi+rdx*4]    ;rcx = x[i] (sign extended)
        imul rcx,r8                     ;rcx = x[i] * a
        add rcx,r9                      ;rcx = x[i] * a + b
        mov qword ptr [rdi+rdx*8],rcx   ;y[i] = rcx

        add rax,rcx                     ;update running sum

        inc edx                         ;edx = i + i
        cmp edx,r11d                    ;is i >= n?
        jl @B                           ;jump if i < n

InvalidCount:

; Function epilog
        pop rdi                         ;restore caller's rdi
        pop rsi                         ;restore caller's rsi
        ret
CalcArrayValues_ endp
        end
;-------------------------------------------------
;####### Ch03_03.asm
;-------------------------------------------------
; void CalcMatrixSquares_(int* y, const int* x, int nrows, int ncols);
; Calculates:     y[i][j] = x[j][i] * x[j][i]

        .code
CalcMatrixSquares_ proc frame

; Function prolog
        push rsi                            ;save caller's rsi
        .pushreg rsi
        push rdi                            ;save caller's rdi
        .pushreg rdi
        .endprolog

; Make sure nrows and ncols are valid
        cmp r8d,0
        jle InvalidCount                    ;jump if nrows <= 0
        cmp r9d,0
        jle InvalidCount                    ;jump if ncols <= 0

; Initialize pointers to source and destination arrays
        mov rsi,rdx                         ;rsi = x
        mov rdi,rcx                         ;rdi = y
        xor rcx,rcx                         ;rcx = i
        movsxd r8,r8d                       ;r8 = nrows sign extended
        movsxd r9,r9d                       ;r9 = ncols sign extended

; Perform the required calculations
```

```
Loop1:
        xor rdx,rdx                         ;rdx = j
Loop2:
        mov rax,rdx                         ;rax = j
        imul rax,r9                         ;rax = j * ncols
        add rax,rcx                         ;rax = j * ncols + i
        mov r10d,dword ptr [rsi+rax*4]      ;r10d = x[j][i]
        imul r10d,r10d                      ;r10d = x[j][i] * x[j][i]

        mov rax,rcx                         ;rax = i
        imul rax,r9                         ;rax = i * ncols
        add rax,rdx                         ;rax = i * ncols + j;
        mov dword ptr [rdi+rax*4],r10d      ;y[i][j] = r10d

        inc rdx                             ;j += 1
        cmp rdx,r9
        jl Loop2                            ;jump if j < ncols

        inc rcx                             ;i += 1
        cmp rcx,r8
        jl Loop1                            ;jump if i < nrows

InvalidCount:

; Function epilog
        pop rdi                             ;restore caller's rdi
        pop rsi                             ;restore caller's rsi
        ret

CalcMatrixSquares_ endp
        end
;-------------------------------------------------
;####### Ch03_04.asm
;-------------------------------------------------
; extern "C" int CalcMatrixRowColSums_(int* row_sums, int* col_sums, const int* x, int nrows,
int ncols)
; Returns:      0 = nrows <= 0 or ncols <= 0, 1 = success

        .code
CalcMatrixRowColSums_ proc frame

; Function prolog
        push rbx                        ;save caller's rbx
        .pushreg rbx
        push rsi                        ;save caller's rsi
        .pushreg rsi
        push rdi                        ;save caller's rdi
        .pushreg rdi
        .endprolog

; Make sure nrows and ncols are valid
        xor eax,eax                     ;set error return code

        cmp r9d,0
        jle InvalidArg                  ;jump if nrows <= 0

        mov r10d,[rsp+64]               ;r10d = ncols
        cmp r10d,0

        jle InvalidArg                  ;jump if ncols <= 0

; Initialize elements of col_sums array to zero
        mov rbx,rcx                     ;temp save of row_sums
        mov rdi,rdx                     ;rdi = col_sums
        mov ecx,r10d                    ;rcx = ncols
        xor eax,eax                     ;eax = fill value
        rep stosd                       ;fill array with zeros
```

```
; The code below uses the following registers:
;   rcx = row_sums          rdx = col_sums
;   r9d = nrows             r10d = ncols
;   eax = i                 ebx = j
;   edi = i * ncols         esi = i * ncols + j
;   r8 = x                  r11d = x[i][j]

; Initialize outer loop variables.
        mov rcx,rbx                     ;rcx = row_sums
        xor eax,eax                     ;i = 0

Lp1:    mov dword ptr [rcx+rax*4],0     ;row_sums[i] = 0
        xor ebx,ebx                     ;j = 0
        mov edi,eax                     ;edi = i
        imul edi,r10d                   ;edi = i * ncols

; Inner loop
Lp2:    mov esi,edi                     ;esi = i * ncols
        add esi,ebx                     ;esi = i * ncols + j
        mov r11d,[r8+rsi*4]             ;r11d = x[i * ncols + j]
        add [rcx+rax*4],r11d            ;row_sums[i] += x[i * ncols + j]
        add [rdx+rbx*4],r11d            ;col_sums[j] += x[i * ncols + j]

; Is the inner loop finished?
        inc ebx                         ;j += 1
        cmp ebx,r10d
        jl Lp2                          ;jump if j < ncols

; Is the outer loop finished?
        inc eax                         ;i += 1
        cmp eax,r9d
        jl Lp1                          ;jump if i < nrows

        mov eax,1                       ;set success return code

; Function epilog
InvalidArg:
        pop rdi                         ;restore NV registers and return
        pop rsi
        pop rbx
        ret
CalcMatrixRowColSums_ endp
        end
;-------------------------------------------------
;####### Ch03_05.asm
;-------------------------------------------------
TestStruct struct
Val8    byte ?
Pad8    byte ?
Val16   word ?
Val32   dword ?
Val64   qword ?
TestStruct ends

; extern "C" int64_t CalcTestStructSum_(const TestStruct* ts);
; Returns:      Sum of structure's values as a 64-bit integer.

        .code
CalcTestStructSum_ proc

; Compute ts->Val8 + ts->Val16, note sign extension to 32-bits
        movsx eax,byte ptr [rcx+TestStruct.Val8]
        movsx edx,word ptr [rcx+TestStruct.Val16]
        add eax,edx

; Sign extend previous result to 64 bits
```

```
        movsxd rax,eax

; Add ts->Val32 to sum
        movsxd rdx,[rcx+TestStruct.Val32]
        add rax,rdx

; Add ts->Val64 to sum
        add rax,[rcx+TestStruct.Val64]
        ret

CalcTestStructSum_ endp
        end
;-------------------------------------------------
;####### Ch03_06.asm
;-------------------------------------------------
; extern "C" unsigned long long CountChars_(const char* s, char c);
; Description:  This function counts the number of occurrences
;               of a character in a string.
; Returns:      Number of occurrences found.

        .code
CountChars_ proc frame

; Save non-volatile registers
        push rsi                        ;save caller's rsi
        .pushreg rsi
        .endprolog

; Load parameters and initialize count registers
        mov rsi,rcx                     ;rsi = s
        mov cl,dl                       ;cl = c
        xor edx,edx                     ;rdx = Number of occurrences
        xor r8d,r8d                     ;r8 = 0 (required for add below)
; Repeat loop until the entire string has been scanned
@@:     lodsb                           ;load next char into register al
        or al,al                        ;test for end-of-string
        jz @F                           ;jump if end-of-string found
        cmp al,cl                       ;test current char
        sete r8b                        ;r8b = 1 if match, 0 otherwise
        add rdx,r8                      ;update occurrence count
        jmp @B

@@:     mov rax,rdx                    ;rax = number of occurrences

; Restore non-volatile registers and return
        pop rsi
        ret
CountChars_ endp
        end
;-------------------------------------------------
;####### Ch03_07.asm
;-------------------------------------------------
; extern "C" size_t ConcatStrings_(char* des, size_t des_size, const char* const* src, size_t
src_n);
; Returns:      -1      Invalid 'des_size'
;               n >= 0  Length of concatenated string

        .code
ConcatStrings_ proc frame

; Save non-volatile registers
        push rbx
        .pushreg rbx
        push rsi
        .pushreg rsi
        push rdi
        .pushreg rdi
```

```
        .endprolog

; Make sure des_size and src_n are valid
        mov rax,-1                      ;set error code

        test rdx,rdx                    ;test des_size
        jz InvalidArg                   ;jump if des_size is 0

        test r9,r9                      ;test src_n
        jz InvalidArg                   ;jump if src_n is 0

; Registers used processing loop below
;   rbx = des               rdx = des_size
;   r8 = src                r9 = src_n
;   r10 = des_index         r11 = i
;   rcx = string length
;   rsi, rdi = pointers for scasb & movsb instructions

; Perform required initializations
        xor r10,r10                     ;des_index = 0
        xor r11,r11                     ;i = 0
        mov rbx,rcx                     ;rbx = des
        mov byte ptr [rbx],0            ;*des = '¥0'

; Repeat loop until concatenation is finished
Loop1:  mov rax,r8                      ;rax = 'src'
        mov rdi,[rax+r11*8]             ;rdi = src[i]
        mov rsi,rdi                     ;rsi = src[i]

; Compute length of s[i]
        xor eax,eax
        mov rcx,-1
        repne scasb                     ;find '¥0'
        not rcx
        dec rcx                         ;rcx = len(src[i])
; Compute des_index + src_len
        mov rax,r10                     ;rax = des_index
        add rax,rcx                     ;des_index + len(src[i])
        cmp rax,rdx                     ;is des_index + src_len >= des_size?
        jge Done                        ;jump if des is too small

; Update des_index
        mov rax,r10                     ;des_index_old = des_index
        add r10,rcx                     ;des_index += len(src[i])
; Copy src[i] to &des[des_index] (rsi already contains src[i])
        inc rcx                         ;rcx = len(src[i]) + 1
        lea rdi,[rbx+rax]               ;rdi = &des[des_index_old]
        rep movsb                       ;perform string move

; Update i and repeat if not done
        inc r11                         ;i += 1
        cmp r11,r9
        jl Loop1                        ;jump if i < src_n

; Return length of concatenated string

Done:   mov rax,r10                    ;rax = des_index (final length)
; Restore non-volatile registers and return

InvalidArg:
        pop rdi
        pop rsi
        pop rbx
        ret
ConcatStrings_ endp
        end
;-------------------------------------------------
```

```
;####### Ch03_08.asm
;-------------------------------------------------
; extern "C" long long CompareArrays_(const int* x, const int* y, long long n)
; Returns       -1          Value of 'n' is invalid
;               0 <= i < n  Index of first non-matching element
;               n           All elements match

        .code
CompareArrays_ proc frame

; Save non-volatile registers
        push rsi
        .pushreg rsi
        push rdi
        .pushreg rdi
        .endprolog

; Load arguments and validate 'n'
        mov rax,-1                      ;rax = return code for invalid n
        test r8,r8
        jle @F                          ;jump if n <= 0

; Compare the arrays for equality
        mov rsi,rcx                     ;rsi = x
        mov rdi,rdx                     ;rdi = y
        mov rcx,r8                      ;rcx = n
        mov rax,r8                      ;rax = n
        repe cmpsd
        je @F                           ;arrays are equal

; Calculate index of first non-match
        sub rax,rcx                     ;rax = index of mismatch + 1
        dec rax                         ;rax = index of mismatch

; Restore non-volatile registers and return
@@:     pop rdi
        pop rsi
        ret
CompareArrays_ endp
        end
;-------------------------------------------------
;####### Ch03_09.asm
;-------------------------------------------------
; extern "C" int ReverseArray_(int* y, const int* x, int n);
; Returns       0 = invalid n, 1 = success

        .code
ReverseArray_ proc frame

; Save non-volatile registers
        push rsi
        .pushreg rsi
        push rdi
        .pushreg rdi
        .endprolog

; Make sure n is valid
        xor eax,eax                     ;error return code
        test r8d,r8d                    ;is n <= 0?
        jle InvalidArg                  ;jump if n <= 0

; Initialize registers for reversal operation
        mov rsi,rdx                     ;rsi = x
        mov rdi,rcx                     ;rdi = y
        mov ecx,r8d                     ;rcx = n
        lea rsi,[rsi+rcx*4-4]           ;rsi = &x[n - 1]

; Save caller's RFLAGS.DF, then set RFLAGS.DF to 1
        pushfq                          ;save caller's RFLAGS.DF
        std                             ;RFLAGS.DF = 1

; Repeat loop until array reversal is complete
@@:     lodsd                           ;eax = *x--
        mov [rdi],eax                   ;*y = eax
        add rdi,4                       ;y++
        dec rcx                         ;n--
        jnz @B

; Restore caller's RFLAGS.DF and set return code
        popfq                           ;restore caller's RFLAGS.DF
        mov eax,1                       ;set success return code

; Restore non-volatile registers and return
InvalidArg:
        pop rdi
        pop rsi
        ret
ReverseArray_ endp
        end
;-------------------------------------------------
;####### Ch05_01.asm
;-------------------------------------------------
        .const
r4_ScaleFtoC   real4 0.55555556            ; 5 / 9
r4_ScaleCtoF   real4 1.8                   ; 9 / 5
r4_32p0        real4 32.0

; extern "C" float ConvertFtoC_(float deg_f)
; Returns: xmm0[31:0] = temperature in Celsius.

        .code
ConvertFtoC_ proc
        vmovss xmm1,[r4_32p0]           ;xmm1 = 32
        vsubss xmm2,xmm0,xmm1           ;xmm2 = f - 32

        vmovss xmm1,[r4_ScaleFtoC]      ;xmm1 = 5 / 9
        vmulss xmm0,xmm2,xmm1           ;xmm0 = (f - 32) * 5 / 9
        ret
ConvertFtoC_ endp

; extern "C" float CtoF_(float deg_c)
; Returns: xmm0[31:0] = temperature in Fahrenheit.

ConvertCtoF_ proc
        vmulss xmm0,xmm0,[r4_ScaleCtoF] ;xmm0 = c * 9 / 5
        vaddss xmm0,xmm0,[r4_32p0]      ;xmm0 = c * 9 / 5 + 32
        ret
ConvertCtoF_ endp
        end
;-------------------------------------------------
;####### Ch05_02.asm
;-------------------------------------------------
        .const
r8_PI   real8 3.14159265358979323846
r8_4p0  real8 4.0
r8_3p0  real8 3.0

; extern "C" void CalcSphereAreaVolume_(double r, double* sa, double* vol);

        .code
CalcSphereAreaVolume_ proc

; Calculate surface area = 4 * PI * r * r
        vmulsd xmm1,xmm0,xmm0           ;xmm1 = r * r
        vmulsd xmm2,xmm1,[r8_PI]        ;xmm2 = r * r * PI
        vmulsd xmm3,xmm2,[r8_4p0]       ;xmm3 = r * r * PI * 4

; Calculate volume = sa * r / 3
        vmulsd xmm4,xmm3,xmm0           ;xmm4 = r * r * r * PI * 4
        vdivsd xmm5,xmm4,[r8_3p0]       ;xmm5 = r * r * r * PI * 4 / 3

; Save results
        vmovsd real8 ptr [rdx],xmm3     ;save surface area
        vmovsd real8 ptr [r8],xmm5      ;save volume
        ret
CalcSphereAreaVolume_ endp
        end
;-------------------------------------------------
;####### Ch05_03.asm
;-------------------------------------------------
; extern "C" double CalcDistance_(double x1, double y1, double z1, double x2, double y2, double
z2)
        .code
CalcDistance_ proc
; Load arguments from stack
        vmovsd xmm4,real8 ptr [rsp+40]  ;xmm4 = y2
        vmovsd xmm5,real8 ptr [rsp+48]  ;xmm5 = z2

; Calculate squares of coordinate distances
        vsubsd xmm0,xmm3,xmm0           ;xmm0 = x2 - x1
        vmulsd xmm0,xmm0,xmm0           ;xmm0 = (x2 - x1) * (x2 - x1)
        vsubsd xmm1,xmm4,xmm1           ;xmm1 = y2 - y1
        vmulsd xmm1,xmm1,xmm1           ;xmm1 = (y2 - y1) * (y2 - y1)

        vsubsd xmm2,xmm5,xmm2           ;xmm2 = z2 - z1
        vmulsd xmm2,xmm2,xmm2           ;xmm2 = (z2 - z1) * (z2 - z1)
; Calculate final distance
        vaddsd xmm3,xmm0,xmm1
        vaddsd xmm4,xmm2,xmm3           ;xmm4 = sum of squares
        vsqrtsd xmm0,xmm0,xmm4          ;xmm0 = final distance value
        ret
CalcDistance_ endp
        end
;-------------------------------------------------
;####### Ch05_04.asm
;-------------------------------------------------
; extern "C" void CompareVCOMISS_(float a, float b, bool* results);


        .code
CompareVCOMISS_ proc

; Set result flags based on compare status
        vcomiss xmm0,xmm1
        setp byte ptr [r8]              ;RFLAGS.PF = 1 if unordered
        jnp @F
        xor al,al
        mov byte ptr [r8+1],al          ;Use default result values
        mov byte ptr [r8+2],al
        mov byte ptr [r8+3],al
        mov byte ptr [r8+4],al
        mov byte ptr [r8+5],al
        mov byte ptr [r8+6],al
        jmp Done

@@:     setb byte ptr [r8+1]            ;set byte if a < b
        setbe byte ptr [r8+2]           ;set byte if a <= b
        sete byte ptr [r8+3]            ;set byte if a == b
        setne byte ptr [r8+4]           ;set byte if a != b
        seta byte ptr [r8+5]            ;set byte if a > b
        setae byte ptr [r8+6]           ;set byte if a >= b
```

```
Done:   ret
CompareVCOMISS_ endp

; extern "C" void CompareVCOMISD_(double a, double b, bool* results);

CompareVCOMISD_ proc

; Set result flags based on compare status
        vcomisd xmm0,xmm1
        setp byte ptr [r8]              ;RFLAGS.PF = 1 if unordered
        jnp @F
        xor al,al
        mov byte ptr [r8+1],al          ;Use default result values
        mov byte ptr [r8+2],al
        mov byte ptr [r8+3],al
        mov byte ptr [r8+4],al
        mov byte ptr [r8+5],al
        mov byte ptr [r8+6],al
        jmp Done

@@:     setb byte ptr [r8+1]            ;set byte if a < b
        setbe byte ptr [r8+2]           ;set byte if a <= b
        sete byte ptr [r8+3]            ;set byte if a == b
        setne byte ptr [r8+4]           ;set byte if a != b
        seta byte ptr [r8+5]            ;set byte if a > b
        setae byte ptr [r8+6]           ;set byte if a >= b

Done:   ret
CompareVCOMISD_ endp
        end
;-------------------------------------------------
;####### Ch05_05.asm
        include <cmpequ.asmh>

; extern "C" void CompareVCMPSD_(double a, double b, bool* results)
        .code
CompareVCMPSD_ proc

; Perform compare for equality
        vcmpsd xmm2,xmm0,xmm1,CMP_EQ    ;perform compare operation
        vmovq rax,xmm2                  ;rax = compare result (all 1s or 0s)
        and al,1                        ;mask out unneeded bits
        mov byte ptr [r8],al            ;save result as C++ bool

; Perform compare for inequality
        vcmpsd xmm2,xmm0,xmm1,CMP_NEQ
        vmovq rax,xmm2
        and al,1
        mov byte ptr [r8+1],al

; Perform compare for less than
        vcmpsd xmm2,xmm0,xmm1,CMP_LT
        vmovq rax,xmm2
        and al,1
        mov byte ptr [r8+2],al

; Perform compare for less than or equal
        vcmpsd xmm2,xmm0,xmm1,CMP_LE
        vmovq rax,xmm2
        and al,1
        mov byte ptr [r8+3],al

; Perform compare for greater than
        vcmpsd xmm2,xmm0,xmm1,CMP_GT
        vmovq rax,xmm2
        and al,1
        mov byte ptr [r8+4],al
```

```
; Perform compare for greater than or equal
        vcmpsd xmm2,xmm0,xmm1,CMP_GE
        vmovq rax,xmm2
        and al,1
        mov byte ptr [r8+5],al

; Perform compare for ordered
        vcmpsd xmm2,xmm0,xmm1,CMP_ORD
        vmovq rax,xmm2
        and al,1
        mov byte ptr [r8+6],al

; Perform compare for unordered
        vcmpsd xmm2,xmm0,xmm1,CMP_UNORD
        vmovq rax,xmm2
        and al,1
        mov byte ptr [r8+7],al

        ret
CompareVCMPSD_ endp
        end
;-------------------------------------------------
;####### Ch05_06.asm
;-------------------------------------------------
MxcsrRcMask equ 9fffh                   ;bit pattern for MXCSR.RC
MxcsrRcShift equ 13                     ;shift count for MXCSR.RC

; extern "C" RoundingMode GetMxcsrRoundingMode_(void);
; Description:  The following function obtains the current
;               floating-point rounding mode from MXCSR.RC.
; Returns:      Current MXCSR.RC rounding mode.

        .code
GetMxcsrRoundingMode_ proc
        vstmxcsr dword ptr [rsp+8]      ;save mxcsr register
        mov eax,[rsp+8]
        shr eax,MxcsrRcShift            ;eax[1:0] = MXCSR.RC bits
        and eax,3                       ;masked out unwanted bits
        ret
GetMxcsrRoundingMode_ endp

;extern "C" void SetMxcsrRoundingMode_(RoundingMode rm);
; Description:  The following function updates the rounding mode
;               value in MXCSR.RC.

SetMxcsrRoundingMode_ proc
        and ecx,3                       ;masked out unwanted bits
        shl ecx,MxcsrRcShift            ;ecx[14:13] = rm

        vstmxcsr dword ptr [rsp+8]      ;save current MXCSR
        mov eax,[rsp+8]
        and eax,MxcsrRcMask             ;masked out old MXCSR.RC bits
        or eax,ecx                      ;insert new MXCSR.RC bits
        mov [rsp+8],eax
        vldmxcsr dword ptr [rsp+8]      ;load updated MXCSR
        ret
SetMxcsrRoundingMode_ endp

; extern "C" bool ConvertScalar_(Uval* des, const Uval* src, CvtOp cvt_op)
; Note:         This function requires linker option /LARGEADDRESSAWARE:NO
;               to be explicitly set.

ConvertScalar_ proc

; Make sure cvt_op is valid, then jump to target conversion code
        mov eax,r8d                     ;eax = CvtOp
```

```
        cmp eax,CvtOpTableCount
        jae BadCvtOp                    ;jump if cvt_op is invalid
        jmp [CvtOpTable+rax*8]          ;jump to specified conversion

; Conversions between int32_t and float/double

I32_F32:
        mov eax,[rdx]                   ;load integer value
        vcvtsi2ss xmm0,xmm0,eax         ;convert to float
        vmovss real4 ptr [rcx],xmm0     ;save result
        mov eax,1
        ret

F32_I32:
        vmovss xmm0,real4 ptr [rdx]     ;load float value
        vcvtss2si eax,xmm0              ;convert to integer
        mov [rcx],eax                   ;save result
        mov eax,1
        ret

I32_F64:
        mov eax,[rdx]                   ;load integer value
        vcvtsi2sd xmm0,xmm0,eax         ;convert to double
        vmovsd real8 ptr [rcx],xmm0     ;save result
        mov eax,1
        ret

F64_I32:
        vmovsd xmm0,real8 ptr [rdx]     ;load double value
        vcvtsd2si eax,xmm0              ;convert to integer
        mov [rcx],eax                   ;save result
        mov eax,1
        ret

; Conversions between int64_t and float/double

I64_F32:
        mov rax,[rdx]                   ;load integer value
        vcvtsi2ss xmm0,xmm0,rax         ;convert to float
        vmovss real4 ptr [rcx],xmm0     ;save result
        mov eax,1
        ret

F32_I64:
        vmovss xmm0,real4 ptr [rdx]     ;load float value
        vcvtss2si rax,xmm0              ;convert to integer
        mov [rcx],rax                   ;save result
        mov eax,1
        ret

I64_F64:
        mov rax,[rdx]                   ;load integer value
        vcvtsi2sd xmm0,xmm0,rax         ;convert to double
        vmovsd real8 ptr [rcx],xmm0     ;save result
        mov eax,1
        ret

F64_I64:
        vmovsd xmm0,real8 ptr [rdx]     ;load double value
        vcvtsd2si rax,xmm0              ;convert to integer
        mov [rcx],rax                   ;save result
        mov eax,1
        ret

; Conversions between float and double

F32_F64:
        vmovss xmm0,real4 ptr [rdx]     ;load float value
        vcvtss2sd xmm1,xmm1,xmm0        ;convert to double
        vmovsd real8 ptr [rcx],xmm1     ;save result
        mov eax,1
        ret

F64_F32:
        vmovsd xmm0,real8 ptr [rdx]     ;load double value
        vcvtsd2ss xmm1,xmm1,xmm0        ;convert to float
        vmovss real4 ptr [rcx],xmm1     ;save result
        mov eax,1
        ret

BadCvtOp:
        xor eax,eax                     ;set error return code
        ret

; The order of values in following table must match the enum CvtOp
; that's defined in the .cpp file.

        align 8
CvtOpTable equ $
        qword I32_F32, F32_I32
        qword I32_F64, F64_I32
        qword I64_F32, F32_I64
        qword I64_F64, F64_I64
        qword F32_F64, F64_F32
CvtOpTableCount equ ($ - CvtOpTable) / size qword

ConvertScalar_ endp
        end
;-------------------------------------------------
;####### Ch05_07.asm
;-------------------------------------------------
; extern "C" bool CalcMeanStdev(double* mean, double* stdev, const double* a, int n);
; Returns:      0 = invalid n, 1 = valid n

        .code
CalcMeanStdev_  proc

; Make sure 'n' is valid
        xor eax,eax                     ;set error return code (also i = 0)
        cmp r9d,2
        jl InvalidArg                   ;jump if n < 2

; Compute sample mean
        vxorpd xmm0,xmm0,xmm0           ;sum = 0.0

@@:     vaddsd xmm0,xmm0,real8 ptr [r8+rax*8]   ;sum += x[i]
        inc eax                            ;i += 1
        cmp eax,r9d
        jl @B                              ;jump if i < n

        vcvtsi2sd xmm1,xmm1,r9d         ;convert n to DPFP
        vdivsd xmm3,xmm0,xmm1           ;xmm3 = mean (sum / n)
        vmovsd real8 ptr [rcx],xmm3     ;save mean

; Compute sample stdev
        xor eax,eax                     ;i = 0
        vxorpd xmm0,xmm0,xmm0           ;sum2 = 0.0

@@:     vmovsd xmm1,real8 ptr [r8+rax*8]    ;xmm1 = x[i]
        vsubsd xmm2,xmm1,xmm3           ;xmm2 = x[i] - mean
        vmulsd xmm2,xmm2,xmm2           ;xmm2 = (x[i] - mean) ** 2
        vaddsd xmm0,xmm0,xmm2           ;sum2 += (x[i] - mean) ** 2
        inc eax                         ;i += 1
        cmp eax,r9d
```

```
        jl @B                           ;jump if i < n

        dec r9d                         ;r9d = n - 1
        vcvtsi2sd xmm1,xmm1,r9d         ;convert n - 1 to DPFP
        vdivsd xmm0,xmm0,xmm1           ;xmm0 = sum2 / (n - 1)
        vsqrtsd xmm0,xmm0,xmm0          ;xmm0 = stdev
        vmovsd real8 ptr [rdx],xmm0     ;save stdev

        mov eax,1                       ;set success return code

InvalidArg:
        ret
CalcMeanStdev_ endp
        end
;-------------------------------------------------
;####### Ch05_08.asm
;-------------------------------------------------
; void CalcMatrixSquaresF32_(float* y, const float* x, float offset, int nrows, int ncols);
; Calculates:     y[i][j] = x[j][i] * x[j][i] + offset

        .code
CalcMatrixSquaresF32_ proc frame

; Function prolog
        push rsi                            ;save caller's rsi
        .pushreg rsi
        push rdi                            ;save caller's rdi
        .pushreg rdi
        .endprolog

; Make sure nrows and ncols are valid
        movsxd r9,r9d                       ;r9 = nrows
        test r9,r9
        jle InvalidCount                    ;jump if nrows <= 0

        movsxd r10,dword ptr [rsp+56]       ;r10 = ncols
        test r10,r10
        jle InvalidCount                    ;jump if ncols <= 0

; Initialize pointers to source and destination arrays
        mov rsi,rdx                         ;rsi = x
        mov rdi,rcx                         ;rdi = y
        xor rcx,rcx                         ;rcx = i

; Perform the required calculations
Loop1:  xor rdx,rdx                         ;rdx = j

Loop2:  mov rax,rdx                         ;rax = j
        imul rax,r10                        ;rax = j * ncols
        add rax,rcx                         ;rax = j * ncols + i
        vmovss xmm0,real4 ptr [rsi+rax*4]   ;xmm0 = x[j][i]
        vmulss xmm1,xmm0,xmm0               ;xmm1 = x[j][i] * x[j][i]
        vaddss xmm3,xmm1,xmm2               ;xmm2 = x[j][i] * x[j][i] + offset

        mov rax,rcx                         ;rax = i
        imul rax,r10                        ;rax = i * ncols
        add rax,rdx                         ;rax = i * ncols + j;
        vmovss real4 ptr [rdi+rax*4],xmm3   ;y[i][j] = x[j][i] * x[j][i] + offset

        inc rdx                             ;j += 1
        cmp rdx,r10
        jl Loop2                            ;jump if j < ncols

        inc rcx                             ;i += 1
        cmp rcx,r9
        jl Loop1                            ;jump if i < nrows
```

```
InvalidCount:

; Function epilog
        pop rdi                                 ;restore caller's rdi
        pop rsi                                 ;restore caller's rsi
        ret

CalcMatrixSquaresF32_ endp
        end
;-------------------------------------------------
;####### Ch05_09.asm
;-------------------------------------------------
; extern "C" Int64 Cc1_(int8_t a, int16_t b, int32_t c, int64_t d, int8_t e, int16_t f, int32_t g, int64_t h);

        .code
Cc1_    proc frame

; Function prolog
        push rbp                            ;save caller's rbp register
        .pushreg rbp

        sub rsp,16                          ;allocate local stack space
        .allocstack 16

        mov rbp,rsp                         ;set frame pointer
        .setframe rbp,0

RBP_RA = 24                                 ;offset from rbp to return addr
        .endprolog                          ;mark end of prolog

; Save argument registers to home area (optional)
        mov [rbp+RBP_RA+8],rcx
        mov [rbp+RBP_RA+16],rdx
        mov [rbp+RBP_RA+24],r8
        mov [rbp+RBP_RA+32],r9

; Sum the argument values a, b, c, and d
        movsx rcx,cl                        ;rcx = a
        movsx rdx,dx                        ;rdx = b
        movsxd r8,r8d                       ;r8 = c;
        add rcx,rdx                         ;rcx = a + b
        add r8,r9                           ;r8 = c + d
        add r8,rcx                          ;r8 = a + b + c + d
        mov [rbp],r8                        ;save a + b + c + d

; Sum the argument values e, f, g, and h
        movsx rcx,byte ptr [rbp+RBP_RA+40]  ;rcx = e
        movsx rdx,word ptr [rbp+RBP_RA+48]  ;rdx = f
        movsxd r8,dword ptr [rbp+RBP_RA+56] ;r8 = g
        add rcx,rdx                         ;rcx = e + f
        add r8,qword ptr [rbp+RBP_RA+64]    ;r8 = g + h
        add r8,rcx                          ;r8 = e + f + g + h

; Compute the final sum
        mov rax,[rbp]                       ;rax = a + b + c + d
        add rax,r8                          ;rax = final sum

; Function epilog
        add rsp,16                          ;release local stack space
        pop rbp                             ;restore caller's rbp register
        ret

Cc1_ endp
        end
;-------------------------------------------------
;####### Ch05_10.asm
;-------------------------------------------------
```

```
; extern "C" void Cc2_(const int64_t* a, const int64_t* b, int32_t n, int64_t* sum_a, int64_t*
sum_b, int64_t* prod_a, int64_t* prod_b)
; Named expressions for constant values:
; NUM_PUSHREG   = number of prolog non-volatile register pushes
; STK_LOCAL1    = size in bytes of STK_LOCAL1 area (see figure in text)
; STK_LOCAL2    = size in bytes of STK_LOCAL2 area (see figure in text)
; STK_PAD       = extra bytes (0 or 8) needed to 16-byte align RSP
; STK_TOTAL     = total size in bytes of local stack
; RBP_RA        = number of bytes between RBP and ret addr on stack

NUM_PUSHREG     = 4
STK_LOCAL1      = 32
STK_LOCAL2      = 16
STK_PAD         = ((NUM_PUSHREG AND 1) XOR 1) * 8
STK_TOTAL       = STK_LOCAL1 + STK_LOCAL2 + STK_PAD
RBP_RA          = NUM_PUSHREG * 8 + STK_LOCAL1 + STK_PAD

        .const
TestVal db 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15

        .code
Cc2_    proc frame

; Save non-volatile GP registers on the stack
        push rbp
        .pushreg rbp
        push rbx
        .pushreg rbx
        push r12
        .pushreg r12
        push r13
        .pushreg r13

; Allocate local stack space and set frame pointer
        sub rsp,STK_TOTAL                   ;allocate local stack space
        .allocstack STK_TOTAL

        lea rbp,[rsp+STK_LOCAL2]            ;set frame pointer
        .setframe rbp,STK_LOCAL2

        .endprolog                          ;end of prolog

; Initialize local variables on the stack (demonstration only)
        vmovdqu xmm5, xmmword ptr [TestVal]
        vmovdqa xmmword ptr [rbp-16],xmm5   ;save xmm5 to LocalVar2A/2B
        mov qword ptr [rbp],0aah            ;save 0xaa to LocalVar1A
        mov qword ptr [rbp+8],0bbh          ;save 0xbb to LocalVar1B
        mov qword ptr [rbp+16],0cch         ;save 0xcc to LocalVar1C
        mov qword ptr [rbp+24],0ddh         ;save 0xdd to LocalVar1D

; Save argument values to home area (optional)
        mov qword ptr [rbp+RBP_RA+8],rcx
        mov qword ptr [rbp+RBP_RA+16],rdx
        mov qword ptr [rbp+RBP_RA+24],r8
        mov qword ptr [rbp+RBP_RA+32],r9

; Perform required initializations for processing loop
        test r8d,r8d                        ;is n <= 0?
        jle Error                           ;jump if n <= 0

        xor rbx,rbx                         ;rbx = current element offset
        xor r10,r10                         ;r10 = sum_a
        xor r11,r11                         ;r11 = sum_b
        mov r12,1                           ;r12 = prod_a
        mov r13,1                           ;r13 = prod_b

; Compute the array sums and products
```

```
@@:     mov rax,[rcx+rbx]               ;rax = a[i]
        add r10,rax                     ;update sum_a
        imul r12,rax                    ;update prod_a
        mov rax,[rdx+rbx]               ;rax = b[i]
        add r11,rax                     ;update sum_b
        imul r13,rax                    ;update prod_b

        add rbx,8                       ;set ebx to next element
        dec r8d                         ;adjust count
        jnz @B                          ;repeat until done

; Save the final results
        mov [r9],r10                    ;save sum_a
        mov rax,[rbp+RBP_RA+40]         ;rax = ptr to sum_b
        mov [rax],r11                   ;save sum_b
        mov rax,[rbp+RBP_RA+48]         ;rax = ptr to prod_a
        mov [rax],r12                   ;save prod_a
        mov rax,[rbp+RBP_RA+56]         ;rax = ptr to prod_b
        mov [rax],r13                   ;save prod_b
        mov eax,1                       ;set return code to true

; Function epilog
Done:   lea rsp,[rbp+STK_LOCAL1+STK_PAD]    ;restore rsp
        pop r13                         ;restore non-volatile GP registers
        pop r12
        pop rbx
        pop rbp
        ret

Error:  xor eax,eax                     ;set return code to false
        jmp Done
Cc2_    endp
        end
;-------------------------------------------------
;####### Ch05_11.asm
;-------------------------------------------------
; extern "C" bool Cc3_(const double* r, const double* h, int n, double* sa_cone, double* vol_cone)
; Named expressions for constant values
; NUM_PUSHREG   = number of prolog non-volatile register pushes
; STK_LOCAL1    = size in bytes of STK_LOCAL1 area (see figure in text)
; STK_LOCAL2    = size in bytes of STK_LOCAL2 area (see figure in text)
; STK_PAD       = extra bytes (0 or 8) needed to 16-byte align RSP
; STK_TOTAL     = total size in bytes of local stack
; RBP_RA        = number of bytes between RBP and ret addr on stack

NUM_PUSHREG     = 7
STK_LOCAL1      = 16
STK_LOCAL2      = 64
STK_PAD         = ((NUM_PUSHREG AND 1) XOR 1) * 8
STK_TOTAL       = STK_LOCAL1 + STK_LOCAL2 + STK_PAD
RBP_RA          = NUM_PUSHREG * 8 + STK_LOCAL1 + STK_PAD

            .const
r8_3p0      real8 3.0
r8_pi       real8 3.14159265358979323846

        .code
Cc3_    proc frame

; Save non-volatile registers on the stack.
        push rbp
        .pushreg rbp
        push rbx
        .pushreg rbx
        push rsi
        .pushreg rsi
        push r12
        .pushreg r12
        push r13
        .pushreg r13
        push r14
        .pushreg r14
        push r15
        .pushreg r15

; Allocate local stack space and initialize frame pointer
        sub rsp,STK_TOTAL               ;allocate local stack space
        .allocstack STK_TOTAL
        lea rbp,[rsp+STK_LOCAL2]        ;rbp = stack frame pointer
        .setframe rbp,STK_LOCAL2

; Save non-volatile registers XMM12 - XMM15.  Note that STK_LOCAL2 must
; be greater than or equal to the number of XMM register saves times 16.
        vmovdqa xmmword ptr [rbp-STK_LOCAL2+48],xmm12
       .savexmm128 xmm12,48
        vmovdqa xmmword ptr [rbp-STK_LOCAL2+32],xmm13
       .savexmm128 xmm13,32
        vmovdqa xmmword ptr [rbp-STK_LOCAL2+16],xmm14
       .savexmm128 xmm14,16
        vmovdqa xmmword ptr [rbp-STK_LOCAL2],xmm15
       .savexmm128 xmm15,0
        .endprolog

; Access local variables on the stack (demonstration only)
        mov qword ptr [rbp],-1          ;LocalVar1A = -1
        mov qword ptr [rbp+8],-2        ;LocalVar1B = -2

; Initialize the processing loop variables. Note that many of the
; register initializations below are performed merely to illustrate
; use of the non-volatile GP and XMM registers.
        mov esi,r8d                     ;esi = n
        test esi,esi                    ;is n > 0?
        jg @F                           ;jump if n > 0

        xor eax,eax                     ;set error return code
        jmp done

@@:     xor rbx,rbx                     ;rbx = array element offset
        mov r12,rcx                     ;r12 = ptr to r
        mov r13,rdx                     ;r13 = ptr to h
        mov r14,r9                      ;r14 = ptr to sa_cone
        mov r15,[rbp+RBP_RA+40]         ;r15 = ptr to vol_cone
        vmovsd xmm14,real8 ptr [r8_pi]  ;xmm14 = pi
        vmovsd xmm15,real8 ptr [r8_3p0] ;xmm15 = 3.0

; Calculate cone surface areas and volumes
; sa = pi * r * (r + sqrt(r * r + h * h))
; vol = pi * r * r * h / 3
@@:     vmovsd xmm0,real8 ptr [r12+rbx]  ;xmm0 = r
        vmovsd xmm1,real8 ptr [r13+rbx]  ;xmm1 = h
        vmovsd xmm12,xmm12,xmm0          ;xmm12 = r
        vmovsd xmm13,xmm13,xmm1          ;xmm13 = h

        vmulsd xmm0,xmm0,xmm0     ;xmm0 = r * r
        vmulsd xmm1,xmm1,xmm1     ;xmm1 = h * h
        vaddsd xmm0,xmm0,xmm1     ;xmm0 = r * r + h * h

        vsqrtsd xmm0,xmm0,xmm0    ;xmm0 = sqrt(r * r + h * h)
        vaddsd xmm0,xmm0,xmm12    ;xmm0 = r + sqrt(r * r + h * h)
        vmulsd xmm0,xmm0,xmm12    ;xmm0 = r * (r + sqrt(r * r + h * h))
        vmulsd xmm0,xmm0,xmm14    ;xmm0 = pi * r * (r + sqrt(r * r + h * h))
        vmulsd xmm12,xmm12,xmm12   ;xmm12 = r * r
        vmulsd xmm13,xmm13,xmm14   ;xmm13 = h * pi
        vmulsd xmm13,xmm13,xmm12   ;xmm13 = pi * r * r * h
        vdivsd xmm13,xmm13,xmm15   ;xmm13 = pi * r * r * h / 3

        vmovsd real8 ptr [r14+rbx],xmm0  ;save surface area
        vmovsd real8 ptr [r15+rbx],xmm13 ;save volume

        add rbx,8                        ;set rbx to next element
        dec esi                          ;update counter
        jnz @B                           ;repeat until done

        mov eax,1                        ;set success return code

; Restore non-volatile XMM registers
Done:   vmovdqa xmm12,xmmword ptr [rbp-STK_LOCAL2+48]
        vmovdqa xmm13,xmmword ptr [rbp-STK_LOCAL2+32]
        vmovdqa xmm14,xmmword ptr [rbp-STK_LOCAL2+16]
        vmovdqa xmm15,xmmword ptr [rbp-STK_LOCAL2]

; Function epilog
        lea rsp,[rbp+STK_LOCAL1+STK_PAD]    ;restore rsp
        pop r15                          ;restore NV GP registers
        pop r14
        pop r13
        pop r12
        pop rsi
        pop rbx
        pop rbp
        ret

Cc3_    endp
        end
;-------------------------------------------------
;####### Ch05_12.asm
;-------------------------------------------------
; extern "C" bool Cc4_(const double* ht, const double* wt, int n, double* bsa1, double* bsa2,
double* bsa3);
        include <MacrosX86-64-AVX.asmh>

                .const
r8_0p007184     real8 0.007184
r8_0p725        real8 0.725
r8_0p425        real8 0.425
r8_0p0235       real8 0.0235
r8_0p42246      real8 0.42246
r8_0p51456      real8 0.51456
r8_3600p0       real8 3600.0

        .code
        extern pow:proc

Cc4_ proc frame
        _CreateFrame Cc4_,16,64,rbx,rsi,r12,r13,r14,r15
        _SaveXmmRegs xmm6,xmm7,xmm8,xmm9
        _EndProlog

; Save argument registers to home area (optional). Note that the home
; area can also be used to store other transient data values.
        mov qword ptr [rbp+Cc4_OffsetHomeRCX],rcx
        mov qword ptr [rbp+Cc4_OffsetHomeRDX],rdx
        mov qword ptr [rbp+Cc4_OffsetHomeR8],r8
        mov qword ptr [rbp+Cc4_OffsetHomeR9],r9

; Initialize processing loop pointers.  Note that the pointers are
; maintained in non-volatile registers, which eliminates reloads
; after the calls to pow().
        test r8d,r8d                        ;is n > 0?
        jg @F                               ;jump if n > 0

        xor eax,eax                         ;set error return code
        jmp Done

@@:     mov [rbp],r8d                       ;save n to local var
        mov r12,rcx                         ;r12 = ptr to ht
        mov r13,rdx                         ;r13 = ptr to wt
        mov r14,r9                          ;r14 = ptr to bsa1
        mov r15,[rbp+Cc4_OffsetStackArgs]   ;r15 = ptr to bsa2
        mov rbx,[rbp+Cc4_OffsetStackArgs+8] ;rbx = ptr to bsa3
        xor rsi,rsi                         ;array element offset

; Allocate home space on stack for use by pow()
        sub rsp,32

; Calculate bsa1 = 0.007184 * pow(ht, 0.725) * pow(wt, 0.425);
@@:     vmovsd xmm0,real8 ptr [r12+rsi]         ;xmm0 = height
        vmovsd xmm8,xmm8,xmm0
        vmovsd xmm1,real8 ptr [r8_0p725]
        call pow                                ;xmm0 = pow(ht, 0.725)
        vmovsd xmm6,xmm6,xmm0

        vmovsd xmm0,real8 ptr [r13+rsi]         ;xmm0 = weight
        vmovsd xmm9,xmm9,xmm0
        vmovsd xmm1,real8 ptr [r8_0p425]
        call pow                                ;xmm0 = pow(wt, 0.425)
        vmulsd xmm6,xmm6,real8 ptr [r8_0p007184]
        vmulsd xmm6,xmm6,xmm0                   ;xmm6 = bsa1

; Calculate bsa2 = 0.0235 * pow(ht, 0.42246) * pow(wt, 0.51456);
        vmovsd xmm0,xmm0,xmm8                   ;xmm0 = height
        vmovsd xmm1,real8 ptr [r8_0p42246]
        call pow                                ;xmm0 = pow(ht, 0.42246)
        vmovsd xmm7,xmm7,xmm0

        vmovsd xmm0,xmm0,xmm9                   ;xmm0 = weight
        vmovsd xmm1,real8 ptr [r8_0p51456]
        call pow                                ;xmm0 = pow(wt, 0.51456)
        vmulsd xmm7,xmm7,real8 ptr [r8_0p0235]
        vmulsd xmm7,xmm7,xmm0                   ;xmm7 = bsa2

; Calculate bsa3 = sqrt(ht * wt / 60.0);
        vmulsd xmm8,xmm8,xmm9
        vdivsd xmm8,xmm8,real8 ptr [r8_3600p0]
        vsqrtsd xmm8,xmm8,xmm8              ;xmm8 = bsa3

; Save BSA results
        vmovsd real8 ptr [r14+rsi],xmm6     ;save bsa1 result
        vmovsd real8 ptr [r15+rsi],xmm7     ;save bsa2 result
        vmovsd real8 ptr [rbx+rsi],xmm8     ;save bsa3 result

        add rsi,8                           ;update array offset
        dec dword ptr [rbp]                 ;n = n - 1
        jnz @B
        mov eax,1                           ;set success return code

Done:   _RestoreXmmRegs xmm6,xmm7,xmm8,xmm9
        _DeleteFrame rbx,rsi,r12,r13,r14,r15
        ret

Cc4_ endp
        end
;-------------------------------------------------
;####### Ch06_01.asm
;-------------------------------------------------
            .const
            align 16
AbsMaskF32  dword 7fffffffh, 7fffffffh, 7fffffffh, 7fffffffh  ;Absolute value mask for SPFP
```

```
AbsMaskF64  qword 7fffffffffffffffh, 7fffffffffffffffh    ;Absolute value mask for DPFP

; extern "C" void AvxPackedMathF32_(const XmmVal& a, const XmmVal& b, XmmVal c[8]);

            .code
AvxPackedMathF32_ proc
; Load packed SPFP values
        vmovaps xmm0,xmmword ptr [rcx]   ;xmm0 = a
        vmovaps xmm1,xmmword ptr [rdx]   ;xmm1 = b

; Packed SPFP addition
        vaddps xmm2,xmm0,xmm1
        vmovaps [r8+0],xmm2

; Packed SPFP subtraction
        vsubps xmm2,xmm0,xmm1
        vmovaps [r8+16],xmm2

; Packed SPFP multiplication
        vmulps xmm2,xmm0,xmm1
        vmovaps [r8+32],xmm2

; Packed SPFP division
        vdivps xmm2,xmm0,xmm1
        vmovaps [r8+48],xmm2

; Packed SPFP absolute value (b)
        vandps xmm2,xmm1,xmmword ptr [AbsMaskF32]
        vmovaps [r8+64],xmm2

; Packed SPFP square root (a)
        vsqrtps xmm2,xmm0
        vmovaps [r8+80],xmm2

; Packed SPFP minimum
        vminps xmm2,xmm0,xmm1
        vmovaps [r8+96],xmm2

; Packed SPFP maximum
        vmaxps xmm2,xmm0,xmm1
        vmovaps [r8+112],xmm2
        ret
AvxPackedMathF32_ endp

; extern "C" void AvxPackedMathF64_(const XmmVal& a, const XmmVal& b, XmmVal c[8]);

AvxPackedMathF64_ proc
; Load packed DPFP values
        vmovapd xmm0,xmmword ptr [rcx]   ;xmm0 = a
        vmovapd xmm1,xmmword ptr [rdx]   ;xmm1 = b

; Packed DPFP addition
        vaddpd xmm2,xmm0,xmm1
        vmovapd [r8+0],xmm2

; Packed DPFP subtraction
        vsubpd xmm2,xmm0,xmm1
        vmovapd [r8+16],xmm2

; Packed DPFP multiplication
        vmulpd xmm2,xmm0,xmm1
        vmovapd [r8+32],xmm2

; Packed DPFP division
        vdivpd xmm2,xmm0,xmm1
        vmovapd [r8+48],xmm2
```

```
; Packed DPFP absolute value (b)
        vandpd xmm2,xmm1,xmmword ptr [AbsMaskF64]
        vmovapd [r8+64],xmm2

; Packed DPFP square root (a)
        vsqrtpd xmm2,xmm0
        vmovapd [r8+80],xmm2

; Packed DPFP minimum
        vminpd xmm2,xmm0,xmm1
        vmovapd [r8+96],xmm2

; Packed DPFP maximum
        vmaxpd xmm2,xmm0,xmm1
        vmovapd [r8+112],xmm2
        ret
AvxPackedMathF64_ endp
        end
;-------------------------------------------------
;####### Ch06_02.asm
        include <cmpequ.asmh>

; extern "C" void AvxPackedCompareF32_(const XmmVal& a, const XmmVal& b, XmmVal c[8]);

        .code
AvxPackedCompareF32_ proc
        vmovaps xmm0,[rcx]               ;xmm0 = a
        vmovaps xmm1,[rdx]               ;xmm1 = b

; Perform packed EQUAL compare
        vcmpps xmm2,xmm0,xmm1,CMP_EQ
        vmovdqa xmmword ptr [r8],xmm2

; Perform packed NOT EQUAL compare
        vcmpps xmm2,xmm0,xmm1,CMP_NEQ
        vmovdqa xmmword ptr [r8+16],xmm2

; Perform packed LESS THAN compare
        vcmpps xmm2,xmm0,xmm1,CMP_LT
        vmovdqa xmmword ptr [r8+32],xmm2

; Perform packed LESS THAN OR EQUAL compare
        vcmpps xmm2,xmm0,xmm1,CMP_LE
        vmovdqa xmmword ptr [r8+48],xmm2

 ; Perform packed GREATER THAN compare
        vcmpps xmm2,xmm0,xmm1,CMP_GT
        vmovdqa xmmword ptr [r8+64],xmm2

; Perform packed GREATER THAN OR EQUAL compare
        vcmpps xmm2,xmm0,xmm1,CMP_GE
        vmovdqa xmmword ptr [r8+80],xmm2

; Perform packed ORDERED compare
        vcmpps xmm2,xmm0,xmm1,CMP_ORD
        vmovdqa xmmword ptr [r8+96],xmm2

; Perform packed UNORDERED compare
        vcmpps xmm2,xmm0,xmm1,CMP_UNORD
        vmovdqa xmmword ptr [r8+112],xmm2
        ret
AvxPackedCompareF32_ endp

; extern "C" void AvxPackedCompareF64_(const XmmVal& a, const XmmVal& b, XmmVal c[8]);

AvxPackedCompareF64_ proc
        vmovapd xmm0,[rcx]               ;xmm0 = a
        vmovapd xmm1,[rdx]                       ;xmm1 = b

; Perform packed EQUAL compare
        vcmppd xmm2,xmm0,xmm1,CMP_EQ
        vmovdqa xmmword ptr [r8],xmm2

; Perform packed NOT EQUAL compare
        vcmppd xmm2,xmm0,xmm1,CMP_NEQ
        vmovdqa xmmword ptr [r8+16],xmm2

; Perform packed LESS THAN compare
        vcmppd xmm2,xmm0,xmm1,CMP_LT
        vmovdqa xmmword ptr [r8+32],xmm2

; Perform packed LESS THAN OR EQUAL compare
        vcmppd xmm2,xmm0,xmm1,CMP_LE
        vmovdqa xmmword ptr [r8+48],xmm2

 ; Perform packed GREATER THAN compare
        vcmppd xmm2,xmm0,xmm1,CMP_GT
        vmovdqa xmmword ptr [r8+64],xmm2

; Perform packed GREATER THAN OR EQUAL compare
        vcmppd xmm2,xmm0,xmm1,CMP_GE
        vmovdqa xmmword ptr [r8+80],xmm2

; Perform packed ORDERED compare
        vcmppd xmm2,xmm0,xmm1,CMP_ORD
        vmovdqa xmmword ptr [r8+96],xmm2

; Perform packed UNORDERED compare
        vcmppd xmm2,xmm0,xmm1,CMP_UNORD
        vmovdqa xmmword ptr [r8+112],xmm2
        ret
AvxPackedCompareF64_ endp
        end
;--------------------------------------------------
;####### Ch06_03.asm
;--------------------------------------------------
; extern "C" bool AvxPackedConvertFP_(const XmmVal& a, XmmVal& b, CvtOp cvt_op);
; Note:         This function requires linker option /LARGEADDRESSAWARE:NO
;               to be explicitly set.

        .code
AvxPackedConvertFP_ proc

; Make sure cvt_op is valid
        mov r9d,r8d                         ;r9 = cvt_op (zero extended)
        cmp r9,CvtOpTableCount              ;is cvt_op valid?
        jae InvalidCvtOp                    ;jmp if cvt_op is invalid

        mov eax,1                           ;set valid cvt_op return code
        jmp [CvtOpTable+r9*8]               ;jump to specified conversion

; Convert packed signed doubleword integers to packed SPFP values
I32_F32:
        vmovdqa xmm0,xmmword ptr [rcx]
        vcvtdq2ps xmm1,xmm0
        vmovaps xmmword ptr [rdx],xmm1
        ret

; Convert packed SPFP values to packed signed doubleword integers
F32_I32:
        vmovaps xmm0,xmmword ptr [rcx]
        vcvtps2dq xmm1,xmm0
        vmovdqa xmmword ptr [rdx],xmm1
        ret
```

```

; Convert packed signed doubleword integers to packed DPFP values
I32_F64:
        vmovdqa xmm0,xmmword ptr [rcx]
        vcvtdq2pd xmm1,xmm0
        vmovapd xmmword ptr [rdx],xmm1
        ret

; Convert packed DPFP values to packed signed doubleword integers
F64_I32:
        vmovapd xmm0,xmmword ptr [rcx]
        vcvtpd2dq xmm1,xmm0
        vmovdqa xmmword ptr [rdx],xmm1
        ret

; Convert packed SPFP to packed DPFP
F32_F64:
        vmovaps xmm0,xmmword ptr [rcx]
        vcvtps2pd xmm1,xmm0
        vmovapd xmmword ptr [rdx],xmm1
        ret

; Convert packed DPFP to packed SPFP
F64_F32:
        vmovapd xmm0,xmmword ptr [rcx]
        vcvtpd2ps xmm1,xmm0
        vmovaps xmmword ptr [rdx],xmm1
        ret

InvalidCvtOp:
        xor eax,eax                         ;set invalid cvt_op return code
        ret

; The order of values in the following table must match the enum CvtOp
; that's defined in Ch06_03.cpp.

            align 8
CvtOpTable  qword I32_F32, F32_I32
            qword I32_F64, F64_I32
            qword F32_F64, F64_F32
CvtOpTableCount equ ($ - CvtOpTable) / size qword

AvxPackedConvertFP_ endp
        end
;--------------------------------------------------
;####### Ch06_04.asm
;--------------------------------------------------
; extern "C" bool AvxCalcSqrts_(float* y, const float* x, size_t n);

        .code
AvxCalcSqrts_ proc
        xor eax,eax                         ;set error return code (also array offset)
        test r8,r8
        jz Done                             ;jump if n is zero

        test rcx,0fh
        jnz Done                            ;jump if 'y' is not aligned

        test rdx,0fh
        jnz Done                            ;jump if 'x' is not aligned

; Calculate packed square roots
        cmp r8,4
        jb FinalVals                        ;jump if n < 4
@@:     vsqrtps xmm0,xmmword ptr [rdx+rax]  ;calculate 4 square roots x[i+3:i]
        vmovaps xmmword ptr [rcx+rax],xmm0  ;save results to y[i+3:i]
        add rax,16                      ;update offset to next set of values
        sub r8,4
        cmp r8,4                        ;are there 4 or more elements remaining?
        jae @B                          ;jump if yes

; Calculate square roots of final 1 - 3 values, note switch to scalar instructions
FinalVals:
        test r8,r8                      ;more elements to process?
        jz SetRC                        ;jump if no more elements

        vsqrtss xmm0,xmm0,real4 ptr [rdx+rax]   ;calculate sqrt(x[i])
        vmovss real4 ptr [rcx+rax],xmm0     ;save result to y[i]
        add rax,4
        dec r8
        jz SetRC

        vsqrtss xmm0,xmm0,real4 ptr [rdx+rax]
        vmovss real4 ptr [rcx+rax],xmm0
        add rax,4
        dec r8
        jz SetRC

        vsqrtss xmm0,xmm0,real4 ptr [rdx+rax]
        vmovss real4 ptr [rcx+rax],xmm0

SetRC:  mov eax,1                       ;set success return code

Done:   ret
AvxCalcSqrts_ endp
        end
;-------------------------------------------------
;####### Ch06_05.asm
;-------------------------------------------------
        extern g_MinValInit:real4
        extern g_MaxValInit:real4

; extern "C" bool CalcArrayMinMaxF32_(float* min_val, float* max_val, const float* x, size_t n)
        .code
CalcArrayMinMaxF32_ proc
; Validate arguments
        xor eax,eax                     ;set error return code

        test r8,0fh                     ;is x aligned to 16-byte boundary?
        jnz Done                        ;jump if no

        vbroadcastss xmm4,real4 ptr [g_MinValInit] ;xmm4 = min values
        vbroadcastss xmm5,real4 ptr [g_MaxValInit] ;xmm5 = max values

        cmp r9,4
        jb FinalVals                    ;jump if n < 4

; Main processing loop
@@:     vmovaps xmm0,xmmword ptr [r8]   ;load next set of array values
        vminps xmm4,xmm4,xmm0           ;update packed min values
        vmaxps xmm5,xmm5,xmm0           ;update packed max values

        add r8,16
        sub r9,4
        cmp r9,4
        jae @B

; Process the final 1 - 3 values of the input array
FinalVals:
        test r9,r9
        jz SaveResults

        vminss xmm4,xmm4,real4 ptr [r8] ;update packed min values
        vmaxss xmm5,xmm5,real4 ptr [r8] ;update packed max values
        dec r9
        jz SaveResults

        vminss xmm4,xmm4,real4 ptr [r8+4]
        vmaxss xmm5,xmm5,real4 ptr [r8+4]
        dec r9
        jz SaveResults

        vminss xmm4,xmm4,real4 ptr [r8+8]
        vmaxss xmm5,xmm5,real4 ptr [r8+8]

; Calculate and save final min & max values
SaveResults:
        vshufps xmm0,xmm4,xmm4,00001110b    ;xmm0[63:0] = xmm4[128:64]
        vminps xmm1,xmm0,xmm4           ;xmm1[63:0] contains final 2 values
        vshufps xmm2,xmm1,xmm1,00000001b    ;xmm2[31:0] = xmm1[63:32]
        vminps xmm3,xmm2,xmm1           ;xmm3[31:0] contains final value
        vmovss real4 ptr [rcx],xmm3     ;save array min value

        vshufps xmm0,xmm5,xmm5,00001110b
        vmaxps xmm1,xmm0,xmm5
        vshufps xmm2,xmm1,xmm1,00000001b
        vmaxps xmm3,xmm2,xmm1
        vmovss real4 ptr [rdx],xmm3     ;save array max value

        mov eax,1                       ;set success return code
Done:   ret
CalcArrayMinMaxF32_ endp
        end
;-------------------------------------------------
;####### Ch06_06.asm
        include <MacrosX86-64-AVX.asmh>

        extern LsEpsilon:real8          ;global value defined in C++ file

; extern "C" bool AvxCalcLeastSquares_(const double* x, const double* y, int n, double* m, double* b);
; Returns       0 = error (invalid n or improperly aligned array), 1 = success

        .const
        align 16
AbsMaskF64  qword 7fffffffffffffffh, 7fffffffffffffffh  ;mask for DPFP absolute value

        .code
AvxCalcLeastSquares_ proc frame
        _CreateFrame LS_,0,48,rbx
        _SaveXmmRegs xmm6,xmm7,xmm8
        _EndProlog

; Validate arguments
        xor eax,eax                     ;set error return code
        cmp r8d,2
        jl Done                         ;jump if n < 2
        test rcx,0fh
        jnz Done                        ;jump if x not aligned to 16-byte boundary
        test rdx,0fh
        jnz Done                        ;jump if y not aligned to 16-byte boundary

; Perform required initializations
        vcvtsi2sd xmm3,xmm3,r8d         ;xmm3 = n
        mov eax,r8d
        and r8d,0fffffffeh              ;rd8 = n / 2 * 2
        and eax,1                       ;eax = n % 2

        vxorpd xmm4,xmm4,xmm4           ;sum_x (both qwords)
        vxorpd xmm5,xmm5,xmm5           ;sum_y (both qwords)
        vxorpd xmm6,xmm6,xmm6           ;sum_xx (both qwords)
        vxorpd xmm7,xmm7,xmm7           ;sum_xy (both qwords)
        xor ebx,ebx                     ;rbx = array offset
        mov r10,[rbp+LS_OffsetStackArgs]    ;r10 = b

; Calculate sum variables. Note that two values are processed each iteration.
@@:     vmovapd xmm0,xmmword ptr [rcx+rbx]  ;load next two x values
        vmovapd xmm1,xmmword ptr [rdx+rbx]  ;load next two y values

        vaddpd xmm4,xmm4,xmm0           ;update sum_x
        vaddpd xmm5,xmm5,xmm1           ;update sum_y

        vmulpd xmm2,xmm0,xmm0           ;calc x * x
        vaddpd xmm6,xmm6,xmm2           ;update sum_xx

        vmulpd xmm2,xmm0,xmm1           ;calc x * y
        vaddpd xmm7,xmm7,xmm2           ;update sum_xy

        add rbx,16                      ;rbx = next offset
        sub r8d,2                       ;adjust counter
        jnz @B                          ;repeat until done

; Update sum variables with the final x, y values if 'n' is odd
        or eax,eax
        jz CalcFinalSums                ;jump if n is even
        vmovsd xmm0,real8 ptr [rcx+rbx] ;load final x
        vmovsd xmm1,real8 ptr [rdx+rbx] ;load final y

        vaddsd xmm4,xmm4,xmm0           ;update sum_x
        vaddsd xmm5,xmm5,xmm1           ;update sum_y

        vmulsd xmm2,xmm0,xmm0           ;calc x * x
        vaddsd xmm6,xmm6,xmm2           ;update sum_xx

        vmulsd xmm2,xmm0,xmm1           ;calc x * y
        vaddsd xmm7,xmm7,xmm2           ;update sum_xy

; Calculate final sum_x, sum_y, sum_xx, sum_xy
CalcFinalSums:
        vhaddpd xmm4,xmm4,xmm4          ;xmm4[63:0] = final sum_x
        vhaddpd xmm5,xmm5,xmm5          ;xmm5[63:0] = final sum_y
        vhaddpd xmm6,xmm6,xmm6          ;xmm6[63:0] = final sum_xx
        vhaddpd xmm7,xmm7,xmm7          ;xmm7[63:0] = final sum_xy

; Compute denominator and make sure it's valid
; denom = n * sum_xx - sum_x * sum_x
        vmulsd xmm0,xmm3,xmm6           ;n * sum_xx
        vmulsd xmm1,xmm4,xmm4           ;sum_x * sum_x
        vsubsd xmm2,xmm0,xmm1           ;denom
        vandpd xmm8,xmm2,xmmword ptr [AbsMaskF64] ;fabs(denom)
        vcomisd xmm8,real8 ptr [LsEpsilon]
        jb BadDen                       ;jump if denom < fabs(denom)
; Compute and save slope
; slope = (n * sum_xy - sum_x * sum_y) / denom
        vmulsd xmm0,xmm3,xmm7           ;n * sum_xy
        vmulsd xmm1,xmm4,xmm5           ;sum_x * sum_y
        vsubsd xmm2,xmm0,xmm1           ;slope numerator
        vdivsd xmm3,xmm2,xmm8           ;final slope
        vmovsd real8 ptr [r9],xmm3      ;save slope

; Compute and save intercept
; intercept = (sum_xx * sum_y - sum_x * sum_xy) / denom
        vmulsd xmm0,xmm6,xmm5           ;sum_xx * sum_y
        vmulsd xmm1,xmm4,xmm7           ;sum_x * sum_xy
        vsubsd xmm2,xmm0,xmm1           ;intercept numerator
        vdivsd xmm3,xmm2,xmm8           ;final intercept
        vmovsd real8 ptr [r10],xmm3     ;save intercept
```

```
        mov eax,1                       ;success return code
        jmp Done

; Bad denominator detected, set m and b to 0.0
BadDen: vxorpd xmm0,xmm0,xmm0
        vmovsd real8 ptr [r9],xmm0      ;*m = 0.0
        vmovsd real8 ptr [r10],xmm0     ;*b = 0.0
        xor eax,eax                     ;set error code

Done:   _RestoreXmmRegs xmm6,xmm7,xmm8
        _DeleteFrame rbx
        ret
AvxCalcLeastSquares_ endp
        end
;-------------------------------------------------
;####### Ch06_07.asm
        include <MacrosX86-64-AVX.asmh>

; _Mat4x4TransposeF32 macro
; Description:  This macro transposes a 4x4 matrix of single-precision
;               floating-point values.
;  Input Matrix                    Output Matrix
;  ---------------------------------------------------
;  xmm0    a3 a2 a1 a0             xmm4    d0 c0 b0 a0
;  xmm1    b3 b2 b1 b0             xmm5    d1 c1 b1 a1
;  xmm2    c3 c2 c1 c0             xmm6    d2 c2 b2 a2
;  xmm3    d3 d2 d1 d0             xmm7    d3 c3 b3 a3

_Mat4x4TransposeF32 macro
        vunpcklps xmm6,xmm0,xmm1        ;xmm6 = b1 a1 b0 a0
        vunpckhps xmm0,xmm0,xmm1        ;xmm0 = b3 a3 b2 a2
        vunpcklps xmm7,xmm2,xmm3        ;xmm7 = d1 c1 d0 c0
        vunpckhps xmm1,xmm2,xmm3        ;xmm1 = d3 c3 d2 c2

        vmovlhps xmm4,xmm6,xmm7         ;xmm4 = d0 c0 b0 a0
        vmovhlps xmm5,xmm7,xmm6         ;xmm5 = d1 c1 b1 a1
        vmovlhps xmm6,xmm0,xmm1         ;xmm6 = d2 c2 b2 a2
        vmovhlps xmm7,xmm1,xmm0         ;xmm7 = d3 c3 b2 a3
        endm

; extern "C" void AvxMat4x4TransposeF32_(float* m_des, const float* m_src)
        .code
AvxMat4x4TransposeF32_ proc frame
        _CreateFrame MT_,0,32
        _SaveXmmRegs xmm6,xmm7
        _EndProlog

; Transpose matrix m_src1
        vmovaps xmm0,[rdx]              ;xmm0 = m_src.row_0
        vmovaps xmm1,[rdx+16]           ;xmm1 = m_src.row_1
        vmovaps xmm2,[rdx+32]           ;xmm2 = m_src.row_2
        vmovaps xmm3,[rdx+48]           ;xmm3 = m_src.row_3

        _Mat4x4TransposeF32

        vmovaps [rcx],xmm4              ;save m_des.row_0
        vmovaps [rcx+16],xmm5           ;save m_des.row_1
        vmovaps [rcx+32],xmm6           ;save m_des.row_2
        vmovaps [rcx+48],xmm7           ;save m_des.row_3

Done:   _RestoreXmmRegs xmm6,xmm7
        _DeleteFrame
        ret
AvxMat4x4TransposeF32_ endp
        end
;-------------------------------------------------
;####### Ch06_08.asm
```

```
        include <MacrosX86-64-AVX.asmh>

; _Mat4x4MulCalcRowF32 macro
; Description:  This macro is used to compute one row of a 4x4 matrix
;               multiply.
; Registers:    xmm0 = m_src2.row0
;               xmm1 = m_src2.row1
;               xmm2 = m_src2.row2
;               xmm3 = m_src2.row3
;               rcx = m_des ptr
;               rdx = m_src1 ptr
;               xmm4 - xmm7 = scratch registers

_Mat4x4MulCalcRowF32 macro disp
        vbroadcastss xmm4,real4 ptr [rdx+disp]  ;broadcast m_src1[i][0]
        vbroadcastss xmm5,real4 ptr [rdx+disp+4]    ;broadcast m_src1[i][1]
        vbroadcastss xmm6,real4 ptr [rdx+disp+8]    ;broadcast m_src1[i][2]
        vbroadcastss xmm7,real4 ptr [rdx+disp+12]   ;broadcast m_src1[i][3]

        vmulps xmm4,xmm4,xmm0                   ;m_src1[i][0] * m_src2.row_0
        vmulps xmm5,xmm5,xmm1                   ;m_src1[i][1] * m_src2.row_1
        vmulps xmm6,xmm6,xmm2                   ;m_src1[i][2] * m_src2.row_2
        vmulps xmm7,xmm7,xmm3                   ;m_src1[i][3] * m_src2.row_3

        vaddps xmm4,xmm4,xmm5                   ;calc m_des.row_i
        vaddps xmm6,xmm6,xmm7
        vaddps xmm4,xmm4,xmm6

        vmovaps[rcx+disp],xmm4                  ;save m_des.row_i
        endm

; extern "C" void AvxMat4x4MulF32_(float* m_des, const float* m_src1, const float* m_src2)
; Description:  The following function computes the product of two
;               single-precision floating-point 4x4 matrices.

        .code
AvxMat4x4MulF32_ proc frame
        _CreateFrame MM_,0,32
        _SaveXmmRegs xmm6,xmm7
        _EndProlog

; Compute matrix product m_des = m_src1 * m_src2
        vmovaps xmm0,[r8]               ;xmm0 = m_src2.row_0
        vmovaps xmm1,[r8+16]            ;xmm1 = m_src2.row_1
        vmovaps xmm2,[r8+32]            ;xmm2 = m_src2.row_2
        vmovaps xmm3,[r8+48]            ;xmm3 = m_src2.row_3

        _Mat4x4MulCalcRowF32 0          ;calculate m_des.row_0
        _Mat4x4MulCalcRowF32 16         ;calculate m_des.row_1
        _Mat4x4MulCalcRowF32 32         ;calculate m_des.row_2
        _Mat4x4MulCalcRowF32 48         ;calculate m_des.row_3

Done:   _RestoreXmmRegs xmm6,xmm7
        _DeleteFrame
        ret
AvxMat4x4MulF32_ endp
        end
;-------------------------------------------------
;####### Ch07_01.asm
;-------------------------------------------------
; extern "C" void AvxPackedAddI16_(const XmmVal& a, const XmmVal& b, XmmVal c[2])
        .code
AvxPackedAddI16_ proc

; Packed signed word addition
        vmovdqa xmm0,xmmword ptr [rcx]  ;xmm0 = a
        vmovdqa xmm1,xmmword ptr [rdx]  ;xmm1 = b
```

```
        vpaddw xmm2,xmm0,xmm1           ;packed add - wraparound
        vpaddsw xmm3,xmm0,xmm1          ;packed add - saturated

        vmovdqa xmmword ptr [r8],xmm2   ;save c[0]
        vmovdqa xmmword ptr [r8+16],xmm3    ;save c[1]
        ret
AvxPackedAddI16_ endp

; extern "C" void AvxPackedSubI16_(const XmmVal& a, const XmmVal& b, XmmVal c[2])
AvxPackedSubI16_ proc

; Packed signed word subtraction
        vmovdqa xmm0,xmmword ptr [rcx]  ;xmm0 = a
        vmovdqa xmm1,xmmword ptr [rdx]  ;xmm1 = b

        vpsubw xmm2,xmm0,xmm1           ;packed sub - wraparound
        vpsubsw xmm3,xmm0,xmm1          ;packed sub - saturated

        vmovdqa xmmword ptr [r8],xmm2   ;save c[0]
        vmovdqa xmmword ptr [r8+16],xmm3    ;save c[1]
        ret
AvxPackedSubI16_ endp

; extern "C" void AvxPackedAddU16_(const XmmVal& a, const XmmVal& b, XmmVal c[2])
AvxPackedAddU16_ proc

; Packed unsigned word addition
        vmovdqu xmm0,xmmword ptr [rcx]  ;xmm0 = a
        vmovdqu xmm1,xmmword ptr [rdx]  ;xmm1 = b

        vpaddw xmm2,xmm0,xmm1           ;packed add - wraparound
        vpaddusw xmm3,xmm0,xmm1         ;packed add - saturated

        vmovdqu xmmword ptr [r8],xmm2   ;save c[0]
        vmovdqu xmmword ptr [r8+16],xmm3    ;save c[1]
        ret
AvxPackedAddU16_ endp

; extern "C" void AvxPackedSubU16_(const XmmVal& a, const XmmVal& b, XmmVal c[2])
AvxPackedSubU16_ proc

; Packed unsigned word subtraction
        vmovdqu xmm0,xmmword ptr [rcx]  ;xmm0 = a
        vmovdqu xmm1,xmmword ptr [rdx]  ;xmm1 = b

        vpsubw xmm2,xmm0,xmm1           ;packed sub - wraparound
        vpsubusw xmm3,xmm0,xmm1         ;packed sub - saturated

        vmovdqu xmmword ptr [r8],xmm2   ;save c[0]
        vmovdqu xmmword ptr [r8+16],xmm3    ;save c[1]
        ret
AvxPackedSubU16_ endp
        end
;-------------------------------------------------
;####### Ch07_02.asm
;-------------------------------------------------
; extern "C" bool AvxPackedIntegerShift_(XmmVal& b, const XmmVal& a, ShiftOp shift_op, unsigned int count)
; Returns:      0 = invalid shift_op argument, 1 = success
; Note:         This module requires linker option /LARGEADDRESSAWARE:NO
;               to be explicitly set.

        .code
AvxPackedIntegerShift_ proc
; Make sure 'shift_op' is valid
        mov r8d,r8d                 ;zero extend shift_op
        cmp r8,ShiftOpTableCount    ;compare against table count
        jae Error                   ;jump if shift_op is invalid

; Jump to the operation specified by shift_op
        vmovdqa xmm0,xmmword ptr [rdx]  ;xmm0 = a
        vmovd xmm1,r9d              ;xmm1[31:0] = shift count
        mov eax,1                   ;set success return code
        jmp [ShiftOpTable+r8*8]

; Packed shift left logical - word
U16_LL: vpsllw xmm2,xmm0,xmm1
        vmovdqa xmmword ptr [rcx],xmm2
        ret

; Packed shift right logical - word
U16_RL: vpsrlw xmm2,xmm0,xmm1
        vmovdqa xmmword ptr [rcx],xmm2
        ret

; Packed shift right arithmetic - word
U16_RA: vpsraw xmm2,xmm0,xmm1
        vmovdqa xmmword ptr [rcx],xmm2
        ret

; Packed shift left logical - doubleword
U32_LL: vpslld xmm2,xmm0,xmm1
        vmovdqa xmmword ptr [rcx],xmm2
        ret

; Packed shift right logical - doubleword
U32_RL: vpsrld xmm2,xmm0,xmm1
        vmovdqa xmmword ptr [rcx],xmm2
        ret

; Packed shift right arithmetic - doubleword
U32_RA: vpsrad xmm2,xmm0,xmm1
        vmovdqa xmmword ptr [rcx],xmm2
        ret

Error:  xor eax,eax                 ;set error code
        vpxor xmm0,xmm0,xmm0
        vmovdqa xmmword ptr [rcx],xmm0  ;set result to zero
        ret

; The order of the labels in the following table must correspond
; to the enums that are defined in .cpp file.

                align 8
ShiftOpTable    qword U16_LL, U16_RL, U16_RA
                qword U32_LL, U32_RL, U32_RA
ShiftOpTableCount equ ($ - ShiftOpTable) / size qword

AvxPackedIntegerShift_ endp
        end
;-------------------------------------------------
;####### Ch07_03.asm
;-------------------------------------------------
; extern "C" void AvxPackedMulI16_(XmmVal c[2], const XmmVal* a, const XmmVal* b)
        .code
AvxPackedMulI16_ proc
        vmovdqa xmm0,xmmword ptr [rdx]  ;xmm0 = a
        vmovdqa xmm1,xmmword ptr [r8]   ;xmm1 = b

        vpmullw xmm2,xmm0,xmm1          ;xmm2 = packed a * b low result
        vpmulhw xmm3,xmm0,xmm1          ;xmm3 = packed a * b high result

        vpunpcklwd xmm4,xmm2,xmm3       ;merge low and high results
        vpunpckhwd xmm5,xmm2,xmm3       ;into final signed dwords
```

```
        vmovdqa xmmword ptr [rcx],xmm4  ;save final results
        vmovdqa xmmword ptr [rcx+16],xmm5
        ret
AvxPackedMulI16_ endp

; extern "C" void AvxPackedMulI32A_(XmmVal c[2], const XmmVal* a, const XmmVal* b)
AvxPackedMulI32A_ proc

; Perform packed signed dword multiplication.  Note that vpmuldq
; performs following operations:
; xmm2[63:0]   = xmm0[31:0]  * xmm1[31:0]
; xmm2[127:64] = xmm0[95:64] * xmm1[95:64]

        vmovdqa xmm0,xmmword ptr [rdx]  ;xmm0 = a
        vmovdqa xmm1,xmmword ptr [r8]   ;xmm1 = b
        vpmuldq xmm2,xmm0,xmm1

; Shift source operands right by 4 bytes and repeat vpmuldq
        vpsrldq xmm0,xmm0,4
        vpsrldq xmm1,xmm1,4
        vpmuldq xmm3,xmm0,xmm1

; Save results
        vpextrq qword ptr [rcx],xmm2,0  ;save xmm2[63:0]
        vpextrq qword ptr [rcx+8],xmm3,0    ;save xmm3[63:0]
        vpextrq qword ptr [rcx+16],xmm2,1   ;save xmm2[127:63]
        vpextrq qword ptr [rcx+24],xmm3,1   ;save xmm3[127:63]
        ret
AvxPackedMulI32A_ endp

; extern "C" void AvxPackedMulI32B_(XmmVal*, const XmmVal* a, const XmmVal* b)
AvxPackedMulI32B_ proc

; Perform packed signed integer multiplication and save low packed dword result
        vmovdqa xmm0,xmmword ptr [rdx]          ;xmm0 = a
        vpmulld xmm1,xmm0,xmmword ptr [r8]      ;xmm1 = packed a * b
        vmovdqa xmmword ptr [rcx],xmm1          ;save packed dword result
        ret
AvxPackedMulI32B_ endp
        end
;-------------------------------------------------
;####### Ch07_04_.asm
;-------------------------------------------------
; extern "C" bool AvxCalcMinMaxU8_(uint8_t* x, size_t n, uint8_t* x_min, uint8_t* x_max)
; Returns:      0 = invalid n or unaligned array, 1 = success

            .const
            align 16
StartMinVal qword 0ffffffffffffffffh    ;Initial packed min values
            qword 0ffffffffffffffffh

StartMaxVal qword 0000000000000000h ;Initial packed max values
            qword 0000000000000000h

            .code
AvxCalcMinMaxU8_ proc

; Make sure 'n' is valid
        xor eax,eax                     ;set error return code
        or rdx,rdx                      ;is n == 0?
        jz Done                         ;jump if yes

        test rdx,3fh                    ;is n a multiple of 64?
        jnz Done                        ;jump if no

        test rcx,0fh                    ;is x properly aligned?
        jnz Done                        ;jump if no

; Initialize packed min-max values
        vmovdqa xmm2,xmmword ptr [StartMinVal]
        vmovdqa xmm3,xmm2                   ;xmm3:xmm2 = packed min values
        vmovdqa xmm4,xmmword ptr [StartMaxVal]
        vmovdqa xmm5,xmm4                   ;xmm5:xmm4 = packed max values

; Scan array for min & max values
@@:     vmovdqa xmm0,xmmword ptr [rcx]    ;xmm0 = x[i + 15] : x[i]
        vmovdqa xmm1,xmmword ptr [rcx+16] ;xmm1 = x[i + 31] : x[i + 16]
        vpminub xmm2,xmm2,xmm0
        vpminub xmm3,xmm3,xmm1            ;xmm3:xmm2 = updated min values
        vpmaxub xmm4,xmm4,xmm0
        vpmaxub xmm5,xmm5,xmm1            ;xmm5:xmm4 = updated max values

        vmovdqa xmm0,xmmword ptr [rcx+32] ;xmm0 = x[i + 47] : x[i + 32]
        vmovdqa xmm1,xmmword ptr [rcx+48] ;xmm1 = x[i + 63] : x[i + 48]
        vpminub xmm2,xmm2,xmm0
        vpminub xmm3,xmm3,xmm1            ;xmm3:xmm2 = updated min values
        vpmaxub xmm4,xmm4,xmm0
        vpmaxub xmm5,xmm5,xmm1            ;xmm5:xmm4 = updated max values

        add rcx,64
        sub rdx,64
        jnz @B

; Determine final minimum value
        vpminub xmm0,xmm2,xmm3          ;xmm0[127:0] = final 16 min vals
        vpsrldq xmm1,xmm0,8             ;xmm1[63:0] = xmm0[127:64]
        vpminub xmm2,xmm1,xmm0          ;xmm2[63:0] = final 8 min vals
        vpsrldq xmm3,xmm2,4             ;xmm3[31:0] = xmm2[63:32]
        vpminub xmm0,xmm3,xmm2          ;xmm0[31:0] = final 4 min vals
        vpsrldq xmm1,xmm0,2             ;xmm1[15:0] = xmm0[31:16]
        vpminub xmm2,xmm1,xmm0          ;xmm2[15:0] = final 2 min vals
        vpextrw eax,xmm2,0              ;ax = final 2 min vals
        cmp al,ah
        jbe @F                          ;jump if al <= ah
        mov al,ah                       ;al = final min value
@@:     mov [r8],al                     ;save final min

; Determine final maximum value
        vpmaxub xmm0,xmm4,xmm5          ;xmm0[127:0] = final 16 max vals
        vpsrldq xmm1,xmm0,8             ;xmm1[63:0] = xmm0[127:64]
        vpmaxub xmm2,xmm1,xmm0          ;xmm2[63:0] = final 8 max vals
        vpsrldq xmm3,xmm2,4             ;xmm3[31:0] = xmm2[63:32]
        vpmaxub xmm0,xmm3,xmm2          ;xmm0[31:0] = final 4 max vals
        vpsrldq xmm1,xmm0,2             ;xmm1[15:0] = xmm0[31:16]
        vpmaxub xmm2,xmm1,xmm0          ;xmm2[15:0] = final 2 max vals
        vpextrw eax,xmm2,0              ;ax = final 2 min vals
        cmp al,ah
        jae @F                          ;jump if al >= ah
        mov al,ah                       ;al = final max value
@@:     mov [r9],al                     ;save final max

        mov eax,1                       ;set success return code
Done:   ret
AvxCalcMinMaxU8_ endp
        end
;-------------------------------------------------
;####### Ch07_05.asm
        include <MacrosX86-64-AVX.asmh>
        extern g_NumElementsMax:qword

; extern "C" bool AvxCalcMeanU8_(const Uint8* x, size_t n, int64_t* sum_x, double* mean);
; Returns       0 = invalid n or unaligned array, 1 = success
```

```
        .code
AvxCalcMeanU8_ proc frame
        _CreateFrame CM_,0,64
        _SaveXmmRegs xmm6,xmm7,xmm8,xmm9
        _EndProlog

; Verify function arguments
        xor eax,eax                     ;set error return code
        or rdx,rdx
        jz Done                         ;jump if n == 0

        cmp rdx,[g_NumElementsMax]
        jae Done                        ;jump if n > NumElementsMax

        test rdx,3fh
        jnz Done                        ;jump if (n % 64) != 0

        test rcx,0fh
        jnz Done                        ;jump if x is not properly aligned

; Perform required initializations
        mov r10,rdx                     ;save n for later use
        add rdx,rcx                     ;rdx = end of array
        vpxor xmm8,xmm8,xmm8            ;xmm8 = packed intermediate sums (4 dwords)
        vpxor xmm9,xmm9,xmm9            ;xmm9 = packed zero for promotions

; Promote 32 pixel values from bytes to words, then sum the words
@@:     vmovdqa xmm0,xmmword ptr [rcx]
        vmovdqa xmm1,xmmword ptr [rcx+16]   ;xmm1:xmm0 = 32 pixels
        vpunpcklbw xmm2,xmm0,xmm9       ;xmm2 = 8 words
        vpunpckhbw xmm3,xmm0,xmm9       ;xmm3 = 8 words
        vpunpcklbw xmm4,xmm1,xmm9       ;xmm4 = 8 words
        vpunpckhbw xmm5,xmm1,xmm9       ;xmm5 = 8 words
        vpaddw xmm0,xmm2,xmm3
        vpaddw xmm1,xmm4,xmm5
        vpaddw xmm6,xmm0,xmm1           ;xmm6 = packed sums (8 words)
; Promote another 32 pixel values from bytes to words, then sum the words
        vmovdqa xmm0,xmmword ptr [rcx+32]
        vmovdqa xmm1,xmmword ptr [rcx+48]   ;xmm1:xmm0 = 32 pixels
        vpunpcklbw xmm2,xmm0,xmm9       ;xmm2 = 8 words
        vpunpckhbw xmm3,xmm0,xmm9       ;xmm3 = 8 words
        vpunpcklbw xmm4,xmm1,xmm9       ;xmm4 = 8 words
        vpunpckhbw xmm5,xmm1,xmm9       ;xmm5 = 8 words
        vpaddw xmm0,xmm2,xmm3
        vpaddw xmm1,xmm4,xmm5
        vpaddw xmm7,xmm0,xmm1           ;xmm7 = packed sums (8 words)
; Promote packed sums to dwords, then update dword sums
        vpaddw xmm0,xmm6,xmm7           ;xmm0 = packed sums (8 words)
        vpunpcklwd xmm1,xmm0,xmm9       ;xmm1 = packed sums (4 dwords)
        vpunpckhwd xmm2,xmm0,xmm9       ;xmm2 = packed sums (4 dwords)
        vpaddd xmm8,xmm8,xmm1
        vpaddd xmm8,xmm8,xmm2

        add rcx,64                      ;rcx = next 64 byte block
        cmp rcx,rdx
        jne @B                          ;repeat loop if not done

; Compute final sum_x (note vpextrd zero extends extracted dword to 64 bits)
        vpextrd eax,xmm8,0              ;rax = partial sum 0
        vpextrd edx,xmm8,1              ;rdx = partial sum 1
        add rax,rdx
        vpextrd ecx,xmm8,2              ;rcx = partial sum 2
        vpextrd edx,xmm8,3              ;rdx = partial sum 3
        add rax,rcx
        add rax,rdx
        mov [r8],rax                    ;save sum_x
```

```
; Compute mean value
        vcvtsi2sd xmm0,xmm0,rax         ;xmm0 = sum_x (DPFP)
        vcvtsi2sd xmm1,xmm1,r10         ;xmm1 = n (DPFP)
        vdivsd xmm2,xmm0,xmm1           ;calc mean = sum_x / n
        vmovsd real8 ptr [r9],xmm2      ;save mean

        mov eax,1                       ;set success return code

Done:   _RestoreXmmRegs xmm6,xmm7,xmm8,xmm9
        _DeleteFrame
        ret
AvxCalcMeanU8_ endp
        end
;-----------------------------------------------
;####### Ch07_06.asm
        include <MacrosX86-64-AVX.asmh>        include <cmpequ.asmh>

                    .const
                    align 16
Uint8ToFloat        real4 255.0, 255.0, 255.0, 255.0
FloatToUint8Min     real4 0.0, 0.0, 0.0, 0.0
FloatToUint8Max     real4 1.0, 1.0, 1.0, 1.0
FloatToUint8Scale   real4 255.0, 255.0, 255.0, 255.0

        extern c_NumPixelsMax:dword

; extern "C" bool ConvertImgU8ToF32_(float* des, const uint8_t* src, uint32_t num_pixels)
        .code
ConvertImgU8ToF32_ proc frame
        _CreateFrame U2F_,0,160
        _SaveXmmRegs xmm6,xmm7,xmm8,xmm9,xmm10,xmm11,xmm12,xmm13,xmm14,xmm15
        _EndProlog

; Make sure num_pixels is valid and pixel buffers are properly aligned
        xor eax,eax                     ;set error return code
        or r8d,r8d
        jz Done                         ;jump if num_pixels is zero
        cmp r8d,[c_NumPixelsMax]
        ja Done                         ;jump if num_pixels too big
        test r8d,1fh
        jnz Done                        ;jump if num_pixels % 32 != 0
        test rcx,0fh
        jnz Done                        ;jump if des not aligned
        test rdx,0fh
        jnz Done                        ;jump if src not aligned

; Initialize processing loop registers
        shr r8d,5                           ;number of pixel blocks
        vmovaps xmm6,xmmword ptr [Uint8ToFloat] ;xmm6 = packed 255.0f
        vpxor xmm7,xmm7,xmm7                ;xmm7 = packed 0

; Load the next block of 32 pixels
@@:     vmovdqa xmm0,xmmword ptr [rdx]  ;xmm0 = 16 pixels (x[i+15]:x[i])
        vmovdqa xmm1,xmmword ptr [rdx+16]   ;xmm8 = 16 pixels (x[i+31]:x[i+16])
; Promote the pixel values in xmm0 from unsigned bytes to unsigned dwords
        vpunpcklbw xmm2,xmm0,xmm7
        vpunpckhbw xmm3,xmm0,xmm7
        vpunpcklwd xmm8,xmm2,xmm7
        vpunpckhwd xmm9,xmm2,xmm7
        vpunpcklwd xmm10,xmm3,xmm7
        vpunpckhwd xmm11,xmm3,xmm7      ;xmm11:xmm8 = 16 dword pixels

; Promote the pixel values in xmm1 from unsigned bytes to unsigned dwords
        vpunpcklbw xmm2,xmm1,xmm7
        vpunpckhbw xmm3,xmm1,xmm7
        vpunpcklwd xmm12,xmm2,xmm7
        vpunpckhwd xmm13,xmm2,xmm7
```

```
        vpunpcklwd xmm14,xmm3,xmm7
        vpunpckhwd xmm15,xmm3,xmm7      ;xmm15:xmm12 = 16 dword pixels

; Convert pixel values from dwords to SPFP
        vcvtdq2ps xmm8,xmm8
        vcvtdq2ps xmm9,xmm9
        vcvtdq2ps xmm10,xmm10
        vcvtdq2ps xmm11,xmm11           ;xmm11:xmm8 = 16 SPFP pixels

        vcvtdq2ps xmm12,xmm12
        vcvtdq2ps xmm13,xmm13
        vcvtdq2ps xmm14,xmm14
        vcvtdq2ps xmm15,xmm15           ;xmm15:xmm12 = 16 SPFP pixels

; Normalize all pixel values to [0.0, 1.0] and save the results
        vdivps xmm0,xmm8,xmm6
        vmovaps xmmword ptr [rcx],xmm0  ;save pixels 0 - 3
        vdivps xmm1,xmm9,xmm6
        vmovaps xmmword ptr [rcx+16],xmm1   ;save pixels 4 - 7
        vdivps xmm2,xmm10,xmm6
        vmovaps xmmword ptr [rcx+32],xmm2   ;save pixels 8 - 11
        vdivps xmm3,xmm11,xmm6
        vmovaps xmmword ptr [rcx+48],xmm3   ;save pixels 12 - 15

        vdivps xmm0,xmm12,xmm6
        vmovaps xmmword ptr [rcx+64],xmm0   ;save pixels 16 - 19
        vdivps xmm1,xmm13,xmm6
        vmovaps xmmword ptr [rcx+80],xmm1   ;save pixels 20 - 23
        vdivps xmm2,xmm14,xmm6
        vmovaps xmmword ptr [rcx+96],xmm2   ;save pixels 24 - 27
        vdivps xmm3,xmm15,xmm6
        vmovaps xmmword ptr [rcx+112],xmm3  ;save pixels 28 - 31

        add rdx,32                      ;update src ptr
        add rcx,128                     ;update des ptr
        sub r8d,1
        jnz @B                          ;repeat until done
        mov eax,1                       ;set success return code

Done:   _RestoreXmmRegs xmm6,xmm7,xmm8,xmm9,xmm10,xmm11,xmm12,xmm13,xmm14,xmm15
        _DeleteFrame
        ret

ConvertImgU8ToF32_ endp

; extern "C" bool ConvertImgF32ToU8_(uint8_t* des, const float* src, uint32_t num_pixels)
ConvertImgF32ToU8_ proc frame
        _CreateFrame F2U_,0,96
        _SaveXmmRegs xmm6,xmm7,xmm12,xmm13,xmm14,xmm15
        _EndProlog

; Make sure num_pixels is valid and pixel buffers are properly aligned
        xor eax,eax                     ;set error return code
        or r8d,r8d
        jz Done                         ;jump if num_pixels is zero
        cmp r8d,[c_NumPixelsMax]
        ja Done                         ;jump if num_pixels too big
        test r8d,1fh
        jnz Done                        ;jump if num_pixels % 32 != 0
        test rcx,0fh
        jnz Done                        ;jump if des not aligned
        test rdx,0fh
        jnz Done                        ;jump if src not aligned

; Load required packed constants into registers
        vmovaps xmm13,xmmword ptr [FloatToUint8Scale] ;xmm13 = packed 255.0
        vmovaps xmm14,xmmword ptr [FloatToUint8Min]   ;xmm14 = packed 0.0
        vmovaps xmm15,xmmword ptr [FloatToUint8Max]   ;xmm15 = packed 1.0

        shr r8d,4                       ;number of pixel blocks
LP1:    mov r9d,4                       ;num pixel quartets per block

; Convert 16 float pixels to uint8_t
LP2:    vmovaps xmm0,xmmword ptr [rdx]  ;xmm0 = next pixel quartet
        vcmpps xmm1,xmm0,xmm14,CMP_LT   ;compare pixels to 0.0
        vandnps xmm2,xmm1,xmm0          ;clip pixels < 0.0 to 0.0

        vcmpps xmm3,xmm2,xmm15,CMP_GT   ;compare pixels to 1.0
        vandps xmm4,xmm3,xmm15          ;clip pixels > 1.0 to 1.0
        vandnps xmm5,xmm3,xmm2          ;xmm5 = pixels <= 1.0
        vorps xmm6,xmm5,xmm4            ;xmm6 = final clipped pixels
        vmulps xmm7,xmm6,xmm13          ;xmm7 = FP pixels [0.0, 255.0]

        vcvtps2dq xmm0,xmm7             ;xmm0 = dword pixels [0, 255]
        vpackusdw xmm1,xmm0,xmm0        ;xmm1[63:0] = word pixels
        vpackuswb xmm2,xmm1,xmm1        ;xmm2[31:0] = bytes pixels

; Save the current byte pixel quartet
        vpextrd eax,xmm2,0              ;eax = new pixel quartet
        vpsrldq xmm12,xmm12,4           ;adjust xmm12 for new quartet
        vpinsrd xmm12,xmm12,eax,3       ;xmm12[127:96] = new quartet

        add rdx,16                      ;update src ptr
        sub r9d,1
        jnz LP2                         ;repeat until done

; Save the current byte pixel block (16 pixels)
        vmovdqa xmmword ptr [rcx],xmm12 ;save current pixel block
        add rcx,16                      ;update des ptr
        sub r8d,1
        jnz LP1                         ;repeat until done
        mov eax,1                       ;set success return code

Done:   _RestoreXmmRegs xmm6,xmm7,xmm12,xmm13,xmm14,xmm15
        _DeleteFrame
        ret
ConvertImgF32ToU8_ endp
        end
;-------------------------------------------------
;####### Ch07_07.asm
        include <MacrosX86-64-AVX.asmh>

;  extern bool AvxBuildImageHistogram_(uint32_t* histo, const uint8_t* pixel_buff, uint32_t
num_pixels)
; Returns:      0 = invalid argument value, 1 = success

        .code
        extern c_NumPixelsMax:dword

AvxBuildImageHistogram_ proc frame
        _CreateFrame BIH_,1024,0,rbx,rsi,rdi
        _EndProlog

; Make sure num_pixels is valid
        xor eax,eax                     ;set error code
        test r8d,r8d
        jz Done                         ;jump if num_pixels is zero
        cmp r8d,[c_NumPixelsMax]
        ja Done                         ;jump if num_pixels too big
        test r8d,1fh
        jnz Done                        ;jump if num_pixels % 32 != 0

; Make sure histo & pixel_buff are properly aligned
        mov rsi,rcx                     ;rsi = ptr to histo
```

```
        test rsi,0fh
        jnz Done                        ;jump if histo misaligned
        mov r9,rdx
        test r9,0fh
        jnz Done                        ;jump if pixel_buff misaligned

; Initialize local histogram buffers (set all entries to zero)
        xor eax,eax
        mov rdi,rsi                     ;rdi = ptr to histo
        mov rcx,128                     ;rcx = size in qwords
        rep stosq                       ;zero histo
        mov rdi,rbp                     ;rdi = ptr to histo2
        mov rcx,128                     ;rcx = size in qwords
        rep stosq                       ;zero histo2

; Perform processing loop initializations
        shr r8d,5                       ;number of pixel blocks (32 pixels/block)
        mov rdi,rbp                     ;ptr to histo2

; Build the histograms
        align 16                        ;align jump target
@@:     vmovdqa xmm0,xmmword ptr [r9]   ;load pixel block
        vmovdqa xmm1,xmmword ptr [r9+16]    ;load pixel block

; Process pixels 0 - 3
        vpextrb rax,xmm0,0
        add dword ptr [rsi+rax*4],1     ;count pixel 0
        vpextrb rbx,xmm0,1
        add dword ptr [rdi+rbx*4],1     ;count pixel 1
        vpextrb rcx,xmm0,2
        add dword ptr [rsi+rcx*4],1     ;count pixel 2
        vpextrb rdx,xmm0,3
        add dword ptr [rdi+rdx*4],1     ;count pixel 3

; Process pixels 4 - 7
        vpextrb rax,xmm0,4
        add dword ptr [rsi+rax*4],1     ;count pixel 4
        vpextrb rbx,xmm0,5
        add dword ptr [rdi+rbx*4],1     ;count pixel 5
        vpextrb rcx,xmm0,6
        add dword ptr [rsi+rcx*4],1     ;count pixel 6
        vpextrb rdx,xmm0,7
        add dword ptr [rdi+rdx*4],1     ;count pixel 7

; Process pixels 8 - 11
        vpextrb rax,xmm0,8
        add dword ptr [rsi+rax*4],1     ;count pixel 8
        vpextrb rbx,xmm0,9
        add dword ptr [rdi+rbx*4],1     ;count pixel 9
        vpextrb rcx,xmm0,10
        add dword ptr [rsi+rcx*4],1     ;count pixel 10
        vpextrb rdx,xmm0,11
        add dword ptr [rdi+rdx*4],1     ;count pixel 11

; Process pixels 12 - 15
        vpextrb rax,xmm0,12
        add dword ptr [rsi+rax*4],1     ;count pixel 12
        vpextrb rbx,xmm0,13
        add dword ptr [rdi+rbx*4],1     ;count pixel 13
        vpextrb rcx,xmm0,14
        add dword ptr [rsi+rcx*4],1     ;count pixel 14
        vpextrb rdx,xmm0,15
        add dword ptr [rdi+rdx*4],1     ;count pixel 15

; Process pixels 16 - 19
        vpextrb rax,xmm1,0
        add dword ptr [rsi+rax*4],1     ;count pixel 16
        vpextrb rbx,xmm1,1
        add dword ptr [rdi+rbx*4],1     ;count pixel 17
        vpextrb rcx,xmm1,2
        add dword ptr [rsi+rcx*4],1     ;count pixel 18
        vpextrb rdx,xmm1,3
        add dword ptr [rdi+rdx*4],1     ;count pixel 19

; Process pixels 20 - 23
        vpextrb rax,xmm1,4
        add dword ptr [rsi+rax*4],1     ;count pixel 20
        vpextrb rbx,xmm1,5
        add dword ptr [rdi+rbx*4],1     ;count pixel 21
        vpextrb rcx,xmm1,6
        add dword ptr [rsi+rcx*4],1     ;count pixel 22
        vpextrb rdx,xmm1,7
        add dword ptr [rdi+rdx*4],1     ;count pixel 23

; Process pixels 24 - 27
        vpextrb rax,xmm1,8
        add dword ptr [rsi+rax*4],1     ;count pixel 24
        vpextrb rbx,xmm1,9
        add dword ptr [rdi+rbx*4],1     ;count pixel 25
        vpextrb rcx,xmm1,10
        add dword ptr [rsi+rcx*4],1     ;count pixel 26
        vpextrb rdx,xmm1,11
        add dword ptr [rdi+rdx*4],1     ;count pixel 27

; Process pixels 28 - 31
        vpextrb rax,xmm1,12
        add dword ptr [rsi+rax*4],1     ;count pixel 28
        vpextrb rbx,xmm1,13
        add dword ptr [rdi+rbx*4],1     ;count pixel 29
        vpextrb rcx,xmm1,14
        add dword ptr [rsi+rcx*4],1     ;count pixel 30
        vpextrb rdx,xmm1,15
        add dword ptr [rdi+rdx*4],1     ;count pixel 31

        add r9,32                       ;r9  = next pixel block
        sub r8d,1
        jnz @B                          ;repeat loop if not done

; Merge intermediate histograms into final histogram
        mov ecx,32                      ;ecx = num iterations
        xor eax,eax                     ;rax = common offset

@@:     vmovdqa xmm0,xmmword ptr [rsi+rax]      ;load histo counts
        vmovdqa xmm1,xmmword ptr [rsi+rax+16]
        vpaddd xmm0,xmm0,xmmword ptr [rdi+rax]  ;add counts from histo2
        vpaddd xmm1,xmm1,xmmword ptr [rdi+rax+16]
        vmovdqa xmmword ptr [rsi+rax],xmm0      ;save final result
        vmovdqa xmmword ptr [rsi+rax+16],xmm1

        add rax,32
        sub ecx,1
        jnz @B
        mov eax,1                       ;set success return code

Done:  _DeleteFrame rbx,rsi,rdi
        ret
AvxBuildImageHistogram_ endp
        end
;-------------------------------------------------
;####### Ch07_08.asm
        include <MacrosX86-64-AVX.asmh>

; Image threshold data structure (see Ch07_08.h)
ITD                 struct
PbSrc               qword ?
PbMask              qword ?
NumPixels           dword ?
NumMaskedPixels     dword ?
SumMaskedPixels     dword ?
Threshold           byte ?
Pad                 byte 3 dup(?)
MeanMaskedPixels    real8 ?
ITD                 ends

                .const
                align 16
PixelScale      byte 16 dup(80h)        ;uint8 to int8 scale value
CountPixelsMask byte 16 dup(01h)        ;mask to count pixels
R8_MinusOne     real8 -1.0              ;invalid mean value

                .code
                extern IsValid:proc

; extern "C" bool AvxThresholdImage_(ITD* itd);
; Returns:      0 = invalid size or unaligned image buffer, 1 = success

AvxThresholdImage_ proc frame
        _CreateFrame TI_,0,0,rbx
        _EndProlog

; Verify the arguments in the ITD structure
        mov rbx,rcx                     ;copy itd ptr to non-volatile register
        mov ecx,[rbx+ITD.NumPixels]     ;ecx = num_pixels
        mov rdx,[rbx+ITD.PbSrc]         ;rdx = pb_src
        mov r8,[rbx+ITD.PbMask]         ;r8 = pb_mask
        sub rsp,32                      ;allocate home area for IsValid
        call IsValid                    ;validate args
        or al,al
        jz Done                         ;jump if invalid

; Initialize registers for processing loop
        mov ecx,[rbx+ITD.NumPixels]         ;ecx = num_pixels
        shr ecx,6                           ;ecx = number of 64b pixel blocks
        mov rdx,[rbx+ITD.PbSrc]             ;rdx = pb_src
        mov r8,[rbx+ITD.PbMask]             ;r8 = pb_mask

        movzx r9d,byte ptr [rbx+ITD.Threshold]  ;r9d = threshold
        vmovd xmm1,r9d                      ;xmm1[7:0] = threshold
        vpxor xmm0,xmm0,xmm0                ;mask for vpshufb
        vpshufb xmm1,xmm1,xmm0              ;xmm1 = packed threshold

        vmovdqa xmm4,xmmword ptr [PixelScale]   ;packed pixel scale factor
        vpsubb xmm5,xmm1,xmm4               ;scaled threshold

; Create the mask image
@@:     vmovdqa xmm0,xmmword ptr [rdx]  ;original image pixels
        vpsubb xmm1,xmm0,xmm4           ;scaled image pixels
        vpcmpgtb xmm2,xmm1,xmm5         ;mask pixels
        vmovdqa xmmword ptr [r8],xmm2   ;save mask result

        vmovdqa xmm0,xmmword ptr [rdx+16]
        vpsubb xmm1,xmm0,xmm4
        vpcmpgtb xmm2,xmm1,xmm5
        vmovdqa xmmword ptr [r8+16],xmm2

        vmovdqa xmm0,xmmword ptr [rdx+32]
        vpsubb xmm1,xmm0,xmm4
        vpcmpgtb xmm2,xmm1,xmm5
        vmovdqa xmmword ptr [r8+32],xmm2

        vmovdqa xmm0,xmmword ptr [rdx+48]
        vpsubb xmm1,xmm0,xmm4
        vpcmpgtb xmm2,xmm1,xmm5
        vmovdqa xmmword ptr [r8+48],xmm2

        add rdx,64
        add r8,64                       ;update pointers
        sub ecx,1                       ;update counter
        jnz @B                          ;repeat until done

        mov eax,1                       ;set success return code

Done:   _DeleteFrame rbx
        ret
AvxThresholdImage_ endp

; Macro _UpdateBlockSums

_UpdateBlockSums macro disp
        vmovdqa xmm0,xmmword ptr [rdx+disp] ;xmm0 = 16 image pixels
        vmovdqa xmm1,xmmword ptr [r8+disp]  ;xmm1 = 16 mask pixels
        vpand xmm2,xmm1,xmm8            ;xmm2 = 16 mask pixels (0x00 or 0x01)
        vpaddb xmm6,xmm6,xmm2           ;update block num_masked_pixels
        vpand xmm2,xmm0,xmm1            ;zero out unmasked image pixel
        vpunpcklbw xmm3,xmm2,xmm9       ;promote image pixels from byte to word
        vpunpckhbw xmm4,xmm2,xmm9
        vpaddw xmm4,xmm4,xmm3
        vpaddw xmm7,xmm7,xmm4           ;update block sum_mask_pixels
        endm

; extern "C" bool AvxCalcImageMean_(ITD* itd);
; Returns:  0 = invalid image size or unaligned image buffer, 1 = success

AvxCalcImageMean_ proc frame
        _CreateFrame CIM_,0,64,rbx
        _SaveXmmRegs xmm6,xmm7,xmm8,xmm9
        _EndProlog

; Verify the arguments in the ITD structure
        mov rbx,rcx                     ;rbx = itd ptr
        mov ecx,[rbx+ITD.NumPixels]     ;ecx = num_pixels
        mov rdx,[rbx+ITD.PbSrc]         ;rdx = pb_src
        mov r8,[rbx+ITD.PbMask]         ;r8 = pb_mask
        sub rsp,32                      ;allocate home area for IsValid
        call IsValid                    ;validate args
        or al,al
        jz Done                         ;jump if invalid

; Initialize registers for processing loop
        mov ecx,[rbx+ITD.NumPixels]     ;ecx = num_pixels
        shr ecx,6                       ;ecx = number of 64b pixel blocks
        mov rdx,[rbx+ITD.PbSrc]         ;rdx = pb_src
        mov r8,[rbx+ITD.PbMask]         ;r8 = pb_mask

        vmovdqa xmm8,xmmword ptr [CountPixelsMask]  ;mask for counting pixels
        vpxor xmm9,xmm9,xmm9                    ;xmm9 = packed zero

        xor r10d,r10d                   ;r10d = num_masked_pixels (1 dword)
        vpxor xmm5,xmm5,xmm5            ;sum_masked_pixels (4 dwords)
;Calculate num_mask_pixels and sum_mask_pixels
LP1:    vpxor xmm6,xmm6,xmm6            ;num_masked_pixels_tmp (16 byte values)
        vpxor xmm7,xmm7,xmm7            ;sum_masked_pixels_tmp (8 word values)
        _UpdateBlockSums 0
        _UpdateBlockSums 16
        _UpdateBlockSums 32
        _UpdateBlockSums 48

; Update num_masked_pixels
        vpsrldq xmm0,xmm6,8
        vpaddb xmm6,xmm6,xmm0           ;num_mask_pixels_tmp (8 byte vals)
        vpsrldq xmm0,xmm6,4
        vpaddb xmm6,xmm6,xmm0           ;num_mask_pixels_tmp (4 byte vals)
        vpsrldq xmm0,xmm6,2
        vpaddb xmm6,xmm6,xmm0           ;num_mask_pixels_tmp (2 byte vals)
        vpsrldq xmm0,xmm6,1
        vpaddb xmm6,xmm6,xmm0           ;num_mask_pixels_tmp (1 byte val)
        vpextrb eax,xmm6,0
        add r10d,eax                    ;num_mask_pixels += num_mask_pixels_tmp

; Update sum_masked_pixels
        vpunpcklwd xmm0,xmm7,xmm9       ;promote sum_mask_pixels_tmp to dwords
        vpunpckhwd xmm1,xmm7,xmm9
        vpaddd xmm5,xmm5,xmm0
        vpaddd xmm5,xmm5,xmm1           ;sum_mask_pixels += sum_masked_pixels_tmp

        add rdx,64                      ;update pb_src pointer
        add r8,64                       ;update pb_mask pointer

        sub rcx,1                       ;update loop counter
        jnz LP1                         ;repeat if not done

; Compute mean of masked pixels
        vphaddd xmm0,xmm5,xmm5
        vphaddd xmm1,xmm0,xmm0
        vmovd eax,xmm1                  ;eax = final sum_mask_pixels

        test r10d,r10d                  ;is num_mask_pixels zero?
        jz NoMean                       ;if yes, skip calc of mean
        vcvtsi2sd xmm0,xmm0,eax         ;xmm0 = sum_masked_pixels
        vcvtsi2sd xmm1,xmm1,r10d        ;xmm1 = num_masked_pixels
        vdivsd xmm2,xmm0,xmm1           ;xmm2 = mean_masked_pixels
        jmp @F

NoMean: vmovsd xmm2,[R8_MinusOne]           ;use -1.0 for no mean

@@:     mov [rbx+ITD.SumMaskedPixels],eax   ;save sum_masked_pixels
        mov [rbx+ITD.NumMaskedPixels],r10d  ;save num_masked_pixels
        vmovsd [rbx+ITD.MeanMaskedPixels],xmm2  ;save mean
        mov eax,1                           ;set success return code

Done:   _RestoreXmmRegs xmm6,xmm7,xmm8,xmm9
        _DeleteFrame rbx
        ret
AvxCalcImageMean_  endp
        end
;-------------------------------------------------
;####### Ch09_01.asm
;-------------------------------------------------
; Mask values used to calculate floating-point absolute values
            .const
AbsMaskF32  dword 8 dup(7fffffffh)
AbsMaskF64  qword 4 dup(7fffffffffffffffh)
; extern "C" void AvxPackedMathF32_(const YmmVal& a, const YmmVal& b, YmmVal c[8]);

            .code
AvxPackedMathF32_ proc

; Load packed SP floating-point values
        vmovaps ymm0,ymmword ptr [rcx]  ;ymm0 = *a
        vmovaps ymm1,ymmword ptr [rdx]  ;ymm1 = *b

; Packed SP floating-point addition
        vaddps ymm2,ymm0,ymm1
        vmovaps ymmword ptr [r8],ymm2
```

```
; Packed SP floating-point subtraction
        vsubps ymm2,ymm0,ymm1
        vmovaps ymmword ptr [r8+32],ymm2

; Packed SP floating-point multiplication
        vmulps ymm2,ymm0,ymm1
        vmovaps ymmword ptr [r8+64],ymm2

; Packed SP floating-point division
        vdivps ymm2,ymm0,ymm1
        vmovaps ymmword ptr [r8+96],ymm2

; Packed SP floating-point absolute value (b)
        vandps ymm2,ymm1,ymmword ptr [AbsMaskF32]
        vmovaps ymmword ptr [r8+128],ymm2

; Packed SP floating-point square root (a)
        vsqrtps ymm2,ymm0
        vmovaps ymmword ptr [r8+160],ymm2

; Packed SP floating-point minimum
        vminps ymm2,ymm0,ymm1
        vmovaps ymmword ptr [r8+192],ymm2

; Packed SP floating-point maximum
        vmaxps ymm2,ymm0,ymm1
        vmovaps ymmword ptr [r8+224],ymm2

        vzeroupper
        ret
AvxPackedMathF32_ endp

; extern "C" void AvxPackedMathF64_(const YmmVal& a, const YmmVal& b, YmmVal c[8]);

AvxPackedMathF64_ proc

; Load packed DP floating-point values
        vmovapd ymm0,ymmword ptr [rcx]   ;ymm0 = *a
        vmovapd ymm1,ymmword ptr [rdx]   ;ymm1 = *b

; Packed DP floating-point addition
        vaddpd ymm2,ymm0,ymm1
        vmovapd ymmword ptr [r8],ymm2

; Packed DP floating-point subtraction
        vsubpd ymm2,ymm0,ymm1
        vmovapd ymmword ptr [r8+32],ymm2

; Packed DP floating-point multiplication
        vmulpd ymm2,ymm0,ymm1
        vmovapd ymmword ptr [r8+64],ymm2

; Packed DP floating-point division
        vdivpd ymm2,ymm0,ymm1
        vmovapd ymmword ptr [r8+96],ymm2

; Packed DP floating-point absolute value (b)
        vandpd ymm2,ymm1,ymmword ptr [AbsMaskF64]
        vmovapd ymmword ptr [r8+128],ymm2

; Packed DP floating-point square root (a)
        vsqrtpd ymm2,ymm0
        vmovapd ymmword ptr [r8+160],ymm2

; Packed DP floating-point minimum
        vminpd ymm2,ymm0,ymm1
        vmovapd ymmword ptr [r8+192],ymm2
```

```
; Packed DP floating-point maximum
        vmaxpd ymm2,ymm0,ymm1
        vmovapd ymmword ptr [r8+224],ymm2

        vzeroupper
        ret
AvxPackedMathF64_ endp
        end
;-------------------------------------------------
;####### Ch09_02.asm
        include <cmpequ.asmh>        include <MacrosX86-64-AVX.asmh>

        .const
r4_3p0  real4 3.0
r4_4p0  real4 4.0

        extern c_PI_F32:real4
        extern c_QNaN_F32:real4

; extern "C" void AvxCalcSphereAreaVolume_(float* sa, float* vol, const float* r, size_t n);

        .code
AvxCalcSphereAreaVolume_ proc frame
        _CreateFrame CC_,0,64
        _SaveXmmRegs xmm6,xmm7,xmm8,xmm9
        _EndProlog

; Initialize
        vbroadcastss ymm0,real4 ptr [r4_4p0]    ;packed 4.0
        vbroadcastss ymm1,real4 ptr [c_PI_F32]  ;packed PI
        vmulps ymm6,ymm0,ymm1                   ;packed 4.0 * PI
        vbroadcastss ymm7,real4 ptr [r4_3p0]    ;packed 3.0
        vbroadcastss ymm8,real4 ptr [c_QNaN_F32]    ;packed QNaN
        vxorps ymm9,ymm9,ymm9                   ;packed 0.0

        xor eax,eax                         ;common offset for arrays

        cmp r9,8
        jb FinalR                           ;skip main loop if n < 8

; Calculate surface area and volume values using packed arithmetic
@@:     vmovdqa ymm0,ymmword ptr [r8+rax]   ;load next 8 radii
        vmulps ymm2,ymm6,ymm0               ;4.0 * PI * r
        vmulps ymm3,ymm2,ymm0               ;4.0 * PI * r * r

        vcmpps ymm1,ymm0,ymm9,CMP_LT        ;ymm1 = mask of radii < 0.0

        vandps ymm4,ymm1,ymm8               ;set surface area to QNaN for radii < 0.0
        vandnps ymm5,ymm1,ymm3              ;keep surface area for radii >= 0.0
        vorps ymm5,ymm4,ymm5                ;final packed surface area
        vmovaps ymmword ptr[rcx+rax],ymm5   ;save packed surface area

        vmulps ymm2,ymm3,ymm0               ;4.0 * PI * r * r * r
        vdivps ymm3,ymm2,ymm7               ;4.0 * PI * r * r * r / 3.0
        vandps ymm4,ymm1,ymm8               ;set volume to QNaN for radii < 0.0
        vandnps ymm5,ymm1,ymm3              ;keep volume for radii >= 0.0
        vorps ymm5,ymm4,ymm5                ;final packed volume
        vmovaps ymmword ptr[rdx+rax],ymm5   ;save packed volume

        add rax,32                          ;rax = offset to next set of radii
        sub r9,8
        cmp r9,8
        jae @B                              ;repeat until n < 8

; Perform final calculations using scalar arithmetic
FinalR: test r9,r9
```

```
        jz Done                             ;skip loop of no more elements

@@:     vmovss xmm0,real4 ptr [r8+rax]
        vmulss xmm2,xmm6,xmm0               ;4.0 * PI * r
        vmulss xmm3,xmm2,xmm0               ;4.0 * PI * r * r

        vcmpss xmm1,xmm0,xmm9,CMP_LT

        vandps xmm4,xmm1,xmm8
        vandnps xmm5,xmm1,xmm3
        vorps xmm5,xmm4,xmm5
        vmovss real4 ptr[rcx+rax],xmm5      ;save surface area

        vmulss xmm2,xmm3,xmm0               ;4.0 * PI * r * r * r
        vdivss xmm3,xmm2,xmm7               ;4.0 * PI * r * r * r / 3.0
        vandps xmm4,xmm1,xmm8
        vandnps xmm5,xmm1,xmm3
        vorps xmm5,xmm4,xmm5
        vmovss real4 ptr[rdx+rax],xmm5      ;save volume

        add rax,4
        dec r9
        jnz @B                              ;repeat until done

Done:   vzeroupper

        _RestoreXmmRegs xmm6,xmm7,xmm8,xmm9
        _DeleteFrame
        ret
AvxCalcSphereAreaVolume_ endp
        end
;-------------------------------------------------
;####### Ch09_03.asm
;-------------------------------------------------
; extern "C" bool AvxCalcColMeans_(const double* x, size_t nrows, size_t ncols, double* col_means)
        extern c_NumRowsMax:qword
        extern c_NumColsMax:qword

        .code
AvxCalcColumnMeans_ proc

; Validate nrows and ncols
        xor eax,eax                     ;error return code (also col_mean index)
        test rdx,rdx
        jz Done                         ;jump if nrows is zero
        cmp rdx,[c_NumRowsMax]
        ja Done                         ;jump if nrows is too large
        test r8,r8
        jz Done                         ;jump if ncols is zero
        cmp r8,[c_NumColsMax]
        ja Done                         ;jump if ncols is too large

; Initialize elements of col_means to zero
        vxorpd xmm0,xmm0,xmm0           ;xmm0[63:0] = 0.0
@@:     vmovsd real8 ptr[r9+rax*8],xmm0 ;col_means[i] = 0.0
        inc rax
        cmp rax,r8
        jb @B                           ;repeat until done

        vcvtsi2sd xmm2,xmm2,rdx         ;convert nrows for later use

; Compute the sum of each column in x
LP1:    mov r11,r9                      ;r11 = ptr to col_means
        xor r10,r10                     ;r10 = col_index

LP2:    mov rax,r10                     ;rax = col_index
        add rax,4
```

```
        cmp rax,r8                      ;4 or more columns remaining?
        ja @F                           ;jump if no (col_index + 4 > ncols)
; Update col_means using next four columns
        vmovupd ymm0,ymmword ptr [rcx]  ;load next 4 columns of current row
        vaddpd ymm1,ymm0,ymmword ptr [r11]  ;add to col_means
        vmovupd ymmword ptr [r11],ymm1  ;save updated col_means
        add r10,4                       ;col_index += 4
        add rcx,32                      ;update x ptr
        add r11,32                      ;update col_means ptr
        jmp NextColSet

@@:     sub rax,2
        cmp rax,r8                      ;2 or more columns remaining?
        ja @F                           ;jump if no (col_index + 2 > ncols)
; Update col_means using next two columns
        vmovupd xmm0,xmmword ptr [rcx]  ;load next 2 columns of current row
        vaddpd xmm1,xmm0,xmmword ptr [r11]  ;add to col_means
        vmovupd xmmword ptr [r11],xmm1  ;save updated col_means
        add r10,2                       ;col_index += 2
        add rcx,16                      ;update x ptr
        add r11,16                      ;update col_means ptr
        jmp NextColSet

; Update col_means using next column (or last column in the current row)
@@:     vmovsd xmm0,real8 ptr [rcx]     ;load x from last column
        vaddsd xmm1,xmm0,real8 ptr [r11]    ;add to col_means
        vmovsd real8 ptr [r11],xmm1     ;save updated col_means
        inc r10                         ;col_index += 1
        add rcx,8                       ;update x ptr

NextColSet:
        cmp r10,r8                      ;more columns in current row?
        jb LP2                          ;jump if yes
        dec rdx                         ;nrows -= 1
        jnz LP1                         ;jump if more rows

; Compute the final col_means
@@:     vmovsd xmm0,real8 ptr [r9]      ;xmm0 = col_means[i]
        vdivsd xmm1,xmm0,xmm2           ;compute final mean
        vmovsd real8 ptr [r9],xmm1      ;save col_mean[i]
        add r9,8                        ;update col_means ptr
        dec r8                          ;ncols -= 1
        jnz @B                          ;repeat until done

        mov eax,1                       ;set success return code

Done:   vzeroupper
        ret

AvxCalcColumnMeans_ endp
        end
;-------------------------------------------------
;####### Ch09_04.asm
        include <MacrosX86-64-AVX.asmh>

;extern"C" bool AvxCalcCorrCoef(const double* x,const double* y,size_t n, double sums[5], double epsilon, double* rho)
; Returns       0 = error, 1 = success

        .code
AvxCalcCorrCoef_ proc frame
        _CreateFrame CC_,0,32
        _SaveXmmRegs xmm6,xmm7
        _EndProlog

; Validate arguments
        or r8,r8
        jz BadArg                       ;jump if n == 0
        test rcx,1fh
        jnz BadArg                      ;jump if x is not aligned
        test rdx,1fh
        jnz BadArg                      ;jump if y is not aligned

; Initialize sum variables to zero
        vxorpd ymm3,ymm3,ymm3           ;ymm3 = packed sum_x
        vxorpd ymm4,ymm4,ymm4           ;ymm4 = packed sum_y
        vxorpd ymm5,ymm5,ymm5           ;ymm5 = packed sum_xx
        vxorpd ymm6,ymm6,ymm6           ;ymm6 = packed sum_yy
        vxorpd ymm7,ymm7,ymm7           ;ymm7 = packed sum_xy
        mov r10,r8                      ;r10 = n

        cmp r8,4
        jb LP2                          ;jump if n >= 1 && n <= 3

; Calculate intermediate packed sum variables
LP1:    vmovapd ymm0,ymmword ptr [rcx]  ;ymm0 = packed x values
        vmovapd ymm1,ymmword ptr [rdx]  ;ymm1 = packed y values

        vaddpd ymm3,ymm3,ymm0           ;update packed sum_x
        vaddpd ymm4,ymm4,ymm1           ;update packed sum_y

        vmulpd ymm2,ymm0,ymm1           ;ymm2 = packed xy values
        vaddpd ymm7,ymm7,ymm2           ;update packed sum_xy

        vmulpd ymm0,ymm0,ymm0           ;ymm0 = packed xx values
        vmulpd ymm1,ymm1,ymm1           ;ymm1 = packed yy values
        vaddpd ymm5,ymm5,ymm0           ;update packed sum_xx
        vaddpd ymm6,ymm6,ymm1           ;update packed sum_yy

        add rcx,32                      ;update x ptr
        add rdx,32                      ;update y ptr
        sub r8,4                        ;n -= 4
        cmp r8,4                        ;is n >= 4?
        jae LP1                         ;jump if yes

        or r8,r8                        ;is n == 0?
        jz FSV                          ;jump if yes

; Update sum variables with final x & y values
LP2:    vmovsd xmm0,real8 ptr [rcx]     ;xmm0[63:0] = x[i], ymm0[255:64] = 0
        vmovsd xmm1,real8 ptr [rdx]     ;xmm1[63:0] = y[i], ymm1[255:64] = 0

        vaddpd ymm3,ymm3,ymm0           ;update packed sum_x
        vaddpd ymm4,ymm4,ymm1           ;update packed sum_y

        vmulpd ymm2,ymm0,ymm1           ;ymm2 = packed xy values
        vaddpd ymm7,ymm7,ymm2           ;update packed sum_xy

        vmulpd ymm0,ymm0,ymm0           ;ymm0 = packed xx values
        vmulpd ymm1,ymm1,ymm1           ;ymm1 = packed yy values
        vaddpd ymm5,ymm5,ymm0           ;update packed sum_xx
        vaddpd ymm6,ymm6,ymm1           ;update packed sum_yy

        add rcx,8                       ;update x ptr
        add rdx,8                       ;update y ptr
        sub r8,1                        ;n -= 1
        jnz LP2                         ;repeat until done

; Calculate final sum variables
FSV:    vextractf128 xmm0,ymm3,1
        vaddpd xmm1,xmm0,xmm3
        vhaddpd xmm3,xmm1,xmm1          ;xmm3[63:0] = sum_x

        vextractf128 xmm0,ymm4,1
        vaddpd xmm1,xmm0,xmm4
```

```
        vhaddpd xmm4,xmm1,xmm1          ;xmm4[63:0] = sum_y

        vextractf128 xmm0,ymm5,1
        vaddpd xmm1,xmm0,xmm5
        vhaddpd xmm5,xmm1,xmm1          ;xmm5[63:0] = sum_xx

        vextractf128 xmm0,ymm6,1
        vaddpd xmm1,xmm0,xmm6
        vhaddpd xmm6,xmm1,xmm1          ;xmm6[63:0] = sum_yy

        vextractf128 xmm0,ymm7,1
        vaddpd xmm1,xmm0,xmm7
        vhaddpd xmm7,xmm1,xmm1          ;xmm7[63:0] = sum_xy

; Save final sum variables
        vmovsd real8 ptr [r9],xmm3      ;save sum_x
        vmovsd real8 ptr [r9+8],xmm4    ;save sum_y
        vmovsd real8 ptr [r9+16],xmm5   ;save sum_xx
        vmovsd real8 ptr [r9+24],xmm6   ;save sum_yy
        vmovsd real8 ptr [r9+32],xmm7   ;save sum_xy

; Calculate rho numerator
; rho_num = n * sum_xy - sum_x * sum_y;
        vcvtsi2sd xmm2,xmm2,r10         ;xmm2 = n
        vmulsd xmm0,xmm2,xmm7           ;xmm0 = = n * sum_xy
        vmulsd xmm1,xmm3,xmm4           ;xmm1 = sum_x * sum_y
        vsubsd xmm7,xmm0,xmm1           ;xmm7 = rho_num

; Calculate rho denominator
; t1 = sqrt(n * sum_xx - sum_x * sum_x)
; t2 = sqrt(n * sum_yy - sum_y * sum_y)
; rho_den = t1 * t2
        vmulsd xmm0,xmm2,xmm5   ;xmm0 = n * sum_xx
        vmulsd xmm3,xmm3,xmm3   ;xmm3 = sum_x * sum_x
        vsubsd xmm3,xmm0,xmm3   ;xmm3 = n * sum_xx - sum_x * sum_x
        vsqrtsd xmm3,xmm3,xmm3  ;xmm3 = t1

        vmulsd xmm0,xmm2,xmm6   ;xmm0 = n * sum_yy
        vmulsd xmm4,xmm4,xmm4   ;xmm4 = sum_y * sum_y
        vsubsd xmm4,xmm0,xmm4   ;xmm4 = n * sum_yy - sum_y * sum_y
        vsqrtsd xmm4,xmm4,xmm4  ;xmm4 = t2

        vmulsd xmm0,xmm3,xmm4   ;xmm0 = rho_den

; Calculate and save final rho
        xor eax,eax
        vcomisd xmm0,real8 ptr [rbp+CC_OffsetStackArgs] ;rho_den < epsilon?
        setae al                            ;set return code
        jb BadRho                           ;jump if rho_den < epsilon
        vdivsd xmm1,xmm7,xmm0               ;xmm1 = rho

SavRho: mov rdx,[rbp+CC_OffsetStackArgs+8]  ;rdx = ptr to rho
        vmovsd real8 ptr [rdx],xmm1         ;save rho

Done:   vzeroupper
        _RestoreXmmRegs xmm6,xmm7
        _DeleteFrame
        ret

; Error handling code
BadRho: vxorpd xmm1,xmm1,xmm1           ;rho = 0
        jmp SavRho

BadArg: xor eax,eax                     ;eax = invalid arg ret code
        jmp Done

AvxCalcCorrCoef_ endp
```

```
        end
;-------------------------------------------------
;####### Ch09_05.asm
        include <MacrosX86-64-AVX.asmh>

; _Mat4x4TransposeF64 macro
; Description:  This macro computes the transpose of a 4x4
;               double-precision floating-point matrix.
;  Input Matrix                    Output Matrtix
;  ---------------------------------------------------
;  ymm0    a3 a2 a1 a0             ymm0    d0 c0 b0 a0
;  ymm1    b3 b2 b1 b0             ymm1    d1 c1 b1 a1
;  ymm2    c3 c2 c1 c0             ymm2    d2 c2 b2 a2
;  ymm3    d3 d2 d1 d0             ymm3    d3 c3 b3 a3

_Mat4x4TransposeF64 macro
        vunpcklpd ymm4,ymm0,ymm1        ;ymm4 = b2 a2 b0 a0
        vunpckhpd ymm5,ymm0,ymm1        ;ymm5 = b3 a3 b1 a1
        vunpcklpd ymm6,ymm2,ymm3        ;ymm6 = d2 c2 d0 c0
        vunpckhpd ymm7,ymm2,ymm3        ;ymm7 = d3 c3 d1 c1

        vperm2f128 ymm0,ymm4,ymm6,20h   ;ymm0 = d0 c0 b0 a0
        vperm2f128 ymm1,ymm5,ymm7,20h   ;ymm1 = d1 c1 b1 a1
        vperm2f128 ymm2,ymm4,ymm6,31h   ;ymm2 = d2 c2 b2 a2
        vperm2f128 ymm3,ymm5,ymm7,31h   ;ymm3 = d3 c3 b3 a3
        endm

; extern "C" void AvxMat4x4TransposeF64_(double* m_des, const double* m_src1)
        .code
AvxMat4x4TransposeF64_ proc frame
        _CreateFrame MT_,0,32
        _SaveXmmRegs xmm6,xmm7
        _EndProlog

; Transpose matrix m_src1
        vmovaps ymm0,[rdx]              ;ymm0 = m_src1.row_0
        vmovaps ymm1,[rdx+32]           ;ymm1 = m_src2.row_1
        vmovaps ymm2,[rdx+64]           ;ymm2 = m_src3.row_2
        vmovaps ymm3,[rdx+96]           ;ymm3 = m_src4.row_3

        _Mat4x4TransposeF64

        vmovaps [rcx],ymm0              ;save m_des.row_0
        vmovaps [rcx+32],ymm1           ;save m_des.row_1
        vmovaps [rcx+64],ymm2           ;save m_des.row_2
        vmovaps [rcx+96],ymm3           ;save m_des.row_3

        vzeroupper
Done:   _RestoreXmmRegs xmm6,xmm7
        _DeleteFrame
        ret
AvxMat4x4TransposeF64_ endp

; _Mat4x4MulCalcRowF64 macro
; Description:  This macro computes one row of a 4x4 matrix multiplication.
; Registers:    ymm0 = m_src2.row0
;               ymm1 = m_src2.row1
;               ymm2 = m_src2.row2
;               ymm3 = m_src2.row3
;               rcx = m_des ptr
;               rdx = m_src1 ptr
;               ymm4 - ymm4 = scratch registers

_Mat4x4MulCalcRowF64 macro disp
        vbroadcastsd ymm4,real8 ptr [rdx+disp]  ;broadcast m_src1[i][0]
        vbroadcastsd ymm5,real8 ptr [rdx+disp+8]    ;broadcast m_src1[i][1]
        vbroadcastsd ymm6,real8 ptr [rdx+disp+16]   ;broadcast m_src1[i][2]
        vbroadcastsd ymm7,real8 ptr [rdx+disp+24]   ;broadcast m_src1[i][3]

        vmulpd ymm4,ymm4,ymm0                   ;m_src1[i][0] * m_src2.row_0
        vmulpd ymm5,ymm5,ymm1                   ;m_src1[i][1] * m_src2.row_1
        vmulpd ymm6,ymm6,ymm2                   ;m_src1[i][2] * m_src2.row_2
        vmulpd ymm7,ymm7,ymm3                   ;m_src1[i][3] * m_src2.row_3

        vaddpd ymm4,ymm4,ymm5                   ;calc m_des.row_i
        vaddpd ymm6,ymm6,ymm7
        vaddpd ymm4,ymm4,ymm6

        vmovapd [rcx+disp],ymm4                 ;save m_des.row_i
        endm

; extern "C" void AvxMat4x4MulF64_(double* m_des, const double* m_src1, const double* m_src2)
AvxMat4x4MulF64_ proc frame
        _CreateFrame MM_,0,32
        _SaveXmmRegs xmm6,xmm7
        _EndProlog

; Load m_src2 into YMM3:YMM0
        vmovapd ymm0,[r8]               ;ymm0 = m_src2.row_0
        vmovapd ymm1,[r8+32]            ;ymm1 = m_src2.row_1
        vmovapd ymm2,[r8+64]            ;ymm2 = m_src2.row_2
        vmovapd ymm3,[r8+96]            ;ymm3 = m_src2.row_3

; Compute matrix product
        _Mat4x4MulCalcRowF64 0          ;calculate m_des.row_0
        _Mat4x4MulCalcRowF64 32         ;calculate m_des.row_1
        _Mat4x4MulCalcRowF64 64         ;calculate m_des.row_2
        _Mat4x4MulCalcRowF64 96         ;calculate m_des.row_3

        vzeroupper
Done:   _RestoreXmmRegs xmm6,xmm7
        _DeleteFrame
        ret
AvxMat4x4MulF64_ endp
        end
;-------------------------------------------------
;####### Ch09_06.asm
        include <MacrosX86-64-AVX.asmh>

; Custom segment for constants
ConstVals segment readonly align(32) 'const'
Mat4x4I real8 1.0, 0.0, 0.0, 0.0
        real8 0.0, 1.0, 0.0, 0.0
        real8 0.0, 0.0, 1.0, 0.0
        real8 0.0, 0.0, 0.0, 1.0

r8_SignBitMask  qword 4 dup (8000000000000000h)
r8_AbsMask      qword 4 dup (7fffffffffffffffh)
r8_1p0          real8 1.0
r8_N1p0         real8 -1.0
r8_N0p5         real8 -0.5
r8_N0p3333      real8 -0.33333333333333
r8_N0p25        real8 -0.25
ConstVals ends
        .code

; _Mat4x4TraceF64 macro
; Description:  This macro contains instructions that compute the trace
;               of the 4x4 double-precision floating-point matrix in ymm3:ymm0.

_Max4x4TraceF64 macro
        vblendpd ymm0,ymm0,ymm1,00000010b   ;ymm0[127:0] = row 1,0 diag vals
        vblendpd ymm1,ymm2,ymm3,00001000b   ;ymm1[255:128] = row 3,2 diag vals
        vperm2f128 ymm2,ymm1,ymm1,00000001b ;ymm2[127:0] = row 3,2 diag vals
        vaddpd ymm3,ymm0,ymm2
        vhaddpd ymm0,ymm3,ymm3              ;xmm0[63:0] = trace
        endm

; extern "C" double Avx2Mat4x4TraceF64_(const double* m_src1)
; Description:  The following function computes the trace of a
;               4x4 double-precision floating-point array.

Avx2Mat4x4TraceF64_ proc
            vmovapd ymm0,[rcx]          ;ymm0 = m_src1.row_0
            vmovapd ymm1,[rcx+32]       ;ymm1 = m_src1.row_1
            vmovapd ymm2,[rcx+64]       ;ymm2 = m_src1.row_2
            vmovapd ymm3,[rcx+96]       ;ymm3 = m_src1.row_3

            _Max4x4TraceF64             ;xmm0[63:0] = m_src1.trace()
            vzeroupper
            ret
Avx2Mat4x4TraceF64_ endp

; _Mat4x4MulCalcRowF64 macro
; Description:  This macro is used to compute one row of a 4x4 matrix
;               multiply.
; Registers:    ymm0 = m_src2.row0
;               ymm1 = m_src2.row1
;               ymm2 = m_src2.row2
;               ymm3 = m_src2.row3
;               ymm4 - ymm7 = scratch registers

_Mat4x4MulCalcRowF64 macro dreg,sreg,disp
        vbroadcastsd ymm4,real8 ptr [sreg+disp] ;broadcast m_src1[i][0]
        vbroadcastsd ymm5,real8 ptr [sreg+disp+8]   ;broadcast m_src1[i][1]
        vbroadcastsd ymm6,real8 ptr [sreg+disp+16]  ;broadcast m_src1[i][2]
        vbroadcastsd ymm7,real8 ptr [sreg+disp+24]  ;broadcast m_src1[i][3]

        vmulpd ymm4,ymm4,ymm0                   ;m_src1[i][0] * m_src2.row_0
        vmulpd ymm5,ymm5,ymm1                   ;m_src1[i][1] * m_src2.row_1
        vmulpd ymm6,ymm6,ymm2                   ;m_src1[i][2] * m_src2.row_2
        vmulpd ymm7,ymm7,ymm3                   ;m_src1[i][3] * m_src2.row_3

        vaddpd ymm4,ymm4,ymm5                   ;calc m_des.row_i
        vaddpd ymm6,ymm6,ymm7
        vaddpd ymm4,ymm4,ymm6
        vmovapd[dreg+disp],ymm4                 ;save m_des.row_i
        endm

; extern "C" void Avx2Mat4x4MulF64_(double* m_des, const double* m_src1, const double* m_src2)
Avx2Mat4x4MulF64_ proc frame
        _CreateFrame MM_,0,32
        _SaveXmmRegs xmm6,xmm7
        _EndProlog

        vmovapd ymm0,[r8]               ;ymm0 = m_src2.row_0
        vmovapd ymm1,[r8+32]            ;ymm1 = m_src2.row_1
        vmovapd ymm2,[r8+64]            ;ymm2 = m_src2.row_2
        vmovapd ymm3,[r8+96]            ;ymm3 = m_src2.row_3

        _Mat4x4MulCalcRowF64 rcx,rdx,0  ;calculate m_des.row_0
        _Mat4x4MulCalcRowF64 rcx,rdx,32 ;calculate m_des.row_1
        _Mat4x4MulCalcRowF64 rcx,rdx,64 ;calculate m_des.row_2
        _Mat4x4MulCalcRowF64 rcx,rdx,96 ;calculate m_des.row_3

        vzeroupper
        _RestoreXmmRegs xmm6,xmm7
        _DeleteFrame
        ret
Avx2Mat4x4MulF64_ endp
```

```
; extern "C" bool Avx2Mat4x4InvF64_(double* m_inv, const double* m, double epsilon, bool* is_singular);

; Offsets of intermediate matrices on stack relative to rsp
OffsetM2 equ 32
OffsetM3 equ 160
OffsetM4 equ 288

Avx2Mat4x4InvF64_ proc frame
        _CreateFrame MI_,0,160
        _SaveXmmRegs xmm6,xmm7,xmm8,xmm9,xmm10,xmm11,xmm12,xmm13,xmm14,xmm15
        _EndProlog

; Save args to home area for later use
        mov qword ptr [rbp+MI_OffsetHomeRCX],rcx    ;save m_inv ptr
        mov qword ptr [rbp+MI_OffsetHomeRDX],rdx    ;save m ptr
        vmovsd real8 ptr [rbp+MI_OffsetHomeR8],xmm2 ;save epsilon
        mov qword ptr [rbp+MI_OffsetHomeR9],r9      ;save is_singular ptr

; Allocate 384 bytes of stack space for temp matrices + 32 bytes for function calls
        and rsp,0ffffffe0h              ;align rsp to 32-byte boundary
        sub rsp,416                     ;alloc stack space

; Calculate m2
        lea rcx,[rsp+OffsetM2]          ;rcx = m2 ptr
        mov r8,rdx                      ;rdx, r8 = m ptr
        call Avx2Mat4x4MulF64_          ;calculate and save m2

; Calculate m3
        lea rcx,[rsp+OffsetM3]          ;rcx = m3 ptr
        lea rdx,[rsp+OffsetM2]          ;rdx = m2 ptr
        mov r8,[rbp+MI_OffsetHomeRDX]   ;r8 = m
        call Avx2Mat4x4MulF64_          ;calculate and save m3

; Calculate m4
        lea rcx,[rsp+OffsetM4]          ;rcx = m4 ptr
        lea rdx,[rsp+OffsetM3]          ;rdx = m3 ptr
        mov r8,[rbp+MI_OffsetHomeRDX]   ;r8 = m
        call Avx2Mat4x4MulF64_          ;calculate and save m4

; Calculate trace of m, m2, m3, and m4
        mov rcx,[rbp+MI_OffsetHomeRDX]
        call Avx2Mat4x4TraceF64_
        vmovsd xmm8,xmm8,xmm0           ;xmm8 = t1

        lea rcx,[rsp+OffsetM2]
        call Avx2Mat4x4TraceF64_
        vmovsd xmm9,xmm9,xmm0           ;xmm9 = t2

        lea rcx,[rsp+OffsetM3]
        call Avx2Mat4x4TraceF64_
        vmovsd xmm10,xmm10,xmm0         ;xmm10 = t3

        lea rcx,[rsp+OffsetM4]
        call Avx2Mat4x4TraceF64_
        vmovsd xmm11,xmm11,xmm0         ;xmm10 = t4

; Calculate the required coefficients
; c1 = -t1;
; c2 = -1.0f / 2.0f * (c1 * t1 + t2);
; c3 = -1.0f / 3.0f * (c2 * t1 + c1 * t2 + t3);
; c4 = -1.0f / 4.0f * (c3 * t1 + c2 * t2 + c1 * t3 + t4);
; Registers used:
;   t1-t4 = xmm8-xmm11
;   c1-c4 = xmm12-xmm15

        vxorpd xmm12,xmm8,real8 ptr [r8_SignBitMask]    ;xmm12 = c1
        vmulsd xmm13,xmm12,xmm8     ;c1 * t1
        vaddsd xmm13,xmm13,xmm9     ;c1 * t1 + t2
        vmulsd xmm13,xmm13,[r8_N0p5]    ;c2

        vmulsd xmm14,xmm13,xmm8     ;c2 * t1
        vmulsd xmm0,xmm12,xmm9      ;c1 * t2
        vaddsd xmm14,xmm14,xmm0     ;c2 * t1 + c1 * t2
        vaddsd xmm14,xmm14,xmm10    ;c2 * t1 + c1 * t2 + t3
        vmulsd xmm14,xmm14,[r8_N0p3333] ;c3

        vmulsd xmm15,xmm14,xmm8     ;c3 * t1
        vmulsd xmm0,xmm13,xmm9      ;c2 * t2
        vmulsd xmm1,xmm12,xmm10     ;c1 * t3
        vaddsd xmm2,xmm0,xmm1       ;c2 * t2 + c1 * t3
        vaddsd xmm15,xmm15,xmm2     ;c3 * t1 + c2 * t2 + c1 * t3
        vaddsd xmm15,xmm15,xmm11    ;c3 * t1 + c2 * t2 + c1 * t3 + t4
        vmulsd xmm15,xmm15,[r8_N0p25]   ;c4

; Make sure matrix is not singular
        vandpd xmm0,xmm15,[r8_AbsMask]                ;compute fabs(c4)
        vmovsd xmm1,real8 ptr [rbp+MI_OffsetHomeR8]
        vcomisd xmm0,real8 ptr [rbp+MI_OffsetHomeR8]    ;compare against epsilon
        setp al                                       ;set al = if unordered
        setb ah                                       ;set ah = if fabs(c4) < epsilon
        or al,ah                                      ;al = is_singular
        mov rcx,[rbp+MI_OffsetHomeR9]                 ;rax = is_singular ptr
        mov [rcx],al                                  ;save is_singular state
        jnz Error                                     ;jump if singular

; Calculate m_inv = -1.0 / c4 * (m3 + c1 * m2 + c2 * m1 + c3 * I)
        vbroadcastsd ymm14,xmm14                      ;ymm14 = packed c3
        lea rcx,[Mat4x4I]                             ;rcx = I ptr
        vmulpd ymm0,ymm14,ymmword ptr [rcx]
        vmulpd ymm1,ymm14,ymmword ptr [rcx+32]
        vmulpd ymm2,ymm14,ymmword ptr [rcx+64]
        vmulpd ymm3,ymm14,ymmword ptr [rcx+96]        ;c3 * I

        vbroadcastsd ymm13,xmm13                      ;ymm13 = packed c2
        mov rcx,[rbp+MI_OffsetHomeRDX]                ;rcx = m ptr
        vmulpd ymm4,ymm13,ymmword ptr [rcx]
        vmulpd ymm5,ymm13,ymmword ptr [rcx+32]
        vmulpd ymm6,ymm13,ymmword ptr [rcx+64]
        vmulpd ymm7,ymm13,ymmword ptr [rcx+96]        ;c2 * m1
        vaddpd ymm0,ymm0,ymm4
        vaddpd ymm1,ymm1,ymm5
        vaddpd ymm2,ymm2,ymm6
        vaddpd ymm3,ymm3,ymm7                         ;c2 * m1 + c3 * I

        vbroadcastsd ymm12,xmm12                      ;ymm12 = packed c1
        lea rcx,[rsp+OffsetM2]                        ;rcx = m2 ptr
        vmulpd ymm4,ymm12,ymmword ptr [rcx]
        vmulpd ymm5,ymm12,ymmword ptr [rcx+32]
        vmulpd ymm6,ymm12,ymmword ptr [rcx+64]
        vmulpd ymm7,ymm12,ymmword ptr [rcx+96]        ;c1 * m2
        vaddpd ymm0,ymm0,ymm4
        vaddpd ymm1,ymm1,ymm5
        vaddpd ymm2,ymm2,ymm6
        vaddpd ymm3,ymm3,ymm7                         ;c1 * m2 + c2 * m1 + c3 * I

        lea rcx,[rsp+OffsetM3]                        ;rcx = m3 ptr
        vaddpd ymm0,ymm0,ymmword ptr [rcx]
        vaddpd ymm1,ymm1,ymmword ptr [rcx+32]
        vaddpd ymm2,ymm2,ymmword ptr [rcx+64]
        vaddpd ymm3,ymm3,ymmword ptr [rcx+96]         ;m3 + c1 * m2 + c2 * m1 + c3 * I

        vmovsd xmm4,[r8_N1p0]
        vdivsd xmm4,xmm4,xmm15          ;xmm4 = -1.0 / c4
```

```
        vbroadcastsd ymm4,xmm4
        vmulpd ymm0,ymm0,ymm4
        vmulpd ymm1,ymm1,ymm4
        vmulpd ymm2,ymm2,ymm4
        vmulpd ymm3,ymm3,ymm4           ;ymm3:ymm0 = m_inv

; Save m_inv
        mov rcx,[rbp+MI_OffsetHomeRCX]
        vmovapd ymmword ptr [rcx],ymm0
        vmovapd ymmword ptr [rcx+32],ymm1
        vmovapd ymmword ptr [rcx+64],ymm2
        vmovapd ymmword ptr [rcx+96],ymm3
        mov eax,1                       ;set success return code

Done:    vzeroupper
        _RestoreXmmRegs xmm6,xmm7,xmm8,xmm9,xmm10,xmm11,xmm12,xmm13,xmm14,xmm15
        _DeleteFrame
        ret

Error:  xor eax,eax
        jmp Done

Avx2Mat4x4InvF64_ endp
        end
;-------------------------------------------------
;####### Ch09_07.asm
;-------------------------------------------------
; extern "C" void AvxBlendF32_(YmmVal* des1, YmmVal* src1, YmmVal* src2, YmmVal* idx1)
        .code
AvxBlendF32_ proc
        vmovaps ymm0,ymmword ptr [rdx]  ;ymm0 = src1
        vmovaps ymm1,ymmword ptr [r8]   ;ymm1 = src2
        vmovdqa ymm2,ymmword ptr [r9]   ;ymm2 = idx1
        vblendvps ymm3,ymm0,ymm1,ymm2   ;blend ymm0 & ymm1, ymm2 "indices"
        vmovaps ymmword ptr [rcx],ymm3  ;Save result to des1

        vzeroupper
        ret
AvxBlendF32_ endp

; extern "C" void Avx2PermuteF32_(YmmVal* des1, YmmVal* src1, YmmVal* idx1, YmmVal* des2, YmmVal* src2, YmmVal* idx2)
Avx2PermuteF32_ proc

; Perform vpermps permutation
        vmovaps ymm0,ymmword ptr [rdx]  ;ymm0 = src1
        vmovdqa ymm1,ymmword ptr [r8]   ;ymm1 = idx1
        vpermps ymm2,ymm1,ymm0          ;permute ymm0 using ymm1 indices
        vmovaps ymmword ptr [rcx],ymm2  ;save result to des1

; Perform vpermilps permutation
        mov rdx,[rsp+40]                ;rdx = src2 ptr
        mov r8,[rsp+48]                 ;r8 = idx2 ptr
        vmovaps ymm3,ymmword ptr [rdx]  ;ymm3 = src2
        vmovdqa ymm4,ymmword ptr [r8]   ;ymm4 = idx1
        vpermilps ymm5,ymm3,ymm4        ;permute ymm3 using ymm4 indices
        vmovaps ymmword ptr [r9],ymm5   ;save result to des2

        vzeroupper
        ret
Avx2PermuteF32_ endp
        end
;-------------------------------------------------
;####### Ch09_08.asm
;-------------------------------------------------
; For each of the following functions, the contents of y are loaded
; into ymm0 prior to execution of the vgatherXXX instruction in order to
; demonstrate the effects of conditional merging.
```

```
        .code
; extern "C" void Avx2Gather8xF32_I32_(float* y, const float* x, const int32_t* indices, const
int32_t* merge)
Avx2Gather8xF32_I32_ proc
        vmovups ymm0,ymmword ptr [rcx]  ;ymm0 = y[7]:y[0]
        vmovdqu ymm1,ymmword ptr [r8]   ;ymm1 = indices[7]:indices[0]
        vmovdqu ymm2,ymmword ptr [r9]   ;ymm2 = merge[7]:merge[0]
        vpslld ymm2,ymm2,31             ;shift merge vals to high-order bits
        vgatherdps ymm0,[rdx+ymm1*4],ymm2   ;ymm0 = gathered elements
        vmovups ymmword ptr [rcx],ymm0  ;save gathered elements

        vzeroupper
        ret
Avx2Gather8xF32_I32_ endp

; extern "C" void Avx2Gather8xF32_I64_(float* y, const float* x, const int64_t* indices, const
int32_t* merge)
Avx2Gather8xF32_I64_ proc
        vmovups xmm0,xmmword ptr [rcx]  ;xmm0 = y[3]:y[0]
        vmovdqu ymm1,ymmword ptr [r8]   ;ymm1 = indices[3]:indices[0]
        vmovdqu xmm2,xmmword ptr [r9]   ;xmm2 = merge[3]:merge[0]
        vpslld xmm2,xmm2,31             ;shift merge vals to high-order bits
        vgatherqps xmm0,[rdx+ymm1*4],xmm2   ;xmm0 = gathered elements
        vmovups xmmword ptr [rcx],xmm0  ;save gathered elements

        vmovups xmm3,xmmword ptr [rcx+16]   ;xmm0 = des[7]:des[4]
        vmovdqu ymm1,ymmword ptr [r8+32]    ;ymm1 = indices[7]:indices[4]
        vmovdqu xmm2,xmmword ptr [r9+16]    ;xmm2 = merge[7]:merge[4]
        vpslld xmm2,xmm2,31             ;shift merge vals to high-order bits
        vgatherqps xmm3,[rdx+ymm1*4],xmm2   ;xmm0 = gathered elements
        vmovups xmmword ptr [rcx+16],xmm3   ;save gathered elements

        vzeroupper
        ret
Avx2Gather8xF32_I64_ endp

; extern "C" void Avx2Gather8xF64_I32_(double* y, const double* x, const int32_t* indices, const
int64_t* merge)
Avx2Gather8xF64_I32_ proc
        vmovupd ymm0,ymmword ptr [rcx]  ;ymm0 = y[3]:y[0]
        vmovdqu xmm1,xmmword ptr [r8]   ;xmm1 = indices[3]:indices[0]
        vmovdqu ymm2,ymmword ptr [r9]   ;ymm2 = merge[3]:merge[0]
        vpsllq ymm2,ymm2,63             ;shift merge vals to high-order bits
        vgatherdpd ymm0,[rdx+xmm1*8],ymm2   ;ymm0 = gathered elements
        vmovupd ymmword ptr [rcx],ymm0  ;save gathered elements

        vmovupd ymm0,ymmword ptr [rcx+32]   ;ymm0 = y[7]:y[4]
        vmovdqu xmm1,xmmword ptr [r8+16]    ;xmm1 = indices[7]:indices[4]
        vmovdqu ymm2,ymmword ptr [r9+32]    ;ymm2 = merge[7]:merge[4]
        vpsllq ymm2,ymm2,63             ;shift merge vals to high-order bits
        vgatherdpd ymm0,[rdx+xmm1*8],ymm2   ;ymm0 = gathered elements
        vmovupd ymmword ptr [rcx+32],ymm0   ;save gathered elements

        vzeroupper
        ret
Avx2Gather8xF64_I32_ endp

; extern "C" void Avx2Gather8xF64_I64_(double* y, const double* x, const int64_t* indices, const int64_t* merge)
Avx2Gather8xF64_I64_ proc
        vmovupd ymm0,ymmword ptr [rcx]  ;ymm0 = y[3]:y[0]
        vmovdqu ymm1,ymmword ptr [r8]   ;ymm1 = indices[3]:indices[0]
        vmovdqu ymm2,ymmword ptr [r9]   ;ymm2 = merge[3]:merge[0]
        vpsllq ymm2,ymm2,63             ;shift merge vals to high-order bits
        vgatherqpd ymm0,[rdx+ymm1*8],ymm2   ;ymm0 = gathered elements
        vmovupd ymmword ptr [rcx],ymm0  ;save gathered elements
```

```
        vmovupd ymm0,ymmword ptr [rcx+32]   ;ymm0 = y[7]:y[4]
        vmovdqu ymm1,ymmword ptr [r8+32]    ;ymm1 = indices[7]:indices[4]
        vmovdqu ymm2,ymmword ptr [r9+32]    ;ymm2 = merge[7]:merge[4]
        vpsllq ymm2,ymm2,63             ;shift merge vals to high-order bits
        vgatherqpd ymm0,[rdx+ymm1*8],ymm2   ;ymm0 = gathered elements
        vmovupd ymmword ptr [rcx+32],ymm0   ;save gathered elements

        vzeroupper
        ret
Avx2Gather8xF64_I64_ endp
        end
;-------------------------------------------------
;####### Ch10_01.asm
;-------------------------------------------------
; extern "C" void Avx2PackedMathI16_(const YmmVal& a, const YmmVal& b, YmmVal c[6])
        .code
Avx2PackedMathI16_ proc
; Load values a and b, which must be properly aligned
        vmovdqa ymm0,ymmword ptr [rcx]  ;ymm0 = a
        vmovdqa ymm1,ymmword ptr [rdx]  ;ymm1 = b

; Perform packed arithmetic operations
        vpaddw ymm2,ymm0,ymm1           ;add
        vmovdqa ymmword ptr [r8],ymm2   ;save vpaddw result

        vpaddsw ymm2,ymm0,ymm1          ;add with signed saturation
        vmovdqa ymmword ptr [r8+32],ymm2    ;save vpaddsw result

        vpsubw ymm2,ymm0,ymm1           ;sub
        vmovdqa ymmword ptr [r8+64],ymm2    ;save vpsubw result

        vpsubsw ymm2,ymm0,ymm1          ;sub with signed saturation
        vmovdqa ymmword ptr [r8+96],ymm2    ;save vpsubsw result

        vpminsw ymm2,ymm0,ymm1          ;signed minimums
        vmovdqa ymmword ptr [r8+128],ymm2   ;save vpminsw result

        vpmaxsw ymm2,ymm0,ymm1          ;signed maximums
        vmovdqa ymmword ptr [r8+160],ymm2   ;save vpmaxsw result

        vzeroupper
        ret
Avx2PackedMathI16_ endp

; extern "C" void Avx2PackedMathI32_(const YmmVal& a, const YmmVal& b, YmmVal c[6])
Avx2PackedMathI32_ proc
; Load values a and b, which must be properly aligned
        vmovdqa ymm0,ymmword ptr [rcx]  ;ymm0 = a
        vmovdqa ymm1,ymmword ptr [rdx]  ;ymm1 = b

; Perform packed arithmetic operations
        vpaddd ymm2,ymm0,ymm1           ;add
        vmovdqa ymmword ptr [r8],ymm2   ;save vpaddd result

        vpsubd ymm2,ymm0,ymm1           ;sub
        vmovdqa ymmword ptr [r8+32],ymm2    ;save vpsubd result

        vpmulld ymm2,ymm0,ymm1          ;signed mul (low 32 bits)
        vmovdqa ymmword ptr [r8+64],ymm2    ;save vpmulld result

        vpsllvd ymm2,ymm0,ymm1          ;shift left logical
        vmovdqa ymmword ptr [r8+96],ymm2    ;save vpsllvd result

        vpsravd ymm2,ymm0,ymm1          ;shift right arithmetic
        vmovdqa ymmword ptr [r8+128],ymm2   ;save vpsravd result

        vpabsd ymm2,ymm0                ;absolute value
        vmovdqa ymmword ptr [r8+160],ymm2   ;save vpabsd result

        vzeroupper
        ret
Avx2PackedMathI32_ endp
        end;
;-------------------------------------------------
;####### Ch10_02.asm
;-------------------------------------------------
; extern "C" YmmVal2 Avx2UnpackU32_U64_(const YmmVal& a, const YmmVal& b);

        .code
Avx2UnpackU32_U64_ proc

; Load argument values
        vmovdqa ymm0,ymmword ptr [rdx]  ;ymm0 = a
        vmovdqa ymm1,ymmword ptr [r8]   ;ymm1 = b

; Perform dword to qword unpacks
        vpunpckldq ymm2,ymm0,ymm1       ;unpack low doublewords
        vpunpckhdq ymm3,ymm0,ymm1       ;unpack high doublewords

; Save result to YmmVal2 buffer
        vmovdqa ymmword ptr [rcx],ymm2  ;save low result
        vmovdqa ymmword ptr [rcx+32],ymm3   ;save high result

        mov rax,rcx                     ;rax = ptr to YmmVal2

        vzeroupper
        ret
Avx2UnpackU32_U64_ endp

; extern "C" void Avx2PackI32_I16_(const YmmVal& a, const YmmVal& b, YmmVal* c);

Avx2PackI32_I16_ proc

; Load argument values
        vmovdqa ymm0,ymmword ptr [rcx]  ;ymm0 = a
        vmovdqa ymm1,ymmword ptr [rdx]  ;ymm1 = b

; Perform pack dword to word with signed saturation
        vpackssdw ymm2,ymm0,ymm1        ;ymm2 = packed words
        vmovdqa ymmword ptr [r8],ymm2   ;save result

        vzeroupper
        ret
Avx2PackI32_I16_ endp

Foo1_ proc
        ret
Foo1_ endp
        end
;-------------------------------------------------
;####### Ch10_03.asm
;-------------------------------------------------
; extern "C" void Avx2ZeroExtU8_U16_(YmmVal*a, YmmVal b[2]);

        .code
Avx2ZeroExtU8_U16_ proc
        vpmovzxbw ymm0,xmmword ptr [rcx]    ;zero extend a[0] - a[15]
        vpmovzxbw ymm1,xmmword ptr [rcx+16] ;zero extend a[16] - a[31]

        vmovdqa ymmword ptr [rdx],ymm0      ;save results
        vmovdqa ymmword ptr [rdx+32],ymm1

        vzeroupper
        ret
Avx2ZeroExtU8_U16_ endp

; extern "C" void Avx2ZeroExtU8_U32_(YmmVal*a, YmmVal b[4]);

Avx2ZeroExtU8_U32_ proc
        vpmovzxbd ymm0,qword ptr [rcx]      ;zero extend a[0] - a[7]
        vpmovzxbd ymm1,qword ptr [rcx+8]    ;zero extend a[8] - a[15]
        vpmovzxbd ymm2,qword ptr [rcx+16]   ;zero extend a[16] - a[23]
        vpmovzxbd ymm3,qword ptr [rcx+24]   ;zero extend a[24] - a[31]

        vmovdqa ymmword ptr [rdx],ymm0      ;save results
        vmovdqa ymmword ptr [rdx+32],ymm1
        vmovdqa ymmword ptr [rdx+64],ymm2
        vmovdqa ymmword ptr [rdx+96],ymm3

        vzeroupper
        ret
Avx2ZeroExtU8_U32_ endp

; extern "C" void Avx2SignExtI16_I32_(YmmVal*a, YmmVal b[2])
Avx2SignExtI16_I32_ proc
        vpmovsxwd ymm0,xmmword ptr [rcx]    ;sign extend a[0] - a[7]
        vpmovsxwd ymm1,xmmword ptr [rcx+16] ;sign extend a[8] - a[15]

        vmovdqa ymmword ptr [rdx],ymm0      ;save results
        vmovdqa ymmword ptr [rdx+32],ymm1

        vzeroupper
        ret
Avx2SignExtI16_I32_ endp

; extern "C" void Avx2SignExtI16_I64_(YmmVal*a, YmmVal b[4])
Avx2SignExtI16_I64_ proc
        vpmovsxwq ymm0,qword ptr [rcx]      ;sign extend a[0] - a[3]
        vpmovsxwq ymm1,qword ptr [rcx+8]    ;sign extend a[4] - a[7]
        vpmovsxwq ymm2,qword ptr [rcx+16]   ;sign extend a[8] - a[11]
        vpmovsxwq ymm3,qword ptr [rcx+24]   ;sign extend a[12] - a[15]

        vmovdqa ymmword ptr [rdx],ymm0      ;save results
        vmovdqa ymmword ptr [rdx+32],ymm1
        vmovdqa ymmword ptr [rdx+64],ymm2
        vmovdqa ymmword ptr [rdx+96],ymm3

        vzeroupper
        ret
Avx2SignExtI16_I64_ endp
        end
;-------------------------------------------------
;####### Ch10_04.asm
;-------------------------------------------------
; The following structure must match the structure that's declared in the file .h file
ClipData            struct
Src                 qword ?         ;source buffer pointer
Des                 qword ?         ;destination buffer pointer
NumPixels           qword ?         ;number of pixels
NumClippedPixels    qword ?         ;number of clipped pixels
ThreshLo            byte ?          ;low threshold
ThreshHi            byte ?          ;high threshold
ClipData            ends

; extern "C" bool Avx2ClipPixels_(ClipData* cd)
            .code
Avx2ClipPixels_ proc

; Load and validate arguments
        xor eax,eax                     ;set error return code
        xor r8d,r8d                     ;r8 = number of clipped pixels
        mov rdx,[rcx+ClipData.NumPixels]    ;rdx = num_pixels
        or rdx,rdx
        jz Done                             ;jump of num_pixels is zero
        test rdx,1fh
        jnz Done                            ;jump if num_pixels % 32 != 0

        mov r10,[rcx+ClipData.Src]          ;r10 = Src
        test r10,1fh
        jnz Done                            ;jump if Src is misaligned

        mov r11,[rcx+ClipData.Des]          ;r11 = Des
        test r11,1fh
        jnz Done                            ;jump if Des is misaligned

; Create packed thresh_lo and thresh_hi data values
        vpbroadcastb ymm4,[rcx+ClipData.ThreshLo]   ;ymm4 = packed thresh_lo
        vpbroadcastb ymm5,[rcx+ClipData.ThreshHi]   ;ymm5 = packed thresh_hi

; Clip pixels to threshold values
@@:     vmovdqa ymm0,ymmword ptr [r10]  ;ymm0 = 32 pixels
        vpmaxub ymm1,ymm0,ymm4          ;clip to thresh_lo
        vpminub ymm2,ymm1,ymm5          ;clip to thresh_hi
        vmovdqa ymmword ptr [r11],ymm2  ;save clipped pixels

; Count number of clipped pixels
        vpcmpeqb ymm3,ymm2,ymm0         ;compare clipped pixels to original
        vpmovmskb eax,ymm3              ;eax = mask of non-clipped pixels
        not eax                         ;eax = mask of clipped pixels
        popcnt eax,eax                  ;eax = number of clipped pixels
        add r8,rax                      ;update clipped pixel count

; Update pointers and loop counter
        add r10,32                      ;update Src ptr
        add r11,32                      ;update Des ptr
        sub rdx,32                      ;update loop counter
        jnz @B                          ;repeat if not done

        mov eax,1                       ;set success return code

; Save num_clipped_pixels

Done:   mov [rcx+ClipData.NumClippedPixels],r8  ;save num_clipped_pixels
        vzeroupper
        ret

Avx2ClipPixels_ endp
        end
;-------------------------------------------------
;####### Ch10_05.asm
        include <MacrosX86-64-AVX.asmh>

; 256-bit wide constants
ConstVals       segment readonly align(32) 'const'
InitialPminVal  db 32 dup(0ffh)
InitialPmaxVal  db 32 dup(00h)
ConstVals       ends

; Macro _YmmVpextrMinub
; This macro generates code that extracts the smallest unsigned byte from register YmmSrc.

_YmmVpextrMinub macro GprDes,YmmSrc,YmmTmp

; Make sure YmmSrc and YmmTmp are different
.erridni <YmmSrc>, <YmmTmp>, <Invalid registers>

; Construct text strings for the corresponding XMM registers
        YmmSrcSuffix SUBSTR <YmmSrc>,2
        XmmSrc CATSTR <X>,YmmSrcSuffix

        YmmTmpSuffix SUBSTR <YmmTmp>,2
        XmmTmp CATSTR <X>,YmmTmpSuffix

; Reduce the 32 byte values in YmmSrc to the smallest value
        vextracti128 XmmTmp,YmmSrc,1
        vpminub XmmSrc,XmmSrc,XmmTmp    ;XmmSrc = final 16 min values

        vpsrldq XmmTmp,XmmSrc,8
        vpminub XmmSrc,XmmSrc,XmmTmp    ;XmmSrc = final 8 min values

        vpsrldq XmmTmp,XmmSrc,4
        vpminub XmmSrc,XmmSrc,XmmTmp    ;XmmSrc = final 4 min values

        vpsrldq XmmTmp,XmmSrc,2
        vpminub XmmSrc,XmmSrc,XmmTmp    ;XmmSrc = final 2 min values

        vpsrldq XmmTmp,XmmSrc,1
        vpminub XmmSrc,XmmSrc,XmmTmp    ;XmmSrc = final 1 min value

        vpextrb GprDes,XmmSrc,0         ;mov final min value to Gpr
        endm

; Macro _YmmVpextrMaxub
; This macro generates code that extracts the largest unsigned byte from register YmmSrc.

_YmmVpextrMaxub macro GprDes,YmmSrc,YmmTmp

; Make sure YmmSrc and YmmTmp are different
.erridni <YmmSrc>, <YmmTmp>, <Invalid registers>

; Construct text strings for the corresponding XMM registers
        YmmSrcSuffix SUBSTR <YmmSrc>,2
        XmmSrc CATSTR <X>,YmmSrcSuffix

        YmmTmpSuffix SUBSTR <YmmTmp>,2
        XmmTmp CATSTR <X>,YmmTmpSuffix

; Reduce the 32 byte values in YmmSrc to the largest value
        vextracti128 XmmTmp,YmmSrc,1
        vpmaxub XmmSrc,XmmSrc,XmmTmp    ;XmmSrc = final 16 max values

        vpsrldq XmmTmp,XmmSrc,8
        vpmaxub XmmSrc,XmmSrc,XmmTmp    ;XmmSrc = final 8 max values

        vpsrldq XmmTmp,XmmSrc,4
        vpmaxub XmmSrc,XmmSrc,XmmTmp    ;XmmSrc = final 4 max values

        vpsrldq XmmTmp,XmmSrc,2
        vpmaxub XmmSrc,XmmSrc,XmmTmp    ;XmmSrc = final 2 max values

        vpsrldq XmmTmp,XmmSrc,1
        vpmaxub XmmSrc,XmmSrc,XmmTmp    ;XmmSrc = final 1 max value

        vpextrb GprDes,XmmSrc,0         ;mov final max value to Gpr
        endm

; extern "C" bool Avx2CalcRgbMinMax_(uint8_t* rgb[3], size_t num_pixels, uint8_t min_vals[3],
uint8_t max_vals[3])
        .code
Avx2CalcRgbMinMax_ proc frame
        _CreateFrame CalcMinMax_,0,48,r12
        _SaveXmmRegs xmm6,xmm7,xmm8
        _EndProlog
```

```
; Make sure num_pixels and the color plane arrays are valid
        xor eax,eax                     ;set error code

        test rdx,rdx
        jz Done                         ;jump if num_pixels == 0
        test rdx,01fh
        jnz Done                        ;jump if num_pixels % 32 != 0

        mov r10,[rcx]                   ;r10 = color plane R
        test r10,1fh
        jnz Done                        ;jump if color plane R is not aligned

        mov r11,[rcx+8]                 ;r11 = color plane G
        test r11,1fh
        jnz Done                        ;jump if color plane G is not aligned

        mov r12,[rcx+16]                ;r12 = color plane B
        test r12,1fh
        jnz Done                        ;jump if color plane B is not aligned

; Initialize the processing loop registers
        vmovdqa ymm3,ymmword ptr [InitialPminVal]   ;ymm3 = R minimums
        vmovdqa ymm4,ymm3                           ;ymm4 = G minimums
        vmovdqa ymm5,ymm3                           ;ymm5 = B minimums

        vmovdqa ymm6,ymmword ptr [InitialPmaxVal]   ;ymm6 = R maximums
        vmovdqa ymm7,ymm6                           ;ymm7 = G maximums
        vmovdqa ymm8,ymm6                           ;ymm8 = B maximums

        xor rcx,rcx                     ;rcx = common array offset

; Scan RGB color plane arrays for packed minimums and maximums
        align 16
@@:     vmovdqa ymm0,ymmword ptr [r10+rcx]  ;ymm0 = R pixels
        vmovdqa ymm1,ymmword ptr [r11+rcx]  ;ymm1 = G pixels
        vmovdqa ymm2,ymmword ptr [r12+rcx]  ;ymm2 = B pixels

        vpminub ymm3,ymm3,ymm0          ;update R minimums
        vpminub ymm4,ymm4,ymm1          ;update G minimums
        vpminub ymm5,ymm5,ymm2          ;update B minimums

        vpmaxub ymm6,ymm6,ymm0          ;update R maximums
        vpmaxub ymm7,ymm7,ymm1          ;update G maximums
        vpmaxub ymm8,ymm8,ymm2          ;update B maximums

        add rcx,32
        sub rdx,32
        jnz @B

; Calculate the final RGB minimum values
        _YmmVpextrMinub rax,ymm3,ymm0
        mov byte ptr [r8],al            ;save min R
        _YmmVpextrMinub rax,ymm4,ymm0
        mov byte ptr [r8+1],al          ;save min G
        _YmmVpextrMinub rax,ymm5,ymm0
        mov byte ptr [r8+2],al          ;save min B

; Calculate the final RGB maximum values
        _YmmVpextrMaxub rax,ymm6,ymm1
        mov byte ptr [r9],al            ;save max R
        _YmmVpextrMaxub rax,ymm7,ymm1
        mov byte ptr [r9+1],al          ;save max G
        _YmmVpextrMaxub rax,ymm8,ymm1
        mov byte ptr [r9+2],al          ;save max B

        mov eax,1                       ;set success return code
```

```
Done:   vzeroupper
        _RestoreXmmRegs xmm6,xmm7,xmm8
        _DeleteFrame r12
        ret
Avx2CalcRgbMinMax_ endp
        end
;-------------------------------------------------
;####### Ch10_06.asm
        include <MacrosX86-64-AVX.asmh>

                .const
GsMask          dword 0ffffffffh, 0, 0, 0, 0ffffffffh, 0, 0, 0
r4_0p5          real4 0.5
r4_255p0        real4 255.0

                extern c_NumPixelsMin:dword
                extern c_NumPixelsMax:dword

;extern "C" bool Avx2ConvertRgbToGs_(uint8_t* pb_gs, const RGB32* pb_rgb, int num_pixels, const float coef[4])
; Note: Memory pointed to by pb_rgb is ordered as follows:
;       R(0,0), G(0,0), B(0,0), A(0,0), R(0,1), G(0,1), B(0,1), A(0,1), ...

        .code
Avx2ConvertRgbToGs_ proc frame
        _CreateFrame RGBGS_,0,112
        _SaveXmmRegs xmm6,xmm7,xmm11,xmm12,xmm13,xmm14,xmm15
        _EndProlog

; Validate argument values
        xor eax,eax                     ;set error return code
        cmp r8d,[c_NumPixelsMin]
        jl Done                         ;jump if num_pixels < min value
        cmp r8d,[c_NumPixelsMax]
        jg Done                         ;jump if num_pixels > max value
        test r8d,7
        jnz Done                        ;jump if (num_pixels % 8) != 0

        test rcx,1fh
        jnz Done                        ;jump if pb_gs is not aligned
        test rdx,1fh
        jnz Done                        ;jump if pb_rgb is not aligned

; Perform required initializations
        vbroadcastss ymm11,real4 ptr [r4_255p0] ;ymm11 = packed 255.0
        vbroadcastss ymm12,real4 ptr [r4_0p5]   ;ymm12 = packed 0.5
        vpxor ymm13,ymm13,ymm13                 ;ymm13 = packed zero

        vmovups xmm0,xmmword ptr [r9]
        vperm2f128 ymm14,ymm0,ymm0,00000000b    ;ymm14 = packed coef

        vmovups ymm15,ymmword ptr [GsMask]      ;ymm15 = GsMask (SPFP)
; Load next 8 RGB32 pixel values (P0 - P7)
        align 16
@@:     vmovdqa ymm0,ymmword ptr [rdx]  ;ymm0 = 8 rgb32 pixels (P7 - P0)
; Size-promote RGB32 color components from bytes to dwords
        vpunpcklbw ymm1,ymm0,ymm13
        vpunpckhbw ymm2,ymm0,ymm13
        vpunpcklwd ymm3,ymm1,ymm13      ;ymm3 = P1, P0 (dword)
        vpunpckhwd ymm4,ymm1,ymm13      ;ymm4 = P3, P2 (dword)
        vpunpcklwd ymm5,ymm2,ymm13      ;ymm5 = P5, P4 (dword)
        vpunpckhwd ymm6,ymm2,ymm13      ;ymm6 = P7, P6 (dword)
; Convert color component values to single-precision floating-point
        vcvtdq2ps ymm0,ymm3             ;ymm0 = P1, P0 (SPFP)
        vcvtdq2ps ymm1,ymm4             ;ymm1 = P3, P2 (SPFP)
        vcvtdq2ps ymm2,ymm5             ;ymm2 = P5, P4 (SPFP)
        vcvtdq2ps ymm3,ymm6             ;ymm3 = P7, P6 (SPFP)
; Multiply color component values by color conversion coefficients
```

```
        vmulps ymm0,ymm0,ymm14
        vmulps ymm1,ymm1,ymm14
        vmulps ymm2,ymm2,ymm14
        vmulps ymm3,ymm3,ymm14

; Sum weighted color components for final grayscale values
        vhaddps ymm4,ymm0,ymm0
        vhaddps ymm4,ymm4,ymm4          ;ymm4[159:128] = P1, ymm4[31:0] = P0
        vhaddps ymm5,ymm1,ymm1
        vhaddps ymm5,ymm5,ymm5          ;ymm5[159:128] = P3, ymm4[31:0] = P2
        vhaddps ymm6,ymm2,ymm2
        vhaddps ymm6,ymm6,ymm6          ;ymm6[159:128] = P5, ymm4[31:0] = P4
        vhaddps ymm7,ymm3,ymm3
        vhaddps ymm7,ymm7,ymm7          ;ymm7[159:128] = P7, ymm4[31:0] = P6

; Merge SPFP grayscale values into a single YMM register
        vandps ymm4,ymm4,ymm15          ;mask out unneeded SPFP values
        vandps ymm5,ymm5,ymm15
        vandps ymm6,ymm6,ymm15
        vandps ymm7,ymm7,ymm15
        vpslldq ymm5,ymm5,4
        vpslldq ymm6,ymm6,8
        vpslldq ymm7,ymm7,12
        vorps ymm0,ymm4,ymm5            ;merge values
        vorps ymm1,ymm6,ymm7
        vorps ymm2,ymm0,ymm1            ;ymm2 = 8 GS pixel values (SPFP)
; Add 0.5 rounding factor and clip to 0.0 - 255.0
        vaddps ymm2,ymm2,ymm12          ;add 0.5f rounding factor
        vminps ymm3,ymm2,ymm11          ;clip pixels above 255.0
        vmaxps ymm4,ymm3,ymm13          ;clip pixels below 0.0

; Convert SPFP values to bytes and save
        vcvtps2dq ymm3,ymm2             ;convert GS SPFP to dwords
        vpackusdw ymm4,ymm3,ymm13       ;convert GS dwords to words
        vpackuswb ymm5,ymm4,ymm13       ;convert GS words to bytes

        vperm2i128 ymm6,ymm13,ymm5,3    ;xmm5 = GS P3:P0, xmm6 = GS P7:P4

        vmovd dword ptr [rcx],xmm5      ;save P3 - P0
        vmovd dword ptr [rcx+4],xmm6    ;save P7 - P4

        add rdx,32                      ;update pb_rgb to next block
        add rcx,8                       ;update pb_gs to next block
        sub r8d,8                       ;num_pixels -= 8
        jnz @B                          ;repeat until done

        mov eax,1                       ;set success return code

Done:  vzeroupper
        _RestoreXmmRegs xmm6,xmm7,xmm11,xmm12,xmm13,xmm14,xmm15
        _DeleteFrame
        ret
Avx2ConvertRgbToGs_ endp
        end
;-----------------------------------------------
;####### Ch11_01_.asm
        include <MacrosX86-64-AVX.asmh>
        extern c_NumPtsMin:dword
        extern c_NumPtsMax:dword
        extern c_KernelSizeMin:dword
        extern c_KernelSizeMax:dword

; extern ″C″ bool Convolve1_(float* y, const float* x, int num_pts, const float* kernel, int kernel_size)
         .code
Convolve1_ proc frame
        _CreateFrame CV_,0,0,rbx,rsi
        _EndProlog
```

```
; Verify argument values
        xor eax,eax                     ;set error code (rax is also loop index var)
        mov r10d,dword ptr [rbp+CV_OffsetStackArgs]
        test r10d,1
        jz Done                         ;jump if kernel_size is even
        cmp r10d,[c_KernelSizeMin]
        jl Done                         ;jump if kernel_size too small
        cmp r10d,[c_KernelSizeMax]
        jg Done                         ;jump if kernel_size too big

        cmp r8d,[c_NumPtsMin]
        jl Done                         ;jump if num_pts too small
        cmp r8d,[c_NumPtsMax]
        jg Done                         ;jump if num_pts too big

; Perform required initializations
        mov r8d,r8d                     ;r8 = num_pts
        shr r10d,1                      ;ks2 = ks / 2
        lea rdx,[rdx+r10*4]             ;rdx = x + ks2 (first data point)
; Perform convolution
LP1:    vxorps xmm5,xmm5,xmm5           ;sum = 0.0;
        mov r11,r10
        neg r11                         ;k = -ks2

LP2:    mov rbx,rax
        sub rbx,r11                     ;rbx = i - k
        vmovss xmm0,real4 ptr [rdx+rbx*4]   ;xmm0 = x[i - k]
        mov rsi,r11
        add rsi,r10                     ;rsi = k + ks2
        vfmadd231ss xmm5,xmm0,[r9+rsi*4]    ;sum += x[i - k] * kernel[k + ks2]

        add r11,1                       ;k++
        cmp r11,r10
        jle LP2                         ;jump if k <= ks2

        vmovss real4 ptr [rcx+rax*4],xmm5   ;y[i] = sum

        add rax,1                       ;i += 1
        cmp rax,r8
        jl LP1                          ;jump if i < num_pts

        mov eax,1                       ;set success return code

Done:   vzeroupper
        _DeleteFrame rbx,rsi
        ret
Convolve1_ endp

; extern ″C″ bool Convolve1Ks5_(float* y, const float* x, int num_pts, const float* kernel, int kernel_size)
Convolve1Ks5_ proc
; Verify argument values
        xor eax,eax                     ;set error code (rax is also loop index var)
        cmp dword ptr [rsp+40],5
        jne Done                        ;jump if kernel_size is not 5

        cmp r8d,[c_NumPtsMin]
        jl Done                         ;jump if num_pts too small
        cmp r8d,[c_NumPtsMax]
        jg Done                         ;jump if num_pts too big

; Perform required initializations
        mov r8d,r8d                     ;r8 = num_pts
        add rdx,8                       ;x += 2

; Perform convolution
@@:     vxorps xmm4,xmm4,xmm4               ;initialize sum vars
```

```
        vxorps xmm5,xmm5,xmm5
        mov r11,rax
        add r11,2                           ;j = i + ks2

        vmovss xmm0,real4 ptr [rdx+r11*4]   ;xmm0 = x[j]
        vfmadd231ss xmm4,xmm0,[r9]          ;xmm4 += x[j] * kernel[0]

        vmovss xmm1,real4 ptr [rdx+r11*4-4] ;xmm1 = x[j - 1]
        vfmadd231ss xmm5,xmm1,[r9+4]        ;xmm5 += x[j - 1] * kernel[1]

        vmovss xmm0,real4 ptr [rdx+r11*4-8] ;xmm0 = x[j - 2]
        vfmadd231ss xmm4,xmm0,[r9+8]        ;xmm4 += x[j - 2] * kernel[2]

        vmovss xmm1,real4 ptr [rdx+r11*4-12]    ;xmm1 = x[j - 3]
        vfmadd231ss xmm5,xmm1,[r9+12]       ;xmm5 += x[j - 3] * kernel[3]

        vmovss xmm0,real4 ptr [rdx+r11*4-16]    ;xmm0 = x[j - 4]
        vfmadd231ss xmm4,xmm0,[r9+16]       ;xmm4 += x[j - 4] * kernel[4]

        vaddps xmm4,xmm4,xmm5
        vmovss real4 ptr [rcx+rax*4],xmm4   ;save y[i]

        inc rax                             ;i += 1
        cmp rax,r8
        jl @B                               ;jump if i < num_pts

        mov eax,1                           ;set success return code

Done:   vzeroupper
        ret
Convolve1Ks5_ endp
        end
;------------------------------------------------
;####### Ch11_02_.asm
        include <MacrosX86-64-AVX.asmh>
        extern c_NumPtsMin:dword
        extern c_NumPtsMax:dword
        extern c_KernelSizeMin:dword
        extern c_KernelSizeMax:dword

; extern bool Convolve2_(float* y, const float* x, int num_pts, const float* kernel, int kernel_size)
        .code
Convolve2_ proc frame
        _CreateFrame CV2_,0,0,rbx
        _EndProlog

; Validate argument values
        xor eax,eax                     ;set error code

        mov r10d,dword ptr [rbp+CV2_OffsetStackArgs]
        test r10d,1
        jz Done                         ;kernel_size is even
        cmp r10d,[c_KernelSizeMin]
        jl Done                         ;kernel_size too small
        cmp r10d,[c_KernelSizeMax]
        jg Done                         ;kernel_size too big

        cmp r8d,[c_NumPtsMin]
        jl Done                         ;num_pts too small
        cmp r8d,[c_NumPtsMax]
        jg Done                         ;num_pts too big
        test r8d,7
        jnz Done                        ;num_pts not even multiple of 8

        test rcx,1fh
        jnz Done                        ;y is not properly aligned
```

```
; Initialize convolution loop variables
        shr r10d,1                      ;r10 = kernel_size / 2 (ks2)
        lea rdx,[rdx+r10*4]             ;rdx = x + ks2 (first data point)
        xor ebx,ebx                     ;i = 0

; Perform convolution
LP1:    vxorps ymm0,ymm0,ymm0           ;packed sum = 0.0;
        mov r11,r10                     ;r11 = ks2
        neg r11                         ;k = -ks2

LP2:    mov rax,rbx                         ;rax = i
        sub rax,r11                         ;rax = i - k
        vmovups ymm1,ymmword ptr [rdx+rax*4]    ;load x[i - k]:x[i - k + 7]

        mov rax,r11
        add rax,r10                         ;rax = k + ks2
        vbroadcastss ymm2,real4 ptr [r9+rax*4]  ;ymm2 = kernel[k + ks2]
        vfmadd231ps ymm0,ymm1,ymm2          ;ymm0 += x[i-k]:x[i-k+7] * kernel[k+ks2]

        add r11,1                           ;k += 1
        cmp r11,r10
        jle LP2                             ;repeat until k > ks2

        vmovaps ymmword ptr [rcx+rbx*4],ymm0    ;save y[i]:y[i + 7]

        add rbx,8                       ;i += 8
        cmp rbx,r8
        jl LP1                          ;repeat until done
        mov eax,1                       ;set success return code

Done:   vzeroupper
        _DeleteFrame rbx
        ret
Convolve2_ endp

; extern bool Convolve2Ks5_(float* y, const float* x, int num_pts, const float* kernel, int kernel_size)
Convolve2Ks5_ proc frame
        _CreateFrame CKS5_,0,48
        _SaveXmmRegs xmm6,xmm7,xmm8
        _EndProlog

; Validate argument values
        xor eax,eax                     ;set error code (rax is also loop index var)
        cmp dword ptr [rbp+CKS5_OffsetStackArgs],5
        jne Done                        ;jump if kernel_size is not 5

        cmp r8d,[c_NumPtsMin]
        jl Done                         ;jump if num_pts too small
        cmp r8d,[c_NumPtsMax]
        jg Done                         ;jump if num_pts too big
        test r8d,7
        jnz Done                        ;num_pts not even multiple of 8

        test rcx,1fh
        jnz Done                        ;y is not properly aligned

; Perform required initializations
        vbroadcastss ymm4,real4 ptr [r9]    ;kernel[0]
        vbroadcastss ymm5,real4 ptr [r9+4]  ;kernel[1]
        vbroadcastss ymm6,real4 ptr [r9+8]  ;kernel[2]
        vbroadcastss ymm7,real4 ptr [r9+12] ;kernel[3]
        vbroadcastss ymm8,real4 ptr [r9+16] ;kernel[4]
        mov r8d,r8d                         ;r8 = num_pts
        add rdx,8                           ;x += 2

; Perform convolution
@@:     vxorps ymm2,ymm2,ymm2               ;initialize sum vars
```

```
        vxorps ymm3,ymm3,ymm3
        mov r11,rax
        add r11,2                           ;j = i + ks2

        vmovups ymm0,ymmword ptr [rdx+r11*4]    ;ymm0 = x[j]:x[j + 7]
        vfmadd231ps ymm2,ymm0,ymm4          ;ymm2 += x[j]:x[j + 7] * kernel[0]

        vmovups ymm1,ymmword ptr [rdx+r11*4-4]  ;ymm1 = x[j - 1]:x[j + 6]
        vfmadd231ps ymm3,ymm1,ymm5          ;ymm3 += x[j - 1]:x[j + 6] * kernel[1]

        vmovups ymm0,ymmword ptr [rdx+r11*4-8]  ;ymm0 = x[j - 2]:x[j + 5]
        vfmadd231ps ymm2,ymm0,ymm6          ;ymm2 += x[j - 2]:x[j + 5] * kernel[2]

        vmovups ymm1,ymmword ptr [rdx+r11*4-12] ;ymm1 = x[j - 3]:x[j + 4]
        vfmadd231ps ymm3,ymm1,ymm7          ;ymm3 += x[j - 3]:x[j + 4] * kernel[3]

        vmovups ymm0,ymmword ptr [rdx+r11*4-16] ;ymm0 = x[j - 4]:x[j + 3]
        vfmadd231ps ymm2,ymm0,ymm8          ;ymm2 += x[j - 4]:x[j + 3] * kernel[4]

        vaddps ymm0,ymm2,ymm3               ;final values
        vmovaps ymmword ptr [rcx+rax*4],ymm0    ;save y[i]:y[i + 7]

        add rax,8                           ;i += 8
        cmp rax,r8
        jl @B                               ;jump if i < num_pts
        mov eax,1                           ;set success return code

Done:   vzeroupper
        _RestoreXmmRegs xmm6,xmm7,xmm8
        _DeleteFrame
        ret
Convolve2Ks5_ endp
        end
;-------------------------------------------------
;####### Ch11_02_Test_.asm
        include <MacrosX86-64-AVX.asmh>
        extern c_NumPtsMin:dword
        extern c_NumPtsMax:dword

; extern bool Convolve2Ks5Test_(float* y, const float* x, int num_pts, const float* kernel, int
kernel_size)
        .code
Convolve2Ks5Test_ proc frame
        _CreateFrame CKS5T_,0,48
        _SaveXmmRegs xmm6,xmm7,xmm8
        _EndProlog

; Validate argument values
        xor eax,eax                     ;set error code (rax is also loop index var)
        cmp dword ptr [rbp+CKS5T_OffsetStackArgs],5
        jne Done                        ;jump if kernel_size is not 5

        cmp r8d,[c_NumPtsMin]
        jl Done                         ;jump if num_pts too small
        cmp r8d,[c_NumPtsMax]
        jg Done                         ;jump if num_pts too big
        test r8d,7
        jnz Done                        ;num_pts not even multiple of 8

        test rcx,1fh
        jnz Done                        ;y is not properly aligned

; Perform required initializations
        vbroadcastss ymm4,real4 ptr [r9]    ;kernel[0]
        vbroadcastss ymm5,real4 ptr [r9+4]  ;kernel[1]
        vbroadcastss ymm6,real4 ptr [r9+8]  ;kernel[2]
        vbroadcastss ymm7,real4 ptr [r9+12] ;kernel[3]
        vbroadcastss ymm8,real4 ptr [r9+16] ;kernel[4]
        mov r8d,r8d                         ;r8 = num_pts
        add rdx,8                           ;x += 2

; Perform convolution
@@:     vxorps ymm2,ymm2,ymm2               ;initialize sum vars
        vxorps ymm3,ymm3,ymm3
        mov r11,rax
        add r11,2                           ;j = i + ks2

        vmovups ymm0,ymmword ptr [rdx+r11*4]    ;ymm0 = x[j]:x[j + 7]
        vmulps ymm0,ymm0,ymm4
        vaddps ymm2,ymm2,ymm0                   ;ymm2 += x[j]:x[j + 7] * kernel[0]

        vmovups ymm1,ymmword ptr [rdx+r11*4-4]  ;ymm1 = x[j - 1]:x[j + 6]
        vmulps ymm1,ymm1,ymm5
        vaddps ymm3,ymm3,ymm1                   ;ymm3 += x[j - 1]:x[j + 6] * kernel[1]

        vmovups ymm0,ymmword ptr [rdx+r11*4-8]  ;ymm0 = x[j - 2]:x[j + 5]
        vmulps ymm0,ymm0,ymm6
        vaddps ymm2,ymm2,ymm0                   ;ymm2 += x[j - 2]:x[j + 5] * kernel[2]

        vmovups ymm1,ymmword ptr [rdx+r11*4-12] ;ymm1 = x[j - 3]:x[j + 4]
        vmulps ymm1,ymm1,ymm7
        vaddps ymm3,ymm3,ymm1                   ;ymm3 += x[j - 3]:x[j + 4] * kernel[3]

        vmovups ymm0,ymmword ptr [rdx+r11*4-16] ;ymm0 = x[j - 4]:x[j + 3]
        vmulps ymm0,ymm0,ymm8
        vaddps ymm2,ymm2,ymm0               ;ymm2 += x[j - 4]:x[j + 3] * kernel[4]

        vaddps ymm0,ymm2,ymm3               ;final values
        vmovaps ymmword ptr [rcx+rax*4],ymm0    ;save y[i]:y[i + 7]

        add rax,8                           ;i += 8
        cmp rax,r8
        jl @B                               ;jump if i < num_pts
        mov eax,1                           ;set success return code

Done:   vzeroupper
        _RestoreXmmRegs xmm6,xmm7,xmm8
        _DeleteFrame
        ret
Convolve2Ks5Test_ endp
        end
;-------------------------------------------------
;####### Ch11_03_.asm
;-------------------------------------------------
; extern "C" uint64_t GprMulx_(uint32_t a, uint32_t b, uint64_t flags[2]);
; Requires      BMI2

        .code
GprMulx_ proc

; Save copy of status flags before mulx
        pushfq
        pop rax
        mov qword ptr [r8],rax          ;save original status flags

; Perform flagless multiplication. The mulx instruction below computes
; the product of explicit source operand ecx (a) and implicit source
; operand edx (b). The 64-bit result is saved to the register pair r11d:r10d.
        mulx r11d,r10d,ecx              ;r11d:r10d = a * b

; Save copy of status flags after mulx
        pushfq
        pop rax
        mov qword ptr [r8+8],rax        ;save post mulx status flags
```

```
; Move 64-bit result to rax
        mov eax,r10d
        shl r11,32
        or rax,r11
        ret
GprMulx_ endp

; extern "C" void GprShiftx_(uint32_t x, uint32_t count, uint32_t results[3], uint64_t flags[4])
; Requires      BMI2

GprShiftx_ proc

; Save copy of status flags before shifts
        pushfq
        pop rax
        mov qword ptr [r9],rax          ;save original status flags

; Load argument values and perform shifts.  Note that each shift
; instruction requires three operands: DesOp, SrcOp, and CountOp.

        sarx eax,ecx,edx                ;shift arithmetic right
        mov dword ptr [r8],eax
        pushfq
        pop rax
        mov qword ptr [r9+8],rax

        shlx eax,ecx,edx                ;shift logical left
        mov dword ptr [r8+4],eax
        pushfq
        pop rax
        mov qword ptr [r9+16],rax

        shrx eax,ecx,edx                ;shift logical right
        mov dword ptr [r8+8],eax
        pushfq
        pop rax
        mov qword ptr [r9+24],rax

        ret
GprShiftx_ endp
        end
;-------------------------------------------------
;####### Ch11_04_.asm
;-------------------------------------------------
; extern "C" void GprCountZeroBits_(uint32_t x, uint32_t* lzcnt, uint32_t* tzcnt);
; Requires:     BMI1, LZCNT

        .code
GprCountZeroBits_ proc
        lzcnt eax,ecx                   ;count leading zeros
        mov dword ptr [rdx],eax         ;save result

        tzcnt eax,ecx                   ;count trailing zeros
        mov dword ptr [r8],eax          ;save result
        ret
GprCountZeroBits_ endp

; extern "C" uint32_t GprBextr_(uint32_t x, uint8_t start, uint8_t length);
; Requires:     BMI1

GprBextr_ proc
        mov al,r8b
        mov ah,al                       ;ah = length
        mov al,dl                       ;al = start
        bextr eax,ecx,eax               ;eax = extracted bit field (from x)
        ret
GprBextr_ endp

; extern "C" uint32_t GprAndNot_(uint32_t x, uint32_t y);
; Requires:     BMI1

GprAndNot_ proc
        andn eax,ecx,edx                ;eax = ~x & y
        ret
GprAndNot_ endp
        end
;-------------------------------------------------
;####### Ch11_05_.asm
;-------------------------------------------------
; extern "C" void SingleToHalfPrecision_(uint16_t x_hp[8], float x_sp[8], int rc);

        .code
SingleToHalfPrecision_ proc

; Convert packed single-precision to packed half-precision
        vmovups ymm0,ymmword ptr [rdx]      ;ymm0 = 8 SPFP values

        cmp r8d,0
        jne @F
        vcvtps2ph xmm1,ymm0,0               ;round to nearest
        jmp SaveResult

@@:     cmp r8d,1
        jne @F
        vcvtps2ph xmm1,ymm0,1               ;round down
        jmp SaveResult

@@:     cmp r8d,2
        jne @F
        vcvtps2ph xmm1,ymm0,2               ;round up
        jmp SaveResult

@@:     cmp r8d,3
        jne @F
        vcvtps2ph xmm1,ymm0,3               ;truncate
        jmp SaveResult

@@:     vcvtps2ph xmm1,ymm0,4               ;use MXCSR.RC

SaveResult:
        vmovdqu xmmword ptr [rcx],xmm1      ;save 8 HPFP values
        vzeroupper
        ret

SingleToHalfPrecision_ endp

; extern "C" void HalfToSinglePrecision_(float x_sp[8], uint16_t x_hp[8]);

HalfToSinglePrecision_ proc

; Convert packed half-precision to packed single-precision
        vcvtph2ps ymm0,xmmword ptr [rdx]
        vmovups ymmword ptr [rcx],ymm0      ;save 8 SPFP values

        vzeroupper
        ret

HalfToSinglePrecision_ endp
        end
;-------------------------------------------------
;####### Ch13_01.asm
;-------------------------------------------------
            include <cmpequ.asmh>
            .const
r8_three    real8 3.0
r8_four     real8 4.0

            extern g_PI:real8

; extern "C" bool Avx512CalcSphereAreaVol_(double* sa, double* v, double r, double error_val);
; Returns:  false = invalid radius, true = valid radius

        .code
Avx512CalcSphereAreaVol_ proc

; Test radius for value >= 0.0
        vmovsd xmm0,xmm0,xmm2           ;xmm0 = radius
        vxorpd xmm5,xmm5,xmm5           ;xmm5 = 0.0
        vmovsd xmm16,xmm16,xmm3         ;xmm16 = error_val
        vcmpsd k1,xmm0,xmm5,CMP_GE      ;k1[0] = 1 if radius >= 0.0

; Calculate surface area and volume using mask from compare
        vmulsd xmm1{k1},xmm0,xmm0       ;xmm1 = r * r
        vmulsd xmm2{k1},xmm1,[r8_four]  ;xmm2 = 4 * r * r
        vmulsd xmm3{k1},xmm2,[g_PI]     ;xmm3 = 4 * PI * r * r (sa)
        vmulsd xmm4{k1},xmm3,xmm0       ;xmm4 = 4 * PI * r * r * r
        vdivsd xmm5{k1},xmm4,[r8_three] ;xmm5 = 4 * PI * r * r * r / 3 (vol)
; Set surface area and volume to error_val if radius < 0.0 is true
        knotw k2,k1                     ;k2[0] = 1 if radius < 0.0
        vmovsd xmm3{k2},xmm3,xmm16      ;xmm3 = error_val if radius < 0.0
        vmovsd xmm5{k2},xmm5,xmm16      ;xmm5 = error_val if radius < 0.0

; Save results
        vmovsd real8 ptr [rcx],xmm3     ;save surface area
        vmovsd real8 ptr [rdx],xmm5     ;save volume

        kmovw eax,k1                    ;eax = return code
        ret
Avx512CalcSphereAreaVol_ endp
        end
;-------------------------------------------------
;####### Ch13_02.asm
        include <cmpequ.asmh>

; extern "C" bool Avx512CalcValues_(double* c, const double* a, const double* b, size_t n);

        .code
Avx512CalcValues_ proc

; Validate n and initialize array index i
        xor eax,eax                     ;set error return code (also i = 0)
        test r9,r9                      ;is n == 0?
        jz Done                         ;jump if n is zero

        vxorpd xmm5,xmm5,xmm5           ;xmm5 = 0.0

; Load next a[i] and b[i], calculate val
@@:     vmovsd xmm0,real8 ptr [rdx+rax*8]   ;xmm0 = a[i];
        vmovsd xmm1,real8 ptr [r8+rax*8]    ;xmm1 = b[i];
        vmulsd xmm2,xmm0,xmm1           ;val = a[i] * b[i]

; Calculate c[i] = (val >= 0.0) ? sqrt(val) : val * val
        vcmpsd k1,xmm2,xmm5,CMP_GE      ;k1[0] = 1 if val >= 0.0
        vsqrtsd xmm3{k1}{z},xmm3,xmm2   ;xmm3 = (val > 0.0) ? sqrt(val) : 0.0
        knotw k2,k1                     ;k2[0] = 1 if val < 0.0
        vmulsd xmm4{k2}{z},xmm2,xmm2    ;xmm4 = (val < 0.0) ? val * val : 0.0
        vorpd xmm0,xmm4,xmm3            ;xmm0 = (val >= 0.0) ? sqrt(val) : val * val
        vmovsd real8 ptr [rcx+rax*8],xmm0   ;save result to c[i]

; Update index i and repeat until done
```

```
        inc rax                         ;i += 1
        cmp rax,r9
        jl @B
        mov eax,1                      ;set success return code

Done:   ret
Avx512CalcValues_ endp
        end
;-------------------------------------------------
;####### Ch13_03.asm
;-------------------------------------------------
; extern "C" void Avx512CvtF32ToU32_(uint32_t val_cvt[4], float val);

        .code
Avx512CvtF32ToU32_ proc
        vcvtss2usi eax,xmm1{rn-sae}     ;Convert using round to nearest
        mov dword ptr [rcx],eax

        vcvtss2usi eax,xmm1{rd-sae}     ;Convert using round down
        mov dword ptr [rcx+4],eax

        vcvtss2usi eax,xmm1{ru-sae}     ;Convert using round up
        mov dword ptr [rcx+8],eax

        vcvtss2usi eax,xmm1{rz-sae}     ;Convert using round to zero (truncate)
        mov dword ptr [rcx+12],eax
        ret
Avx512CvtF32ToU32_ endp

; extern "C" void Avx512CvtF64ToU64_(uint64_t val_cvt[4], double val);

Avx512CvtF64ToU64_ proc
        vcvtsd2usi rax,xmm1{rn-sae}
        mov qword ptr [rcx],rax

        vcvtsd2usi rax,xmm1{rd-sae}
        mov qword ptr [rcx+8],rax

        vcvtsd2usi rax,xmm1{ru-sae}
        mov qword ptr [rcx+16],rax

        vcvtsd2usi rax,xmm1{rz-sae}
        mov qword ptr [rcx+24],rax
        ret
Avx512CvtF64ToU64_ endp

; extern "C" void Avx512CvtF64ToF32_(float val_cvt[4], double val);

Avx512CvtF64ToF32_ proc
        vcvtsd2ss xmm2,xmm2,xmm1{rn-sae}
        vmovss real4 ptr [rcx],xmm2

        vcvtsd2ss xmm2,xmm2,xmm1{rd-sae}
        vmovss real4 ptr [rcx+4],xmm2

        vcvtsd2ss xmm2,xmm2,xmm1{ru-sae}
        vmovss real4 ptr [rcx+8],xmm2

        vcvtsd2ss xmm2,xmm2,xmm1{rz-sae}
        vmovss real4 ptr [rcx+12],xmm2
        ret
Avx512CvtF64ToF32_ endp
        end
;-------------------------------------------------
;####### Ch13_04.asm
;-------------------------------------------------
; Mask values used to calculate floating-point absolute values
```

```
ConstVals   segment readonly align(64) 'const'
AbsMaskF32  dword 16 dup(7fffffffh)
AbsMaskF64  qword 8 dup(7fffffffffffffffh)
ConstVals   ends
; extern "C" void Avx512PackedMathF32_(const ZmmVal* a, const ZmmVal* b, ZmmVal c[8]);
            .code
Avx512PackedMathF32_ proc

; Load packed SP floating-point values
        vmovaps zmm0,zmmword ptr [rcx]  ;zmm0 = *a
        vmovaps zmm1,zmmword ptr [rdx]  ;zmm1 = *b

; Packed SP floating-point addition
        vaddps zmm2,zmm0,zmm1
        vmovaps zmmword ptr [r8+0],zmm2

; Packed SP floating-point subtraction
        vsubps zmm2,zmm0,zmm1
        vmovaps zmmword ptr [r8+64],zmm2

; Packed SP floating-point multiplication
        vmulps zmm2,zmm0,zmm1
        vmovaps zmmword ptr [r8+128],zmm2

; Packed SP floating-point division
        vdivps zmm2,zmm0,zmm1
        vmovaps zmmword ptr [r8+192],zmm2

; Packed SP floating-point absolute value (b)
        vandps zmm2,zmm1,zmmword ptr [AbsMaskF32]
        vmovaps zmmword ptr [r8+256],zmm2

; Packed SP floating-point square root (a)
        vsqrtps zmm2,zmm0
        vmovaps zmmword ptr [r8+320],zmm2

; Packed SP floating-point minimum
        vminps zmm2,zmm0,zmm1
        vmovaps zmmword ptr [r8+384],zmm2

; Packed SP floating-point maximum
        vmaxps zmm2,zmm0,zmm1
        vmovaps zmmword ptr [r8+448],zmm2

        vzeroupper
        ret
Avx512PackedMathF32_ endp

; extern "C" void Avx512PackedMathF64_(const ZmmVal* a, const ZmmVal* b, ZmmVal c[8]);

Avx512PackedMathF64_ proc

; Load packed DP floating-point values
        vmovapd zmm0,zmmword ptr [rcx]   ;zmm0 = *a
        vmovapd zmm1,zmmword ptr [rdx]   ;zmm1 = *b

; Packed DP floating-point addition
        vaddpd zmm2,zmm0,zmm1
        vmovapd zmmword ptr [r8+0],zmm2

; Packed DP floating-point subtraction
        vsubpd zmm2,zmm0,zmm1
        vmovapd zmmword ptr [r8+64],zmm2

; Packed DP floating-point multiplication
        vmulpd zmm2,zmm0,zmm1
        vmovapd zmmword ptr [r8+128],zmm2
```

```
; Packed DP floating-point division
        vdivpd zmm2,zmm0,zmm1
        vmovapd zmmword ptr [r8+192],zmm2

; Packed DP floating-point absolute value (b)
        vandpd zmm2,zmm1,zmmword ptr [AbsMaskF64]
        vmovapd zmmword ptr [r8+256],zmm2

; Packed DP floating-point square root (a)
        vsqrtpd zmm2,zmm0
        vmovapd zmmword ptr [r8+320],zmm2

; Packed DP floating-point minimum
        vminpd zmm2,zmm0,zmm1
        vmovapd zmmword ptr [r8+384],zmm2

; Packed DP floating-point maximum
        vmaxpd zmm2,zmm0,zmm1
        vmovapd zmmword ptr [r8+448],zmm2
        vzeroupper
        ret
Avx512PackedMathF64_ endp
        end;
-------------------------------------------------
;####### Ch13_05.asm
        include <cmpequ.asmh>

; extern "C" void Avx512PackedCompareF32_(const ZmmVal* a, const ZmmVal* b, ZmmVal c[8]);

        .code
Avx512PackedCompareF32_ proc
        vmovaps zmm0,[rcx]                ;zmm0 = a
        vmovaps zmm1,[rdx]                ;zmm1 = b

; Perform packed EQUAL compare
        vcmpps k1,zmm0,zmm1,CMP_EQ
        kmovw word ptr [r8],k1

; Perform packed NOT EQUAL compare
        vcmpps k1,zmm0,zmm1,CMP_NEQ
        kmovw word ptr [r8+2],k1

; Perform packed LESS THAN compare
        vcmpps k1,zmm0,zmm1,CMP_LT
        kmovw word ptr [r8+4],k1

; Perform packed LESS THAN OR EQUAL compare
        vcmpps k1,zmm0,zmm1,CMP_LE
        kmovw word ptr [r8+6],k1

; Perform packed GREATER THAN compare
        vcmpps k1,zmm0,zmm1,CMP_GT
        kmovw word ptr [r8+8],k1

; Perform packed GREATER THAN OR EQUAL compare
        vcmpps k1,zmm0,zmm1,CMP_GE
        kmovw word ptr [r8+10],k1

; Perform packed ORDERED compare
        vcmpps k1,zmm0,zmm1,CMP_ORD
        kmovw word ptr [r8+12],k1

; Perform packed UNORDERED compare
        vcmpps k1,zmm0,zmm1,CMP_UNORD
        kmovw word ptr [r8+14],k1
        vzeroupper
        ret
Avx512PackedCompareF32_ endp
        end
;-------------------------------------------------
;####### Ch13_06.asm
        include <cmpequ.asmh> include <MacrosX86-64-AVX.asmh>

        extern c_NumRowsMax:qword
        extern c_NumColsMax:qword

; extern "C" bool Avx512CalcColumnMeans_(const double* x, size_t nrows, size_t ncols, double*
col_means, size_t* col_counts, double x_min);

        .code
Avx512CalcColumnMeans_ proc frame
        _CreateFrame CCM_,0,0,rbx,r12,r13
        _EndProlog

; Validate nrows and ncols
        xor eax,eax                     ;set error return code
        test rdx,rdx
        jz Done                         ;jump if nrows is zero
        cmp rdx,[c_NumRowsMax]
        ja Done                         ;jump if nrows is too large
        test r8,r8
        jz Done                         ;jump if ncols is zero
        cmp r8,[c_NumColsMax]
        ja Done                         ;jump if ncols is too large

; Load argument values col_counts and x_min
        mov ebx,1
        vpbroadcastq zmm4,rbx           ;zmm4 = 8 qwords of 1
        mov rbx,[rbp+CCM_OffsetStackArgs]    ;rbx = col_counts ptr
        lea r13,[rbp+CCM_OffsetStackArgs+8] ;r13 = ptr to x_min

; Set initial col_means and col_counts to zero
        xor r10,r10
        vxorpd xmm0,xmm0,xmm0
@@:     vmovsd real8 ptr[r9+rax*8],xmm0      ;col_means[i] = 0.0
        mov [rbx+rax*8],r10                  ;col_counts[i] = 0
        inc rax
        cmp rax,r8
        jne @B                               ;repeat until done

; Compute the sum of each column in x
LP1:    xor r10,r10                          ;r10 = col_index
        mov r11,r9                           ;r11 = ptr to col_means
        mov r12,rbx                          ;r12 = ptr to col_counts

LP2:    mov rax,r10                          ;rax = col_index
        add rax,8
        cmp rax,r8                           ;8 or more columns remaining?
        ja @F                                ;jump if col_index + 8 > ncols

; Update col_means and col_counts using next eight columns
        vmovupd zmm0,zmmword ptr [rcx]       ;load next 8 cols of cur row
        vcmppd k1,zmm0,real8 bcst [r13],CMP_GE  ;k1 = mask of values >= x_min
        vmovupd zmm1{k1}{z},zmm0             ;values >= x_min or 0.0
        vaddpd zmm2,zmm1,zmmword ptr [r11]   ;add values to col_means
        vmovupd zmmword ptr [r11],zmm2       ;save updated col_means

        vpmovm2q zmm0,k1                     ;convert mask to vector
        vpandq zmm1,zmm0,zmm4                ;qword values for add
        vpaddq zmm2,zmm1,zmmword ptr [r12]   ;update col_counts
        vmovdqu64 zmmword ptr [r12],zmm2     ;save updated col_counts
```

```
        add r10,8                            ;col_index += 8
        add rcx,64                           ;x += 8
        add r11,64                           ;col_means += 8
        add r12,64                           ;col_counts += 8
        jmp NextColSet

; Update col_means and col_counts using next four columns
@@:     sub rax,4
        cmp rax,r8                           ;4 or more columns remaining?
        ja @F                                ;jump if col_index + 4 > ncols

        vmovupd ymm0,ymmword ptr [rcx]       ;load next 4 cols of cur row
        vcmppd k1,ymm0,real8 bcst [r13],CMP_GE  ;k1 = mask of values >= x_min
        vmovupd ymm1{k1}{z},ymm0             ;values >= x_min or 0.0
        vaddpd ymm2,ymm1,ymmword ptr [r11]   ;add values to col_means
        vmovupd ymmword ptr [r11],ymm2       ;save updated col_means

        vpmovm2q ymm0,k1                     ;convert mask to vector
        vpandq ymm1,ymm0,ymm4                ;qword values for add
        vpaddq ymm2,ymm1,ymmword ptr [r12]   ;update col_counts
        vmovdqu64 ymmword ptr [r12],ymm2     ;save updated col_counts

        add r10,4                            ;col_index += 4
        add rcx,32                           ;x += 4
        add r11,32                           ;col_means += 4
        add r12,32                           ;col_counts += 4
        jmp NextColSet

; Update col_means and col_counts using next two columns
@@:     sub rax,2
        cmp rax,r8                           ;2 or more columns remaining?
        ja @F                                ;jump if col_index + 2 > ncols

        vmovupd xmm0,xmmword ptr [rcx]       ;load next 2 cols of cur row
        vcmppd k1,xmm0,real8 bcst [r13],CMP_GE  ;k1 = mask of values >= x_min
        vmovupd xmm1{k1}{z},xmm0             ;values >= x_min or 0.0
        vaddpd xmm2,xmm1,xmmword ptr [r11]   ;add values to col_means
        vmovupd xmmword ptr [r11],xmm2       ;save updated col_means

        vpmovm2q xmm0,k1                     ;convert mask to vector
        vpandq xmm1,xmm0,xmm4                ;qword values for add
        vpaddq xmm2,xmm1,xmmword ptr [r12]   ;update col_counts
        vmovdqu64 xmmword ptr [r12],xmm2     ;save updated col_counts

        add r10,2                            ;col_index += 2
        add rcx,16                           ;x += 2
        add r11,16                           ;col_means += 2
        add r12,16                           ;col_counts += 2
        jmp NextColSet

; Update col_means using last column of current row
@@:     vmovsd xmm0,real8 ptr [rcx]          ;load x from last column
        vcmpsd k1,xmm0,real8 ptr [r13],CMP_GE   ;k1 = mask of values >= x_min
        vmovsd xmm1{k1}{z},xmm1,xmm0         ;value or 0.0
        vaddsd xmm2,xmm1,real8 ptr [r11]     ;add to col_means
        vmovsd real8 ptr [r11],xmm2          ;save updated col_means
        kmovb eax,k1                         ;eax = 0 or 1
        add qword ptr [r12],rax              ;update col_counts

        add r10,1                            ;col_index += 1
        add rcx,8                            ;update x ptr

NextColSet:
        cmp r10,r8                       ;more columns in current row?
        jb LP2                           ;jump if yes
        dec rdx                          ;nrows -= 1
        jnz LP1                          ;jump if more rows
```

```
; Compute the final col_means
@@:     vmovsd xmm0,real8 ptr [r9]      ;xmm0 = col_means[i]
        vcvtsi2sd xmm1,xmm1,qword ptr [rbx] ;xmm1 = col_counts[i]
        vdivsd xmm2,xmm0,xmm1           ;compute final mean
        vmovsd real8 ptr [r9],xmm2      ;save col_mean[i]
        add r9,8                        ;update col_means ptr
        add rbx,8                       ;update col_counts ptr
        sub r8,1                        ;ncols -= 1
        jnz @B                          ;repeat until done

        mov eax,1                       ;set success return code

Done:   _DeleteFrame rbx,r12,r13
        vzeroupper
        ret

Avx512CalcColumnMeans_ endp
        end
;-------------------------------------------------
;####### Ch13_07.asm
        include <MacrosX86-64-AVX.asmh>

; Indices for gather and scatter instructions
ConstVals   segment readonly align(64) 'const'
GS_X        qword 0, 3, 6,  9, 12, 15, 18, 21
GS_Y        qword 1, 4, 7, 10, 13, 16, 19, 22
GS_Z        qword 2, 5, 8, 11, 14, 17, 20, 23
ConstVals   ends

; extern "C" bool Avx512VcpAos_(Vector* c, const Vector* a, const Vector* b, size_t num_vectors);

        .code
Avx512VcpAos_ proc

; Make sure num_vec is valid
        xor eax,eax                     ;set error code (also i = 0)
        test r9,r9
        jz Done                         ;jump if num_vec is zero
        test r9,07h
        jnz Done                        ;jump if num_vec % 8 != 0 is true

; Load indices for gather and scatter operations
        vmovdqa64 zmm29,zmmword ptr [GS_X]  ;zmm29 = X component indices
        vmovdqa64 zmm30,zmmword ptr [GS_Y]  ;zmm30 = Y component indices
        vmovdqa64 zmm31,zmmword ptr [GS_Z]  ;zmm31 = Z component indices

; Load next 8 vectors
        align 16
@@:     kxnorb k1,k1,k1
        vgatherqpd zmm0{k1},[rdx+zmm29*8]   ;zmm0 = A.X values

        kxnorb k2,k2,k2
        vgatherqpd zmm1{k2},[rdx+zmm30*8]   ;zmm1 = A.Y values

        kxnorb k3,k3,k3
        vgatherqpd zmm2{k3},[rdx+zmm31*8]   ;zmm2 = A.Z values

        kxnorb k4,k4,k4
        vgatherqpd zmm3{k4},[r8+zmm29*8]    ;zmm3 = B.X values

        kxnorb k5,k5,k5
        vgatherqpd zmm4{k5},[r8+zmm30*8]    ;zmm4 = B.Y values

        kxnorb k6,k6,k6
        vgatherqpd zmm5{k6},[r8+zmm31*8]    ;zmm5 = B.Z values
```

```
; Calculate 8 vector cross products
        vmulpd zmm16,zmm1,zmm5
        vmulpd zmm17,zmm2,zmm4
        vsubpd zmm18,zmm16,zmm17            ;c.X = a.Y * b.Z - a.Z * b.Y

        vmulpd zmm19,zmm2,zmm3
        vmulpd zmm20,zmm0,zmm5
        vsubpd zmm21,zmm19,zmm20            ;c.Y = a.Z * b.X - a.X * b.Z

        vmulpd zmm22,zmm0,zmm4
        vmulpd zmm23,zmm1,zmm3
        vsubpd zmm24,zmm22,zmm23            ;c.Z = a.X * b.Y - a.Y * b.X

; Save calculated cross products
        kxnorb k4,k4,k4
        vscatterqpd [rcx+zmm29*8]{k4},zmm18 ;save C.X components

        kxnorb k5,k5,k5
        vscatterqpd [rcx+zmm30*8]{k5},zmm21 ;save C.Y components

        kxnorb k6,k6,k6
        vscatterqpd [rcx+zmm31*8]{k6},zmm24 ;save C.Z components

; Update pointers and counters
        add rcx,192                         ;c += 8
        add rdx,192                         ;a += 8
        add r8,192                          ;b += 8
        add rax,8                           ;i += 8
        cmp rax,r9
        jb @B

        mov eax,1                           ;set success return code

Done:   vzeroupper
        ret
Avx512VcpAos_ endp

; extern "C" bool Avx512VcpSoa_(VectorSoA* c, const VectorSoA* a, const VectorSoA* b, size_t
num_vectors);

Avx512VcpSoa_ proc frame
        _CreateFrame CP2_,0,0,rbx,rsi,rdi,r12,r13,r14,r15
        _EndProlog

; Make sure num_vec is valid
        xor eax,eax
        test r9,r9
        jz Done                         ;jump if num_vec is zero
        test r9,07h
        jnz Done                        ;jump if num_vec % 8 != 0 is true

; Load vector array pointers and check for proper alignment
        mov r10,[rdx]                   ;r10 = a.X
        or rax,r10
        mov r11,[rdx+8]                 ;r11 = a.Y
        or rax,r11
        mov r12,[rdx+16]                ;r12 = a.Z
        or rax,r12

        mov r13,[r8]                    ;r13 = b.X
        or rax,r13
        mov r14,[r8+8]                  ;r14 = b.Y
        or rax,r14
        mov r15,[r8+16]                 ;r15 = b.Z
        or rax,r15

        mov rbx,[rcx]                   ;rbx = c.X
        or rax,rbx
        mov rsi,[rcx+8]                 ;rsi = c.Y
        or rax,rsi
        mov rdi,[rcx+16]                ;rdi = c.Z
        or rax,rdi

        and rax,3fh                     ;misaligned component array?
        mov eax,0                       ;error return code (also i = 0)
        jnz Done

; Load next block (8 vectors) from a and b
        align 16
@@:     vmovapd zmm0,zmmword ptr [r10+rax*8]    ;zmm0 = a.X values
        vmovapd zmm1,zmmword ptr [r11+rax*8]    ;zmm1 = a.Y values
        vmovapd zmm2,zmmword ptr [r12+rax*8]    ;zmm2 = a.Z values
        vmovapd zmm3,zmmword ptr [r13+rax*8]    ;zmm3 = b.X values
        vmovapd zmm4,zmmword ptr [r14+rax*8]    ;zmm4 = b.Y values
        vmovapd zmm5,zmmword ptr [r15+rax*8]    ;zmm5 = b.Z values

; Calculate cross products
        vmulpd zmm16,zmm1,zmm5
        vmulpd zmm17,zmm2,zmm4
        vsubpd zmm18,zmm16,zmm17            ;c.X = a.Y * b.Z - a.Z * b.Y

        vmulpd zmm19,zmm2,zmm3
        vmulpd zmm20,zmm0,zmm5
        vsubpd zmm21,zmm19,zmm20            ;c.Y = a.Z * b.X - a.X * b.Z

        vmulpd zmm22,zmm0,zmm4
        vmulpd zmm23,zmm1,zmm3
        vsubpd zmm24,zmm22,zmm23            ;c.Z = a.X * b.Y - a.Y * b.X

; Save calculated cross products
        vmovapd zmmword ptr [rbx+rax*8],zmm18   ;save C.X values
        vmovapd zmmword ptr [rsi+rax*8],zmm21   ;save C.Y values
        vmovapd zmmword ptr [rdi+rax*8],zmm24   ;save C.Z values

        add rax,8                           ;i += 8
        cmp rax,r9
        jb @B                               ;repeat until done

Done:   vzeroupper
        _DeleteFrame rbx,rsi,rdi,r12,r13,r14,r15
        ret
Avx512VcpSoa_ endp
        end
;-------------------------------------------------
;####### Ch13_08.asm
;-------------------------------------------------
ConstVals   segment readonly align(64) 'const'
; Indices for matrix permutations
MatPerm0    dword 0, 4, 8, 12, 0, 4, 8, 12, 0, 4, 8, 12, 0, 4, 8, 12
MatPerm1    dword 1, 5, 9, 13, 1, 5, 9, 13, 1, 5, 9, 13, 1, 5, 9, 13
MatPerm2    dword 2, 6, 10, 14, 2, 6, 10, 14, 2, 6, 10, 14, 2, 6, 10, 14
MatPerm3    dword 3, 7, 11, 15, 3, 7, 11, 15, 3, 7, 11, 15, 3, 7, 11, 15
; Indices for vector permutations
VecPerm0    dword 0, 0, 0, 0, 4, 4, 4, 4, 8, 8, 8, 8, 12, 12, 12, 12
VecPerm1    dword 1, 1, 1, 1, 5, 5, 5, 5, 9, 9, 9, 9, 13, 13, 13, 13
VecPerm2    dword 2, 2, 2, 2, 6, 6, 6, 6, 10, 10, 10, 10, 14, 14, 14, 14
VecPerm3    dword 3, 3, 3, 3, 7, 7, 7, 7, 11, 11, 11, 11, 15, 15, 15, 15
ConstVals   ends

; extern "C" bool Avx512MatVecMulF32_(Vec4x1_F32* vec_b, float mat[4][4], Vec4x1_F32* vec_a,
size_t num_vec);

        .code
Avx512MatVecMulF32_ proc
        xor eax,eax                     ;set error code (also i = 0)
        test r9,r9
        jz Done                         ;jump if num_vec is zero
        test r9,3
        jnz Done                        ;jump if n % 4 != 0

        test rcx,3fh
        jnz Done                        ;jump if vec_b is not properly aligned
        test rdx,3fh
        jnz Done                        ;jump if mat is not properly aligned
        test r8,3fh
        jnz Done                        ;jump if vec_a is not properly aligned

; Load permutation indices for matrix columns and vector elements
        vmovdqa32 zmm16,zmmword ptr [MatPerm0]  ;mat col 0 indices
        vmovdqa32 zmm17,zmmword ptr [MatPerm1]  ;mat col 1 indices
        vmovdqa32 zmm18,zmmword ptr [MatPerm2]  ;mat col 2 indices
        vmovdqa32 zmm19,zmmword ptr [MatPerm3]  ;mat col 3 indices

        vmovdqa32 zmm24,zmmword ptr [VecPerm0]  ;W component indices
        vmovdqa32 zmm25,zmmword ptr [VecPerm1]  ;X component indices
        vmovdqa32 zmm26,zmmword ptr [VecPerm2]  ;Y component indices
        vmovdqa32 zmm27,zmmword ptr [VecPerm3]  ;Z component indices

; Load source matrix and duplicate columns
        vmovaps zmm0,zmmword ptr [rdx]  ;zmm0 = mat

        vpermps zmm20,zmm16,zmm0        ;zmm20 = mat col 0 (4x)
        vpermps zmm21,zmm17,zmm0        ;zmm21 = mat col 1 (4x)
        vpermps zmm22,zmm18,zmm0        ;zmm22 = mat col 2 (4x)
        vpermps zmm23,zmm19,zmm0        ;zmm23 = mat col 3 (4x)
; Load the next 4 vectors
        align 16
@@:     vmovaps zmm4,zmmword ptr [r8+rax]   ;zmm4 = vec_a (4 vectors)
; Permute the vector elements for subsequent calculations
        vpermps zmm0,zmm24,zmm4         ;zmm0 = vec_a W components
        vpermps zmm1,zmm25,zmm4         ;zmm1 = vec_a X components
        vpermps zmm2,zmm26,zmm4         ;zmm2 = vec_a Y components
        vpermps zmm3,zmm27,zmm4         ;zmm3 = vec_a Z components

; Perform matrix-vector multiplications (4 vectors)
        vmulps zmm28,zmm20,zmm0
        vmulps zmm29,zmm21,zmm1
        vmulps zmm30,zmm22,zmm2
        vmulps zmm31,zmm23,zmm3
        vaddps zmm4,zmm28,zmm29
        vaddps zmm5,zmm30,zmm31
        vaddps zmm4,zmm4,zmm5           ;zmm4 = vec_b (4 vectors)
        vmovaps zmmword ptr [rcx+rax],zmm4  ;save result

        add rax,64                      ;rax = offset to next block of 4 vectors
        sub r9,4
        jnz @B                          ;repeat until done

        mov eax,1                       ;set success code

Done:   vzeroupper
        ret
Avx512MatVecMulF32_ endp
        end;
-------------------------------------------------
;####### Ch13_09_.asm
        include <MacrosX86-64-AVX.asmh>
        extern c_NumPtsMin:dword
        extern c_NumPtsMax:dword
        extern c_KernelSizeMin:dword
        extern c_KernelSizeMax:dword
```

```
; extern bool Avx512Convolve2_(float* y, const float* x, int num_pts, const float* kernel, int
kernel_size)
        .code
Avx512Convolve2_ proc frame
        _CreateFrame CV2_,0,0,rbx
        _EndProlog

; Validate argument values
        xor eax,eax                     ;set error code

        mov r10d,dword ptr [rbp+CV2_OffsetStackArgs]
        test r10d,1
        jz Done                         ;kernel_size is even
        cmp r10d,[c_KernelSizeMin]
        jl Done                         ;kernel_size too small
        cmp r10d,[c_KernelSizeMax]
        jg Done                         ;kernel_size too big

        cmp r8d,[c_NumPtsMin]
        jl Done                         ;num_pts too small
        cmp r8d,[c_NumPtsMax]
        jg Done                         ;num_pts too big
        test r8d,15
        jnz Done                        ;num_pts not even multiple of 16

        test rcx,3fh
        jnz Done                        ;y is not properly aligned

; Initialize convolution loop variables
        shr r10d,1                      ;r10 = kernel_size / 2 (ks2)
        lea rdx,[rdx+r10*4]             ;rdx = x + ks2 (first data point)
        xor ebx,ebx                     ;i = 0

; Perform convolution
LP1:    vxorps zmm0,zmm0,zmm0           ;packed sum = 0.0;
        mov r11,r10                     ;r11 = ks2
        neg r11                         ;k = -ks2

LP2:    mov rax,rbx                         ;rax = i
        sub rax,r11                         ;rax = i - k
        vmovups zmm1,zmmword ptr [rdx+rax*4]    ;load x[i - k]:x[i - k + 15]

        mov rax,r11
        add rax,r10                         ;rax = k + ks2
        vbroadcastss zmm2,real4 ptr [r9+rax*4]  ;zmm2 = kernel[k + ks2]
        vfmadd231ps zmm0,zmm1,zmm2          ;zmm0 += x[i-k]:x[i-k+15] * kernel[k+ks2]

        add r11,1                           ;k += 1
        cmp r11,r10
        jle LP2                             ;repeat until k > ks2

        vmovaps zmmword ptr [rcx+rbx*4],zmm0    ;save y[i]:y[i + 15]

        add rbx,16                       ;i += 16
        cmp rbx,r8
        jl LP1                          ;repeat until done
        mov eax,1                       ;set success return code

Done:   vzeroupper
        _DeleteFrame rbx
        ret
Avx512Convolve2_ endp

; extern bool Avx512Convolve2Ks5_(float* y, const float* x, int num_pts, const float* kernel, int kernel_size)
Avx512Convolve2Ks5_ proc frame
        _CreateFrame CKS5_,0,48
        _SaveXmmRegs xmm6,xmm7,xmm8
        _EndProlog

; Validate argument values
        xor eax,eax                     ;set error code (rax is also loop index var)
        cmp dword ptr [rbp+CKS5_OffsetStackArgs],5
        jne Done                        ;jump if kernel_size is not 5

        cmp r8d,[c_NumPtsMin]
        jl Done                         ;jump if num_pts too small
        cmp r8d,[c_NumPtsMax]
        jg Done                         ;jump if num_pts too big
        test r8d,15
        jnz Done                        ;num_pts not even multiple of 15

        test rcx,3fh
        jnz Done                        ;y is not properly aligned

; Perform required initializations
        vbroadcastss zmm4,real4 ptr [r9]    ;kernel[0]
        vbroadcastss zmm5,real4 ptr [r9+4]  ;kernel[1]
        vbroadcastss zmm6,real4 ptr [r9+8]  ;kernel[2]
        vbroadcastss zmm7,real4 ptr [r9+12] ;kernel[3]
        vbroadcastss zmm8,real4 ptr [r9+16] ;kernel[4]

        mov r8d,r8d                         ;r8 = num_pts
        add rdx,8                           ;x += 2

; Perform convolution
@@:     vxorps zmm2,zmm2,zmm2               ;initialize sum vars
        vxorps zmm3,zmm3,zmm3

        mov r11,rax
        add r11,2                           ;j = i + ks2

        vmovups zmm0,zmmword ptr [rdx+r11*4]    ;zmm0 = x[j]:x[j + 15]
        vfmadd231ps zmm2,zmm0,zmm4              ;zmm2 += x[j]:x[j + 15] * kernel[0]

        vmovups zmm1,zmmword ptr [rdx+r11*4-4]  ;zmm1 = x[j - 1]:x[j + 14]
        vfmadd231ps zmm3,zmm1,zmm5              ;zmm3 += x[j - 1]:x[j + 14] * kernel[1]

        vmovups zmm0,zmmword ptr [rdx+r11*4-8]  ;zmm0 = x[j - 2]:x[j + 13]
        vfmadd231ps zmm2,zmm0,zmm6              ;zmm2 += x[j - 2]:x[j + 13] * kernel[2]

        vmovups zmm1,zmmword ptr [rdx+r11*4-12] ;zmm1 = x[j - 3]:x[j + 12]
        vfmadd231ps zmm3,zmm1,zmm7              ;zmm3 += x[j - 3]:x[j + 12] * kernel[3]

        vmovups zmm0,zmmword ptr [rdx+r11*4-16] ;zmm0 = x[j - 4]:x[j + 11]
        vfmadd231ps zmm2,zmm0,zmm8              ;zmm2 += x[j - 4]:x[j + 11] * kernel[4]

        vaddps zmm0,zmm2,zmm3                   ;final values
        vmovaps zmmword ptr [rcx+rax*4],zmm0    ;save y[i]:y[i + 15]

        add rax,16                           ;i += 16
        cmp rax,r8
        jl @B                               ;jump if i < num_pts
        mov eax,1                           ;set success return code

Done:   vzeroupper
        _RestoreXmmRegs xmm6,xmm7,xmm8
        _DeleteFrame
        ret
Avx512Convolve2Ks5_ endp
        end
;-------------------------------------------------
;####### Ch14_01.asm
;-------------------------------------------------
```

```
; extern ″C″ void Avx512PackedMathI16_(const ZmmVal* a, const ZmmVal* b, ZmmVal c[6])
        .code
Avx512PackedMathI16_ proc
        vmovdqu16 zmm0,zmmword ptr [rcx]    ;zmm0 = a
        vmovdqu16 zmm1,zmmword ptr [rdx]    ;zmm1 = b

; Perform packed word operations
        vpaddw zmm2,zmm0,zmm1               ;add
        vmovdqa64 zmmword ptr [r8],zmm2     ;save vpaddw result

        vpaddsw zmm2,zmm0,zmm1              ;add with signed saturation
        vmovdqa64 zmmword ptr [r8+64],zmm2  ;save vpaddsw result

        vpsubw zmm2,zmm0,zmm1               ;sub
        vmovdqa64 zmmword ptr [r8+128],zmm2 ;save vpsubw result

        vpsubsw zmm2,zmm0,zmm1              ;sub with signed saturation
        vmovdqa64 zmmword ptr [r8+192],zmm2 ;save vpsubsw result

        vpminsw zmm2,zmm0,zmm1              ;signed minimums
        vmovdqa64 zmmword ptr [r8+256],zmm2 ;save vpminsw result

        vpmaxsw zmm2,zmm0,zmm1              ;signed maximums
        vmovdqa64 zmmword ptr [r8+320],zmm2 ;save vpmaxsw result

        vzeroupper
        ret
Avx512PackedMathI16_ endp

extern ″C″ void Avx512PackedMathI64_(const ZmmVal* a, const ZmmVal* b, ZmmVal c[5], unsigned int opmask)
Avx512PackedMathI64_ proc
        vmovdqa64 zmm0,zmmword ptr [rcx]    ;zmm0 = a
        vmovdqa64 zmm1,zmmword ptr [rdx]    ;zmm1 = b

        and r9d,0ffh                        ;r9d = opmask value
        kmovb k1,r9d                        ;k1 = opmask

; Perform packed quadword operations
        vpaddq zmm2{k1}{z},zmm0,zmm1        ;add
        vmovdqa64 zmmword ptr [r8],zmm2     ;save vpaddq result

        vpsubq zmm2{k1}{z},zmm0,zmm1        ;sub
        vmovdqa64 zmmword ptr [r8+64],zmm2  ;save vpsubq result

        vpmullq zmm2{k1}{z},zmm0,zmm1       ;signed mul (low 64 bits)
        vmovdqa64 zmmword ptr [r8+128],zmm2 ;save vpmullq result

        vpsllvq zmm2{k1}{z},zmm0,zmm1       ;shift left logical
        vmovdqa64 zmmword ptr [r8+192],zmm2 ;save vpsllvq result

        vpsravq zmm2{k1}{z},zmm0,zmm1       ;shift right arithmetic
        vmovdqa64 zmmword ptr [r8+256],zmm2 ;save vpsravq result

        vpabsq zmm2{k1}{z},zmm0             ;absolute value
        vmovdqa64 zmmword ptr [r8+320],zmm2 ;save vpabsq result

        vzeroupper
        ret
Avx512PackedMathI64_ endp
        end
;---------------------------------------------
;####### Ch14_02.asm
        include <cmpequ.asmh>
        extern c_NumPixelsMax:dword

            .const
r4_1p0      real4 1.0
r4_255p0    real4 255.0

;extern ″C″ bool Avx512ConvertImgU8ToF32_(float* des, const uint8_t* src, uint32_t num_pixels)
        .code
Avx512ConvertImgU8ToF32_ proc

; Make sure num_pixels is valid and pixel buffers are properly aligned
        xor eax,eax                     ;set error return code
        or r8d,r8d
        jz Done                         ;jump if num_pixels is zero
        cmp r8d,[c_NumPixelsMax]
        ja Done                         ;jump if num_pixels too big
        test r8d,3fh
        jnz Done                        ;jump if num_pixels % 64 != 0
        test rcx,3fh
        jnz Done                        ;jump if des not aligned
        test rdx,3fh
        jnz Done                        ;jump if src not aligned

; Perform required initializations
        shr r8d,6                           ;number of blocks (64 pixels/block)
        vmovss xmm0,real4 ptr [r4_1p0]
        vdivss xmm1,xmm0,real4 ptr [r4_255p0]
        vbroadcastss zmm5,xmm1              ;packed scale factor (1.0 / 255.0)
        align 16
@@:     vpmovzxbd zmm0,xmmword ptr [rdx]
        vpmovzxbd zmm1,xmmword ptr [rdx+16]
        vpmovzxbd zmm2,xmmword ptr [rdx+32]
        vpmovzxbd zmm3,xmmword ptr [rdx+48] ;zmm3:zmm0 = 64 U32 pixels

; Convert pixels from uint8_t to float [0.0, 255.0]
        vcvtudq2ps zmm16,zmm0
        vcvtudq2ps zmm17,zmm1
        vcvtudq2ps zmm18,zmm2
        vcvtudq2ps zmm19,zmm3           ;zmm19:zmm16 = 64 F32 pixels

; Normalize pixels to [0.0, 1.0]
        vmulps zmm20,zmm16,zmm5
        vmulps zmm21,zmm17,zmm5
        vmulps zmm22,zmm18,zmm5
        vmulps zmm23,zmm19,zmm5         ;zmm23:zmm20 = 64 F32 pixels (normalized)
; Save F32 pixels to des
        vmovaps zmmword ptr [rcx],zmm20
        vmovaps zmmword ptr [rcx+64],zmm21
        vmovaps zmmword ptr [rcx+128],zmm22
        vmovaps zmmword ptr [rcx+192],zmm23

; Update pointers and counters
        add rdx,64
        add rcx,256
        sub r8d,1
        jnz @B

        mov eax,1                   ;set success return code

Done:   vzeroupper
        ret
Avx512ConvertImgU8ToF32_ endp

; extern ″C″ bool Avx512ConvertImgF32ToU8_(uint8_t* des, const float* src, uint32_t num_pixels)
Avx512ConvertImgF32ToU8_ proc
; Make sure num_pixels is valid and pixel buffers are properly aligned
        xor eax,eax                     ;set error return code
        or r8d,r8d
        jz Done                         ;jump if num_pixels is zero
        cmp r8d,[c_NumPixelsMax]
        ja Done                         ;jump if num_pixels too big
        test r8d,3fh
        jnz Done                        ;jump if num_pixels % 64 != 0
        test rcx,3fh
        jnz Done                        ;jump if des not aligned
        test rdx,3fh
        jnz Done                        ;jump if src not aligned

; Perform required initializations
        shr r8d,4                       ;number of pixel blocks (16 pixels / block)
        vxorps zmm29,zmm29,zmm29        ;packed 0.0
        vbroadcastss zmm30,[r4_1p0]     ;packed 1.0
        vbroadcastss zmm31,[r4_255p0]   ;packed 255.0

        align 16
@@:     vmovaps zmm0,zmmword ptr [rdx]  ;zmm0 = block of 16 pixels

; Clip pixels in current block to [0,0. 1.0]
        vcmpps k1,zmm0,zmm29,CMP_GE     ;k1 = mask of pixels >= 0.0
        vmovaps zmm1{k1}{z},zmm0        ;all pixels >= 0.0

        vcmpps k2,zmm1,zmm30,CMP_GT     ;k2 = mask of pixels > 1.0
        vmovaps zmm1{k2},zmm30          ;all pixels clipped to [0.0, 1.0]

; Convert pixels to uint8_t and save to des
        vmulps zmm2,zmm1,zmm31          ;all pixels [0.0, 255.0]
        vcvtps2udq zmm3,zmm2{ru-sae}    ;all pixels [0, 255]
        vpmovusdb xmmword ptr [rcx],zmm3    ;save pixels as unsigned bytes

; Update pointers and counters
        add rdx,64
        add rcx,16
        sub r8d,1
        jnz @B

        mov eax,1                       ;set success return code

Done:   vzeroupper
        ret
Avx512ConvertImgF32ToU8_ endp
        end
;-------------------------------------------------
;####### Ch14_03.asm
        include <cmpequ.asmh>
        extern c_NumPixelsMax:qword

; Macro CmpPixels

_CmpPixels macro CmpOp
        align 16
@@:     vmovdqa64 zmm0,zmmword ptr [rdx+rax]    ;load next block of 64 pixels
        vpcmpub k1,zmm0,zmm4,CmpOp          ;perform compare operation
        vmovdqu8 zmm1{k1}{z},zmm5           ;set mask pixels to 0 or 255 using opmask
        vmovdqa64 zmmword ptr [rcx+rax],zmm1    ;save mask pixels

        add rax,64                          ;update offset
        sub r8,64
        jnz @B                              ;repeat until done
        mov eax,1                           ;set success return code
        vzeroupper
        ret
        endm

; extern "C" bool Avx512ComparePixels_(uint8_t* des, const uint8_t* src,
;   size_t num_pixels, CmpOp cmp_op, uint8_t cmp_val);

        .code
Avx512ComparePixels_ proc
```

```
; Make sure num_pixels is valid and pixel buffers are properly aligned
        xor eax,eax                     ;set error code (also array offset)
        or r8,r8
        jz Done                         ;jump if num_pixels is zero
        cmp r8,[c_NumPixelsMax]
        ja Done                         ;jump if num_pixels too big
        test r8,3fh
        jnz Done                        ;jump if num_pixels % 64 != 0

        test rcx,3fh
        jnz Done                        ;jump if des not aligned
        test rdx,3fh
        jnz Done                        ;jump if src not aligned

; Perform required initializations
        vpbroadcastb zmm4,byte ptr [rsp+40] ;zmm4 = packed cmp_val
        mov r10d,255
        vpbroadcastb zmm5,r10d          ;zmm5 = packed 255

; Perform specified compare operation
        cmp r9d,0
        jne LB_NE
        _CmpPixels CMP_EQ               ;CmpOp::EQ

LB_NE:  cmp r9d,1
        jne LB_LT
        _CmpPixels CMP_NEQ              ;CmpOp::NE

LB_LT:  cmp r9d,2
        jne LB_LE
        _CmpPixels CMP_LT               ;CmpOp::LT

LB_LE:  cmp r9d,3
        jne LB_GT
        _CmpPixels CMP_LE               ;CmpOp::LE

LB_GT:  cmp r9d,4
        jne LB_GE
        _CmpPixels CMP_NLE              ;CmpOp::GT

LB_GE:  cmp r9d,5
        jne Done
        _CmpPixels CMP_NLT              ;CmpOp::GE

Done:   vzeroupper
        ret
Avx512ComparePixels_ endp
        end
;-------------------------------------------------
;####### Ch14_04.asm
        include <cmpequ.asmh>       include <MacrosX86-64-AVX.asmh>
        extern c_NumPixelsMax:qword

; This structure must match the structure that's defined in Ch14_04.h
ImageStats          struct
PixelBuffer         qword ?
NumPixels           qword ?
PixelValMin         dword ?
PixelValMax         dword ?
NumPixelsInRange    qword ?
PixelSum            qword ?
PixelSumOfSquares   qword ?
PixelMean           real8 ?
PixelSd             real8 ?
ImageStats          ends
```

```
_UpdateSums macro Disp
        vpmovzxbd zmm0,xmmword ptr [rcx+Disp]   ;zmm0 = 16 pixels
        vpcmpud k1,zmm0,zmm31,CMP_GE        ;k1 = mask of pixels >= pixel_val_min
        vpcmpud k2,zmm0,zmm30,CMP_LE        ;k2 = mask of pixels <= pixel_val_max
        kandw k3,k2,k1                      ;k3 = mask of in-range pixels
        vmovdqa32 zmm1{k3}{z},zmm0          ;zmm1 = in-range pixels
        vpaddd zmm16,zmm16,zmm1             ;update packed pixel_sum
        vpmulld zmm2,zmm1,zmm1
        vpaddd zmm17,zmm17,zmm2             ;update packed pixel_sum_of_squares
        kmovw rax,k3
        popcnt rax,rax                      ;count number of in-range pixels
        add r10,rax                         ;update num_pixels_in_range
        endm

; extern "C" bool Avx512CalcImageStats_(ImageStats& im_stats);

        .code
Avx512CalcImageStats_ proc frame
        _CreateFrame CIS_,0,0,rsi,r12,r13
        _EndProlog

; Make sure num_pixels is valid and pixel_buff is properly aligned
        xor eax,eax                     ;set error return code

        mov rsi,rcx                                ;rsi = im_stats ptr
        mov rcx,qword ptr [rsi+ImageStats.PixelBuffer] ;rcx = pixel buffer ptr
        mov rdx,qword ptr [rsi+ImageStats.NumPixels]   ;rdx = num_pixels

        test rdx,rdx
        jz Done                         ;jump if num_pixels is zero
        cmp rdx,[c_NumPixelsMax]
        ja Done                         ;jump if num_pixels too big

        test rcx,3fh
        jnz Done                        ;jump if pixel_buff misaligned

; Perform required initializations
        mov r8d,dword ptr [rsi+ImageStats.PixelValMin]
        mov r9d,dword ptr [rsi+ImageStats.PixelValMax]

        vpbroadcastd zmm31,r8d          ;packed pixel_val_min
        vpbroadcastd zmm30,r9d          ;packed pixel_val_max

        vpxorq zmm29,zmm29,zmm29        ;packed pixel_sum
        vpxorq zmm28,zmm28,zmm28        ;packed pixel_sum_of_squares
        xor r10d,r10d                   ;num_pixels_in_range = 0

; Compute packed versions of pixel_sum and pixel_sum_of_squares
        cmp rdx,64
        jb LB1                          ;jump if there are fewer than 64 pixels

        align 16
@@:     vpxord zmm16,zmm16,zmm16        ;loop packed pixel_sum = 0
        vpxord zmm17,zmm17,zmm17        ;loop packed pixel_sum_of_squares = 0

        _UpdateSums 0                   ;process pixel_buff[i+15]:pixel_buff[i]
        _UpdateSums 16                  ;process pixel_buff[i+31]:pixel_buff[i+16]
        _UpdateSums 32                  ;process pixel_buff[i+47]:pixel_buff[i+32]
        _UpdateSums 48                  ;process pixel_buff[i+63]:pixel_buff[i+48]

        vextracti32x8 ymm0,zmm16,1      ;extract top 8 pixel_sum (dwords)
        vpaddd ymm1,ymm0,ymm16
        vpmovzxdq zmm2,ymm1
        vpaddq zmm29,zmm29,zmm2         ;update packed pixel_sum (qwords)
        vextracti32x8 ymm0,zmm17,1      ;extract top 8 pixel_sum_of_squares (dwords)
        vpaddd ymm1,ymm0,ymm17
        vpmovzxdq zmm2,ymm1
```

```
        vpaddq zmm28,zmm28,zmm2         ;update packed pixel_sum_of_squares (qwords)
        add rcx,64                      ;update pb ptr
        sub rdx,64                      ;update num_pixels
        cmp rdx,64
        jae @B                          ;repeat until done

        align 16
LB1:    test rdx,rdx
        jz LB3                          ;jump if no more pixels remain

        xor r13,r13                     ;pixel_sum = 0
        xor r12,r12                     ;pixel_sum_of_squares = 0
        mov r11,rdx                     ;number of remaining pixels

@@:     movzx rax,byte ptr [rcx]        ;load next pixel
        cmp rax,r8
        jb LB2                          ;jump if current pixel < pval_min
        cmp rax,r9
        ja LB2                          ;jump if current pixel > pval_max

        add r13,rax                     ;add to pixel_sum
        mul rax
        add r12,rax                     ;add to pixel_sum_of_squares
        add r10,1                       ;update num_pixels_in_range

LB2:    add rcx,1
        sub r11,1
        jnz @B                          ;repeat until done

; Save num_pixel_in_range
LB3:    mov qword ptr [rsi+ImageStats.NumPixelsInRange],r10

; Reduce packed pixel_sum to single qword
        vextracti64x4 ymm0,zmm29,1
        vpaddq ymm1,ymm0,ymm29
        vextracti64x2 xmm2,ymm1,1
        vpaddq xmm3,xmm2,xmm1
        vpextrq rax,xmm3,0
        vpextrq r11,xmm3,1
        add rax,r11                     ;rax = sum of qwords in zmm29
        add r13,rax                     ;add scalar pixel_sum

        mov qword ptr [rsi+ImageStats.PixelSum],r13

;Reduce packed pixel_sum_of_squares to single qword
        vextracti64x4 ymm0,zmm28,1
        vpaddq ymm1,ymm0,ymm28
        vextracti64x2 xmm2,ymm1,1
        vpaddq xmm3,xmm2,xmm1
        vpextrq rax,xmm3,0
        vpextrq r11,xmm3,1
        add rax,r11                     ;rax = sum of qwords in zmm28
        add r12,rax                     ;add scalar pixel_sum_of_squares

        mov qword ptr [rsi+ImageStats.PixelSumOfSquares],r12

; Calculate final mean and sd
        vcvtusi2sd xmm0,xmm0,r10        ;num_pixels_in_range (DPFP)
        sub r10,1
        vcvtusi2sd xmm1,xmm1,r10        ;num_pixels_in_range - 1 (DPFP)
        vcvtusi2sd xmm2,xmm2,r13        ;pixel_sum (DPFP)
        vcvtusi2sd xmm3,xmm3,r12        ;pixel_sum_of_squares (DPFP)
        vdivsd xmm4,xmm2,xmm0           ;final pixel_mean

        vmovsd real8 ptr [rsi+ImageStats.PixelMean],xmm4

        vmulsd xmm4,xmm0,xmm3           ;num_pixels_in_range * pixel_sum_of_squares
        vmulsd xmm5,xmm2,xmm2           ;pixel_sum * pixel_sum
        vsubsd xmm2,xmm4,xmm5           ;var_num
        vmulsd xmm3,xmm0,xmm1           ;var_den
        vdivsd xmm4,xmm2,xmm3           ;calc variance
        vsqrtsd xmm0,xmm0,xmm4          ;final pixel_sd

        vmovsd real8 ptr [rsi+ImageStats.PixelSd],xmm0

        mov eax,1                       ;set success return code

Done:   vzeroupper
        _DeleteFrame rsi,r12,r13
        ret
Avx512CalcImageStats_ endp
        end
;-----------------------------------------------
;####### Ch14_05.asm
        include <MacrosX86-64-AVX.asmh>
        extern c_NumPixelsMin:dword
        extern c_NumPixelsMax:dword

            .const
r4_0p5      real4 0.5
r4_255p0    real4 255.0

; extern "C" bool Avx512RgbToGs_(uint8_t* pb_gs, const uint8_t* const* pb_rgb, int num_pixels,
const float coef[3]);

        .code
Avx512RgbToGs_ proc frame
        _CreateFrame RGBGS0_,0,96,r13,r14,r15
        _SaveXmmRegs xmm10,xmm11,xmm12,xmm13,xmm14,xmm15
        _EndProlog

        xor eax,eax                     ;error return code (also pixel_buffer offset)
        cmp r8d,[c_NumPixelsMin]
        jl Done                         ;jump if num_pixels < min value
        cmp r8d,[c_NumPixelsMax]
        jg Done                         ;jump if num_pixels > max value
        test r8d,3fh
        jnz Done                        ;jump if (num_pixels % 64) != 0

        test rcx,3fh
        jnz Done                        ;jump if pb_gs is not aligned

        mov r13,[rdx]
        test r13,3fh
        jnz Done                        ;jump if pb_r is not aligned
        mov r14,[rdx+8]
        test r14,3fh
        jnz Done                        ;jump if pb_g is not aligned
        mov r15,[rdx+16]
        test r15,3fh
        jnz Done                        ;jump if pb_b is not aligned

; Perform required initializations
        vbroadcastss zmm10,real4 ptr [r9]   ;zmm10 = packed coef[0]
        vbroadcastss zmm11,real4 ptr [r9+4] ;zmm11 = packed coef[1]
        vbroadcastss zmm12,real4 ptr [r9+8] ;zmm12 = packed coef[2]
        vbroadcastss zmm13,real4 ptr [r4_0p5]   ;zmm13 = packed 0.5
        vbroadcastss zmm14,real4 ptr [r4_255p0] ;zmm14 = packed 255.0
        vxorps zmm15,zmm15,zmm15            ;zmm15 = packed 0.0
        mov r8d,r8d                         ;r8 = num_pixels
        mov r10,16                          ;r10 - number of pixels / iteration

; Load next block of pixels
        align 16
@@:     vpmovzxbd zmm0,xmmword ptr [r13+rax]    ;zmm0 = 16 pixels (r values)
        vpmovzxbd zmm1,xmmword ptr [r14+rax]    ;zmm1 = 16 pixels (g values)
        vpmovzxbd zmm2,xmmword ptr [r15+rax]    ;zmm2 = 16 pixels (b values)
; Convert dword values to SPFP and multiply by coefficients
        vcvtdq2ps zmm0,zmm0             ;zmm0 = 16 pixels SPFP (r values)
        vcvtdq2ps zmm1,zmm1             ;zmm1 = 16 pixels SPFP (g values)
        vcvtdq2ps zmm2,zmm2             ;zmm2 = 16 pixels SPFP (b values)
        vmulps zmm0,zmm0,zmm10          ;zmm0 = r values * coef[0]
        vmulps zmm1,zmm1,zmm11          ;zmm1 = g values * coef[1]
        vmulps zmm2,zmm2,zmm12          ;zmm2 = b values * coef[2]

; Sum color components & clip values to [0.0, 255.0]
        vaddps zmm3,zmm0,zmm1           ;r + g
        vaddps zmm4,zmm3,zmm2           ;r + g + b
        vaddps zmm5,zmm4,zmm13          ;r + g + b + 0.5
        vminps zmm0,zmm5,zmm14          ;clip pixels above 255.0
        vmaxps zmm1,zmm0,zmm15          ;clip pixels below 0.0

; Convert grayscale values from SPFP to byte, save results
        vcvtps2dq zmm2,zmm1             ;convert SPFP values to dwords

        vpmovusdb xmm3,zmm2             ;convert to bytes
        vmovdqa xmmword ptr [rcx+rax],xmm3  ;save grayscale image pixels

        add rax,r10
        sub r8,r10
        jnz @B

        mov eax,1                       ;set success return code
Done:   vzeroupper
        _RestoreXmmRegs xmm10,xmm11,xmm12,xmm13,xmm14,xmm15
        _DeleteFrame r13,r14,r15
        ret
Avx512RgbToGs_ endp

; extern "C" bool Avx2RgbToGs_(uint8_t* pb_gs, const uint8_t* const* pb_rgb, int num_pixels,
const float coef[3]);

        .code
Avx2RgbToGs_ proc frame
        _CreateFrame RGBGS1_,0,96,r13,r14,r15
        _SaveXmmRegs xmm10,xmm11,xmm12,xmm13,xmm14,xmm15
        _EndProlog

        xor eax,eax                     ;error return code (also pixel_buffer offset)
        cmp r8d,[c_NumPixelsMin]
        jl Done                         ;jump if num_pixels < min value
        cmp r8d,[c_NumPixelsMax]
        jg Done                         ;jump if num_pixels > max value
        test r8d,3fh
        jnz Done                        ;jump if (num_pixels % 64) != 0

        test rcx,3fh
        jnz Done                        ;jump if pb_gs is not aligned

        mov r13,[rdx]
        test r13,3fh
        jnz Done                        ;jump if pb_r is not aligned
        mov r14,[rdx+8]
        test r14,3fh
        jnz Done                        ;jump if pb_g is not aligned
        mov r15,[rdx+16]
        test r15,3fh
        jnz Done                        ;jump if pb_b is not aligned

; Perform required initializations
        vbroadcastss ymm10,real4 ptr [r9]   ;ymm10 = packed coef[0]
        vbroadcastss ymm11,real4 ptr [r9+4] ;ymm11 = packed coef[1]
        vbroadcastss ymm12,real4 ptr [r9+8] ;ymm12 = packed coef[2]
        vbroadcastss ymm13,real4 ptr [r4_0p5]   ;ymm13 = packed 0.5
        vbroadcastss ymm14,real4 ptr [r4_255p0] ;ymm14 = packed 255.0
        vxorps ymm15,ymm15,ymm15            ;ymm15 = packed 0.0
        mov r8d,r8d                         ;r8 = num_pixels
        mov r10,8                           ;r10 - number of pixels / iteration

; Load next block of pixels
        align 16
@@:     vpmovzxbd ymm0,qword ptr [r13+rax]  ;ymm0 = 8 pixels (r values)
        vpmovzxbd ymm1,qword ptr [r14+rax]  ;ymm1 = 8 pixels (g values)
        vpmovzxbd ymm2,qword ptr [r15+rax]  ;ymm2 = 8 pixels (b values)
; Convert dword values to SPFP and multiply by coefficients
        vcvtdq2ps ymm0,ymm0             ;ymm0 = 8 pixels SPFP (r values)
        vcvtdq2ps ymm1,ymm1             ;ymm1 = 8 pixels SPFP (g values)
        vcvtdq2ps ymm2,ymm2             ;ymm2 = 8 pixels SPFP (b values)
        vmulps ymm0,ymm0,ymm10          ;ymm0 = r values * coef[0]
        vmulps ymm1,ymm1,ymm11          ;ymm1 = g values * coef[1]
        vmulps ymm2,ymm2,ymm12          ;ymm2 = b values * coef[2]

; Sum color components & clip values to [0.0, 255.0]
        vaddps ymm3,ymm0,ymm1           ;r + g
        vaddps ymm4,ymm3,ymm2           ;r + g + b
        vaddps ymm5,ymm4,ymm13          ;r + g + b + 0.5
        vminps ymm0,ymm5,ymm14          ;clip pixels above 255.0
        vmaxps ymm1,ymm0,ymm15          ;clip pixels below 0.0

; Convert grayscale components from SPFP to byte, save results
        vcvtps2dq ymm2,ymm1             ;convert SPFP values to dwords

        vpackusdw ymm3,ymm2,ymm2
        vextracti128 xmm4,ymm3,1
        vpackuswb xmm5,xmm3,xmm4        ;byte GS pixels in xmm5[31:0] and xmm5[95:64]
        vpextrd r11d,xmm5,0             ;r11d = 4 grayscale pixels
        mov dword ptr [rcx+rax],r11d    ;save grayscale image pixels
        vpextrd r11d,xmm5,2             ;r11d = 4 grayscale pixels
        mov dword ptr [rcx+rax+4],r11d  ;save grayscale image pixels

        add rax,r10
        sub r8,r10
        jnz @B

        mov eax,1                       ;set success return code
Done:   vzeroupper
        _RestoreXmmRegs xmm10,xmm11,xmm12,xmm13,xmm14,xmm15
        _DeleteFrame r13,r14,r15
        ret
Avx2RgbToGs_ endp
        end
;-------------------------------------------------
;####### Ch15_01.asm
;-------------------------------------------------
        .const
r8_2p0  real8 2.0

; extern "C" int CalcResult_(double* y, const double* x, size_t n);

        .code
CalcResult_ proc

; Forward conditional jumps are used in this code block since
; the fall-through cases are more likely to occur
        test r8,r8
        jz Done                         ;jump if n == 0

        test r8,7h
```

```
        jnz Error                       ;jump if (n % 8) != 0
        test rcx,1fh
        jnz Error                       ;jump if y is not aligned to a 32b boundary
        test rdx,1fh
        jnz Error                       ;jump if x is not aligned to a 32b boundary

; Initialize
        xor eax,eax                         ;set array offset to zero
        vbroadcastsd ymm5,real8 ptr [r8_2p0]    ;packed 2.0

; Simple array processing loop
        align 16
@@:     vmovapd ymm0,ymmword ptr [rdx+rax]  ;load x[i+3]:x[i]
        vdivpd ymm1,ymm0,ymm5
        vsqrtpd ymm2,ymm1
        vmovapd ymmword ptr [rcx+rax],ymm2  ;save y[i+3]:y[i]

        vmovapd ymm0,ymmword ptr [rdx+rax+32]   ;load x[i+7]:x[i+4]
        vdivpd ymm1,ymm0,ymm5
        vsqrtpd ymm2,ymm1
        vmovapd ymmword ptr [rcx+rax+32],ymm2   ;save y[i+7]:y[i+4]

; A backward conditional jump is used in this code block since
; the fall-through case is less likely to occur
        add rax,64
        sub r8,8
        jnz @B

Done:   xor eax,eax                     ;set success return code
        vzeroupper
        ret

; Error handling code that's unlikely to execute

Error:  mov eax,1                       ;set error return code
        ret
CalcResult_ endp
        end
;-------------------------------------------------
;####### Ch16_02.asm
;-------------------------------------------------
; _CalcResult Macro
; The following macro contains a simple calculating loop that is used
; to compare performance of the vmovaps and vmovntps instructions.

_CalcResult macro MovInstr

; Load and validate arguments
        xor eax,eax                     ;set error code
        test r9,r9
        jz Done                         ;jump if n <= 0
        test r9,0fh
        jnz Done                        ;jump if (n % 16) != 0

        test rcx,1fh
        jnz Done                        ;jump if c is not aligned
        test rdx,1fh
        jnz Done                        ;jump if a is not aligned
        test r8,1fh
        jnz Done                        ;jump if b is not aligned

; Calculate c[i] = sqrt(a[i] * a[i] + b[i] * b[i])
        align 16
@@:     vmovaps ymm0,ymmword ptr [rdx+rax]  ;ymm0 = a[i+7]:a[i]
        vmovaps ymm1,ymmword ptr [r8+rax]   ;ymm1 = b[i+7]:b[i]
        vmulps ymm2,ymm0,ymm0               ;ymm2 = a[i] * a[i]
        vmulps ymm3,ymm1,ymm1               ;ymm3 = b[i] * b[i]
        vaddps ymm4,ymm2,ymm3               ;ymm4 = sum
        vsqrtps ymm5,ymm4                   ;ymm5 = final result
        MovInstr ymmword ptr [rcx+rax],ymm5 ;save final values to c

        vmovaps ymm0,ymmword ptr [rdx+rax+32]   ;ymm0 = a[i+15]:a[i+8]
        vmovaps ymm1,ymmword ptr [r8+rax+32]    ;ymm1 = b[i+15]:b[i+8]
        vmulps ymm2,ymm0,ymm0               ;ymm2 = a[i] * a[i]
        vmulps ymm3,ymm1,ymm1               ;ymm3 = b[i] * b[i]
        vaddps ymm4,ymm2,ymm3               ;ymm4 = sum
        vsqrtps ymm5,ymm4                   ;ymm5 = final result
        MovInstr ymmword ptr [rcx+rax+32],ymm5  ;save final values to c

        add rax,64                          ;update offset
        sub r9,16                           ;update counter
        jnz @B

        mov eax,1                          ;set success return code

Done:   vzeroupper
        ret
        endm

; extern bool CalcResultA_(float* c, const float* a, const float* b, size_t n)
        .code
CalcResultA_ proc
        _CalcResult vmovaps
CalcResultA_ endp

; extern bool CalcResultB_(float* c, const float* a, const float* b, int n)
CalcResultB_ proc
        _CalcResult vmovntps
CalcResultB_ endp
        end
;-------------------------------------------------
;####### Ch16_03_.asm
        include <Ch16_03_.asmh>

; Macro _LlTraverse
; The following macro generates linked list traversal code using the
; prefetchnta instruction if UsePrefetch is equal to 'Y'.

_LlTraverse macro UsePrefetch
        mov rax,rcx                             ;rax = ptr to 1st node
        test rax,rax
        jz Done                                 ;jump if empty list

        align 16
@@::    mov rcx,[rax+LlNode.Link]               ;rcx = next node
        vmovapd ymm0,ymmword ptr [rax+LlNode.ValA]  ;ymm0 = ValA
        vmovapd ymm1,ymmword ptr [rax+LLNode.ValB]  ;ymm1 = ValB

IFIDNI <UsePrefetch>,<Y>
        mov rdx,rcx
        test rdx,rdx                        ;is there another node?
        cmovz rdx,rax                       ;avoid prefetch of nullptr
        prefetchnta [rdx]                   ;prefetch start of next node
ENDIF

; Calculate ValC[i] = sqrt(ValA[i] * ValA[i] + ValB[i] * ValB[i])
        vmulpd ymm2,ymm0,ymm0               ;ymm2 = ValA * ValA
        vmulpd ymm3,ymm1,ymm1               ;ymm3 = ValB * ValB
        vaddpd ymm4,ymm2,ymm3               ;ymm4 = sums
        vsqrtpd ymm5,ymm4                   ;ymm5 = square roots

        vmovntpd ymmword ptr [rax+LlNode.ValC],ymm5 ;save result

; Calculate ValD[i] = sqrt(ValA[i] / ValB[i] + ValB[i] / ValA[i]);
```

```
        vdivpd ymm2,ymm0,ymm1               ;ymm2 = ValA / ValB
        vdivpd ymm3,ymm1,ymm0               ;ymm3 = ValB / ValA
        vaddpd ymm4,ymm2,ymm3               ;ymm4 = sums
        vsqrtpd ymm5,ymm4                   ;ymm5 = square roots

        vmovntpd ymmword ptr [rax+LlNode.ValD],ymm5 ;save result

        mov rax,rcx                         ;rax = ptr to next node
        test rax,rax
        jnz @B

Done:   vzeroupper
        ret
        endm

; extern "C" void LlTraverseA_(LlNode* first);

        .code
LlTraverseA_ proc
        _LlTraverse n
LlTraverseA_ endp

; extern "C" void LlTraverseB_(LlNode* first);

LlTraverseB_ proc
        _LlTraverse y
LlTraverseB_ endp
        end;-------------------------------------------------
;####### Ch16_04_.asm
        include <MacrosX86-64-AVX.asmh>

CalcInfo struct
    X1 qword ?
    X2 qword ?
    Y1 qword ?
    Y2 qword ?
    Z1 qword ?
    Z2 qword ?
    Result qword ?
    Index0 qword ?
    Index1 qword ?
    Status dword ?
CalcInfo ends

        .const
r8_1p0  real8 1.0

; extern "C" void CalcResult_(CalcInfo* ci)
        .code
CalcResult_ proc frame
        _CreateFrame CR,0,16,r12,r13,r14,r15
        _SaveXmmRegs xmm6
        _EndProlog

        mov dword ptr [rcx+CalcInfo.Status],0

; Make sure num_elements is valid
        mov rax,[rcx+CalcInfo.Index0]   ;rax = start index
        mov rdx,[rcx+CalcInfo.Index1]   ;rdx = stop index
        sub rdx,rax
        add rdx,1                       ;rdx = num_elements
        test rdx,rdx
        jz Done                         ;jump if num_elements == 0
        test rdx,7
        jnz Done                        ;jump if num_elements % 8 != 0

; Make sure all arrays are properly aligned
```

```
        mov r8d,1fh
        mov r9,[rcx+CalcInfo.Result]
        test r9,r8
        jnz Done

        mov r10,[rcx+CalcInfo.X1]
        test r10,r8
        jnz Done
        mov r11,[rcx+CalcInfo.X2]
        test r11,r8
        jnz Done

        mov r12,[rcx+CalcInfo.Y1]
        test r12,r8
        jnz Done
        mov r13,[rcx+CalcInfo.Y2]
        test r13,r8
        jnz Done

        mov r14,[rcx+CalcInfo.Z1]
        test r14,r8
        jnz Done
        mov r15,[rcx+CalcInfo.Z2]
        test r15,r8
        jnz Done

        vbroadcastsd ymm6,real8 ptr [r8_1p0]    ;ymm6 = packed 1.0 (DPFP)
; Perform simulated calculation
        align 16
LP1:    vmovapd ymm0,ymmword ptr [r10+rax*8]
        vmovapd ymm1,ymmword ptr [r12+rax*8]
        vmovapd ymm2,ymmword ptr [r14+rax*8]
        vsubpd ymm0,ymm0,ymmword ptr [r11+rax*8]
        vsubpd ymm1,ymm1,ymmword ptr [r13+rax*8]
        vsubpd ymm2,ymm2,ymmword ptr [r15+rax*8]
        vmulpd ymm3,ymm0,ymm0
        vmulpd ymm4,ymm1,ymm1
        vmulpd ymm5,ymm2,ymm2
        vaddpd ymm0,ymm3,ymm4
        vaddpd ymm1,ymm0,ymm5
        vsqrtpd ymm2,ymm1
        vdivpd ymm3,ymm6,ymm2
        vsqrtpd ymm4,ymm3
        vmovntpd ymmword ptr [r9+rax*8],ymm4

        add rax,4

        vmovapd ymm0,ymmword ptr [r10+rax*8]
        vmovapd ymm1,ymmword ptr [r12+rax*8]
        vmovapd ymm2,ymmword ptr [r14+rax*8]
        vsubpd ymm0,ymm0,ymmword ptr [r11+rax*8]
        vsubpd ymm1,ymm1,ymmword ptr [r13+rax*8]
        vsubpd ymm2,ymm2,ymmword ptr [r15+rax*8]
        vmulpd ymm3,ymm0,ymm0
        vmulpd ymm4,ymm1,ymm1
        vmulpd ymm5,ymm2,ymm2
        vaddpd ymm0,ymm3,ymm4
        vaddpd ymm1,ymm0,ymm5
        vsqrtpd ymm2,ymm1
        vdivpd ymm3,ymm6,ymm2
        vsqrtpd ymm4,ymm3
        vmovntpd ymmword ptr [r9+rax*8],ymm4

        add rax,4
        sub rdx,8
        jnz LP1
```

```
        mov dword ptr [rcx+CalcInfo.Status],1

Done:   vzeroupper
        _RestoreXmmRegs xmm6
        _DeleteFrame r12,r13,r14,r15
        ret
CalcResult_ endp
        end
```