Hardware/ Software Interface

Compilers frequently create branches and labels where they do not appear in the programming language. Avoiding the burden of writing explicit labels and branches is one benefit of writing in high-level programming languages and is a reason coding is faster at that level.

Loops

Decisions are important both for choosing between two alternatives—found in *if* statements—and for iterating a computation—found in loops. The same assembly instructions are the building blocks for both cases.

EXAMPLE

Compiling a *while* Loop in C

Here is a traditional loop in C:

```
while (save[i] == k)
      i += 1;
```

Assume that `i` and `k` correspond to registers `$s3` and `$s5` and the base of the array `save` is in `$s6`. What is the MIPS assembly code corresponding to this C segment?

ANSWER

The first step is to load `save[i]` into a temporary register. Before we can load `save[i]` into a temporary register, we need to have its address. Before we can add `i` to the base of array `save` to form the address, we must multiply the index `i` by 4 due to the byte addressing problem. Fortunately, we can use shift left logical, since shifting left by 2 bits multiplies by 2^2 or 4 (see page 88 in the prior section). We need to add the label `Loop` to it so that we can branch back to that instruction at the end of the loop:

```
Loop: sll  $t1,$s3,2    # Temp reg $t1 = i * 4
```

To get the address of `save[i]`, we need to add `$t1` and the base of save in `$s6`:

```
      add $t1,$t1,$s6     # $t1 = address of save[i]
```

Now we can use that address to load `save[i]` into a temporary register:

```
      lw $t0,0($t1)       # Temp reg $t0 = save[i]
```

The next instruction performs the loop test, exiting if `save[i] ≠ k`:

```
      bne $t0,$s5, Exit   # go to Exit if save[i] ≠ k
```

The next instruction adds 1 to `i`:

```
        addi $s3,$s3,1        # i = i + 1
```

The end of the loop branches back to the *while* test at the top of the loop. We just add the `Exit` label after it, and we're done:

```
        j     Loop            # go to Loop
     Exit:
```

(See the exercises for an optimization of this sequence.)

Hardware/Software Interface

Such sequences of instructions that end in a branch are so fundamental to compiling that they are given their own buzzword: a **basic block** is a sequence of instructions without branches, except possibly at the end, and without branch targets or branch labels, except possibly at the beginning. One of the first early phases of compilation is breaking the program into basic blocks.

basic block A sequence of instructions without branches (except possibly at the end) and without branch targets or branch labels (except possibly at the beginning).

The test for equality or inequality is probably the most popular test, but sometimes it is useful to see if a variable is less than another variable. For example, a *for* loop may want to test to see if the index variable is less than 0. Such comparisons are accomplished in MIPS assembly language with an instruction that compares two registers and sets a third register to 1 if the first is less than the second; otherwise, it is set to 0. The MIPS instruction is called *set on less than,* or `slt`. For example,

```
    slt    $t0, $s3, $s4   # $t0 = 1 if $s3 < $s4
```

means that register `$t0` is set to 1 if the value in register `$s3` is less than the value in register `$s4`; otherwise, register `$t0` is set to 0.

Constant operands are popular in comparisons, so there is an immediate version of the set on less than instruction. To test if register `$s2` is less than the constant 10, we can just write

```
    slti    $t0,$s2,10     # $t0 = 1 if $s2 < 10
```

Hardware/Software Interface

MIPS compilers use the `slt`, `slti`, `beq`, `bne`, and the fixed value of 0 (always available by reading register `$zero`) to create all relative conditions: equal, not equal, less than, less than or equal, greater than, greater than or equal.

Heeding von Neumann's warning about the simplicity of the "equipment," the MIPS architecture doesn't include branch on less than because it is too complicated; either it would stretch the clock cycle time or it would take extra clock cycles per instruction. Two faster instructions are more useful.

Hardware/ Software Interface

Comparison instructions must deal with the dichotomy between signed and unsigned numbers. Sometimes a bit pattern with a 1 in the most significant bit represents a negative number and, of course, is less than any positive number, which must have a 0 in the most significant bit. With unsigned integers, on the other hand, a 1 in the most significant bit represents a number that is *larger* than any that begins with a 0. (We'll soon take advantage of this dual meaning of the most significant bit to reduce the cost of the array bounds checking.)

MIPS offers two versions of the set on less than comparison to handle these alternatives. *Set on less than* (`slt`) and *set on less than immediate* (`slti`) work with signed integers. Unsigned integers are compared using *set on less than unsigned* (`sltu`) and *set on less than immediate unsigned* (`sltiu`).

EXAMPLE

Signed versus Unsigned Comparison

Suppose register `$s0` has the binary number

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two}$

and that register `$s1` has the binary number

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two}$

What are the values of registers `$t0` and `$t1` after these two instructions?

```
slt      $t0, $s0, $s1 # signed comparison
sltu     $t1, $s0, $s1 # unsigned comparison
```

ANSWER

The value in register `$s0` represents -1_{ten} if it is an integer and $4{,}294{,}967{,}295_{ten}$ if it is an unsigned integer. The value in register `$s1` represents 1_{ten} in either case. Then register `$t0` has the value 1, since $-1_{ten} < 1_{ten}$, and register `$t1` has the value 0, since $4{,}294{,}967{,}295_{ten} > 1_{ten}$.

Treating signed numbers as if they were unsigned gives us a low cost way of checking if $0 \leq x < y$, which matches the index out-of-bounds check for arrays. The key is that negative integers in two's complement notation look like large numbers in unsigned notation; that is, the most significant bit is a sign bit in the former notation but a large part of the number in the latter. Thus, an unsigned comparison of $x < y$ also checks if x is negative as well as if x is less than y.

Bounds Check Shortcut

EXAMPLE

Use this shortcut to reduce an index-out-of-bounds check: jump to `IndexOutOfBounds` if `$s1 ≥ $t2` or if `$s1` is negative.

ANSWER

The checking code just uses `u` to do both checks:

```
sltu $t0,$s1,$t2 # $t0=0 if $s1>=length or $s1<0
beq  $t0,$zero,IndexOutOfBounds #if bad, goto Error
```

Case/Switch Statement

Most programming languages have a *case* or *switch* statement that allows the programmer to select one of many alternatives depending on a single value. The simplest way to implement *switch* is via a sequence of conditional tests, turning the *switch* statement into a chain of *if-then-else* statements.

Sometimes the alternatives may be more efficiently encoded as a table of addresses of alternative instruction sequences, called a **jump address table** or **jump table**, and the program needs only to index into the table and then jump to the appropriate sequence. The jump table is then just an array of words containing addresses that correspond to labels in the code. The program loads the appropriate entry from the jump table into a register. It then needs to jump using the address in the register. To support such situations, computers like MIPS include a *jump register* instruction (`jr`), meaning an unconditional jump to the address specified in a register. Then it jumps to the proper address using this instruction. We'll see an even more popular use of `jr` in the next section.

jump address table Also called **jump table**. A table of addresses of alternative instruction sequences.

(Many compilers would divide 5.0 by 9.0 at compile time and save the single constant 5.0/9.0 in memory, thereby avoiding the divide at runtime.) Next, we load the constant 32.0 and then subtract it from `fahr` (`$f12`):

```
lwc1 $f18, const32($gp)# $f18 = 32.0
sub.s $f18, $f12, $f18 # $f18 = fahr - 32.0
```

Finally, we multiply the two intermediate results, placing the product in `$f0` as the return result, and then return

```
mul.s $f0, $f16, $f18 # $f0 = (5/9)*(fahr - 32.0)
jr $ra                # return
```

Now let's perform floating-point operations on matrices, code commonly found in scientific programs.

EXAMPLE

Compiling Floating-Point C Procedure with Two-Dimensional Matrices into MIPS

Most floating-point calculations are performed in double precision. Let's perform matrix multiply of C = C + A * B. It is commonly called DGEMM, for Double precision, General Matrix Multiply. We'll see versions of DGEMM again in Section 3.8 and subsequently in Chapters 4, 5, and 6. Let's assume C, A, and B are all square matrices with 32 elements in each dimension.

```
void mm (double c[][], double a[][], double b[][])
{
        int i, j, k;
        for (i = 0; i != 32; i = i + 1)
        for (j = 0; j != 32; j = j + 1)
        for (k = 0; k != 32; k = k + 1)
          c[i][j] = c[i][j] + a[i][k] *b[k][j];
}
```

The array starting addresses are parameters, so they are in `$a0`, `$a1`, and `$a2`. Assume that the integer variables are in `$s0`, `$s1`, and `$s2`, respectively. What is the MIPS assembly code for the body of the procedure?

ANSWER

Note that `c[i][j]` is used in the innermost loop above. Since the loop index is `k`, the index does not affect `c[i][j]`, so we can avoid loading and storing `c[i][j]` each iteration. Instead, the compiler loads `c[i][j]` into a register outside the loop, accumulates the sum of the products of `a[i][k]` and b[k][j] in that same register, and then stores the sum into c[i][j] upon termination of the innermost loop.

We keep the code simpler by using the assembly language pseudoinstructions li (which loads a constant into a register), and l.d and s.d (which the assembler turns into a pair of data transfer instructions, lwc1 or swc1, to a pair of floating-point registers).

The body of the procedure starts with saving the loop termination value of 32 in a temporary register and then initializing the three *for* loop variables:

```
    mm:...
        li     $t1, 32  # $t1 = 32 (row size/loop end)
        li     $s0, 0   # i = 0; initialize 1st for loop
L1:     li     $s1, 0   # j = 0; restart 2nd for loop
L2:     li     $s2, 0   # k = 0; restart 3rd for loop
```

To calculate the address of c[i][j], we need to know how a 32 × 32, two-dimensional array is stored in memory. As you might expect, its layout is the same as if there were 32 single-dimension arrays, each with 32 elements. So the first step is to skip over the i "single-dimensional arrays," or rows, to get the one we want. Thus, we multiply the index in the first dimension by the size of the row, 32. Since 32 is a power of 2, we can use a shift instead:

```
sll  $t2, $s0, 5      # $t2 = i * 2^5 (size of row of c)
```

Now we add the second index to select the jth element of the desired row:

```
    addu  $t2, $t2, $s1   # $t2 = i * size(row) + j
```

To turn this sum into a byte index, we multiply it by the size of a matrix element in bytes. Since each element is 8 bytes for double precision, we can instead shift left by 3:

```
    sll  $t2, $t2, 3      # $t2 = byte offset of [i][j]
```

Next we add this sum to the base address of c, giving the address of c[i][j], and then load the double precision number c[i][j] into $f4:

```
addu  $t2, $a0, $t2   # $t2 = byte address of c[i][j]
l.d   $f4, 0($t2)     # $f4 = 8 bytes of c[i][j]
```

The following five instructions are virtually identical to the last five: calculate the address and then load the double precision number b[k][j].

```
L3: sll $t0, $s2, 5     # $t0 = k * 2^5 (size of row of b)
    addu $t0, $t0, $s1 # $t0 = k * size(row) + j
    sll $t0, $t0, 3    # $t0 = byte offset of [k][j]
    addu $t0, $a2, $t0 # $t0 = byte address of b[k][j]
    l.d $f16, 0($t0)   # $f16 = 8 bytes of b[k][j]
```

Similarly, the next five instructions are like the last five: calculate the address and then load the double precision number a[i][k].

```
sll     $t0, $s0, 5     # $t0 = i * 2^5 (size of row of a)
addu    $t0, $t0, $s2   # $t0 = i * size(row) + k
sll     $t0, $t0, 3     # $t0 = byte offset of [i][k]
addu    $t0, $a1, $t0   # $t0 = byte address of a[i][k]
l.d     $f18, 0($t0)    # $f18 = 8 bytes of a[i][k]
```

Now that we have loaded all the data, we are finally ready to do some floating-point operations! We multiply elements of `a` and `b` located in registers `$f18` and `$f16`, and then accumulate the sum in `$f4`.

```
mul.d $f16, $f18, $f16 # $f16 = a[i][k] * b[k][j]
add.d $f4, $f4, $f16   # f4 = c[i][j] + a[i][k] * b[k][j]
```

The final block increments the index `k` and loops back if the index is not 32. If it is 32, and thus the end of the innermost loop, we need to store the sum accumulated in `$f4` into `c[i][j]`.

```
addiu  $s2, $s2, 1      # $k = k + 1
bne    $s2, $t1, L3     # if (k != 32) go to L3
s.d    $f4, 0($t2)      # c[i][j] = $f4
```

Similarly, these final four instructions increment the index variable of the middle and outermost loops, looping back if the index is not 32 and exiting if the index is 32.

```
addiu  $s1, $s1, 1      # $j = j + 1
bne    $s1, $t1, L2     # if (j != 32) go to L2
addiu  $s0, $s0, 1      # $i = i + 1
bne    $s0, $t1, L1     # if (i != 32) go to L1
…
```

Figure 3.22 below shows the x86 assembly language code for a slightly different version of DGEMM in Figure 3.21.

Elaboration: The array layout discussed in the example, called *row-major order,* is used by C and many other programming languages. Fortran instead uses *column-major order,* whereby the array is stored column by column.

Elaboration: Only 16 of the 32 MIPS floating-point registers could originally be used for double precision operations: `$f0`, `$f2`, `$f4`, …, `$f30`. Double precision is computed using pairs of these single precision registers. The odd-numbered floating-point registers were used only to load and store the right half of 64-bit floating-point numbers. MIPS-32 added `l.d` and `s.d` to the instruction set. MIPS-32 also added "paired single" versions of all floating-point instructions, where a single instruction results in two parallel floating-point operations on two 32-bit operands inside 64-bit registers (see Section 3.6). For example, `add.ps $f0, $f2, $f4` is equivalent to `add.s $f0, $f2, $f4` followed by `add.s $f1, $f3, $f5`.

In an attempt to squeeze every last bit of precision from a floating-point operation, the standard allows some numbers to be represented in unnormalized form. Rather than having a gap between 0 and the smallest normalized number, IEEE allows *denormalized numbers* (also known as *denorms* or *subnormals*). They have the same exponent as zero but a nonzero fraction. They allow a number to degrade in significance until it becomes 0, called *gradual underflow*. For example, the smallest positive single precision normalized number is

$$1.0000\ 0000\ 0000\ 0000\ 0000\ 000_{\text{two}} \times 2^{-126}$$

but the smallest single precision denormalized number is

$$0.0000\ 0000\ 0000\ 0000\ 0000\ 001_{\text{two}} \times 2^{-126}, \textit{ or } 1.0_{\text{two}} \times 2^{-149}$$

For double precision, the denorm gap goes from 1.0×2^{-1022} to 1.0×2^{-1074}.

The possibility of an occasional unnormalized operand has given headaches to floating-point designers who are trying to build fast floating-point units. Hence, many computers cause an exception if an operand is denormalized, letting software complete the operation. Although software implementations are perfectly valid, their lower performance has lessened the popularity of denorms in portable floating-point software. Moreover, if programmers do not expect denorms, their programs may surprise them.

3.6 Parallelism and Computer Arithmetic: Subword Parallelism

Since every desktop microprocessor by definition has its own graphical displays, as transistor budgets increased it was inevitable that support would be added for graphics operations.

Many graphics systems originally used 8 bits to represent each of the three primary colors plus 8 bits for a location of a pixel. The addition of speakers and microphones for teleconferencing and video games suggested support of sound as well. Audio samples need more than 8 bits of precision, but 16 bits are sufficient.

Every microprocessor has special support so that bytes and halfwords take up less space when stored in memory (see Section 2.9), but due to the infrequency of arithmetic operations on these data sizes in typical integer programs, there was little support beyond data transfers. Architects recognized that many graphics and audio applications would perform the same operation on vectors of this data. By partitioning the carry chains within a 128-bit adder, a processor could use **parallelism** to perform simultaneous operations on short vectors of sixteen 8-bit operands, eight 16-bit operands, four 32-bit operands, or two 64-bit operands. The cost of such partitioned adders was small.

PARALLELISM

Given that the parallelism occurs within a wide word, the extensions are classified as *subword parallelism*. It is also classified under the more general name of *data level parallelism*. They have been also called vector or SIMD, for single instruction, multiple data (see Section 6.6). The rising popularity of multimedia applications led to arithmetic instructions that support narrower operations that can easily operate in parallel.

For example, ARM added more than 100 instructions in the NEON multimedia instruction extension to support subword parallelism, which can be used either with ARMv7 or ARMv8. It added 256 bytes of new registers for NEON that can be viewed as 32 registers 8 bytes wide or 16 registers 16 bytes wide. NEON supports all the subword data types you can imagine *except* 64-bit floating point numbers:

- 8-bit, 16-bit, 32-bit, and 64-bit signed and unsigned integers
- 32-bit floating point numbers

Figure 3.19 gives a summary of the basic NEON instructions.

Data transfer	Arithmetic	Logical/Compare
VLDR.F32	VADD.F32, VADD{L,W}{S8,U8,S16,U16,S32,U32}	VAND.64, VAND.128
VSTR.F32	VSUB.F32, VSUB{L,W}{S8,U8,S16,U16,S32,U32}	VORR.64, VORR.128
VLD{1,2,3.4}.{I8,I16,I32}	VMUL.F32, VMULL{S8,U8,S16,U16,S32,U32}	VEOR.64, VEOR.128
VST{1,2,3.4}.{I8,I16,I32}	VMLA.F32, VMLAL{S8,U8,S16,U16,S32,U32}	VBIC.64, VBIC.128
VMOV.{I8,I16,I32,F32}, #imm	VMLS.F32, VMLSL{S8,U8,S16,U16,S32,U32}	VORN.64, VORN.128
VMVN.{I8,I16,I32,F32}, #imm	VMAX.{S8,U8,S16,U16,S32,U32,F32}	VCEQ.{I8,I16,I32,F32}
VMOV.{I64,I128}	VMIN.{S8,U8,S16,U16,S32,U32,F32}	VCGE.{S8,U8,S16,U16,S32,U32,F32}
VMVN.{I64,I128}	VABS.{S8,S16,S32,F32}	VCGT.{S8,U8,S16,U16,S32,U32,F32}
	VNEG.{S8,S16,S32,F32}	VCLE.{S8,U8,S16,U16,S32,U32,F32}
	VSHL.{S8,U8,S16,U16,S32,S64,U64}	VCLT.{S8,U8,S16,U16,S32,U32,F32}
	VSHR.{S8,U8,S16,U16,S32,S64,U64}	VTST.{I8,I16,I32}

FIGURE 3.19 Summary of ARM NEON instructions for subword parallelism. We use the curly brackets {} to show optional variations of the basic operations: {S8,U8,8} stand for signed and unsigned 8-bit integers or 8-bit data where type doesn't matter, of which 16 fit in a 128-bit register; {S16,U16,16} stand for signed and unsigned 16-bit integers or 16-bit type-less data, of which 8 fit in a 128-bit register; {S32,U32,32} stand for signed and unsigned 32-bit integers or 32-bit type-less data, of which 4 fit in a 128-bit register; {S64,U64,64} stand for signed and unsigned 64-bit integers or type-less 64-bit data, of which 2 fit in a 128-bit register; {F32} stand for signed and unsigned 32-bit floating point numbers, of which 4 fit in a 128-bit register. Vector Load reads one n-element structure from memory into 1, 2, 3, or 4 NEON registers. It loads a single n-element structure to one lane (See Section 6.6), and elements of the register that are not loaded are unchanged. Vector Store writes one n-element structure into memory from 1, 2, 3, or 4 NEON registers.

Elaboration: In addition to signed and unsigned integers, ARM includes "fixed-point" format of four sizes called I8, I16, I32, and I64, of which 16, 8, 4, and 2 fit in a 128-bit register, respectively. A portion of the fixed point is for the fraction (to the right of the binary point) and the rest of the data is the integer portion (to the left of the binary point). The location of the binary point is up to the software. Many ARM processors do not have floating point hardware and thus floating point operations must be performed by library routines. Fixed point arithmetic can be significantly faster than software floating point routines, but more work for the programmer.

3.7 Real Stuff: Streaming SIMD Extensions and Advanced Vector Extensions in x86

The original MMX (*MultiMedia eXtension*) and SSE (*Streaming SIMD Extension*) instructions for the x86 included similar operations to those found in ARM NEON. Chapter 2 notes that in 2001 Intel added 144 instructions to its architecture as part of SSE2, including double precision floating-point registers and operations. It includes eight 64-bit registers that can be used for floating-point operands. AMD expanded the number to 16 registers, called XMM, as part of AMD64, which Intel relabeled EM64T for its use. Figure 3.20 summarizes the SSE and SSE2 instructions.

In addition to holding a single precision or double precision number in a register, Intel allows multiple floating-point operands to be packed into a single 128-bit SSE2 register: four single precision or two double precision. Thus, the 16 floating-point registers for SSE2 are actually 128 bits wide. If the operands can be arranged in memory as 128-bit aligned data, then 128-bit data transfers can load and store multiple operands per instruction. This packed floating-point format is supported by arithmetic operations that can operate simultaneously on four singles (PS) or two doubles (PD).

Data transfer	Arithmetic	Compare
MOV{A/U}{SS/PS/SD/PD} xmm, mem/xmm	ADD{SS/PS/SD/PD} xmm,mem/xmm	CMP{SS/PS/SD/PD}
	SUB{SS/PS/SD/PD} xmm,mem/xmm	
MOV {H/L} {PS/PD} xmm, mem/xmm	MUL{SS/PS/SD/PD} xmm,mem/xmm	
	DIV{SS/PS/SD/PD} xmm,mem/xmm	
	SQRT{SS/PS/SD/PD} mem/xmm	
	MAX {SS/PS/SD/PD} mem/xmm	
	MIN{SS/PS/SD/PD} mem/xmm	

FIGURE 3.20 The SSE/SSE2 floating-point instructions of the x86. xmm means one operand is a 128-bit SSE2 register, and mem/xmm means the other operand is either in memory or it is an SSE2 register. We use the curly brackets {} to show optional variations of the basic operations: {SS} stands for *Scalar Single* precision floating point, or one 32-bit operand in a 128-bit register; {PS} stands for *Packed Single* precision floating point, or four 32-bit operands in a 128-bit register; {SD} stands for Scalar Double precision floating point, or one 64-bit operand in a 128-bit register; {PD} stands for *Packed Double* precision floating point, or two 64-bit operands in a 128-bit register; {A} means the 128-bit operand is aligned in memory; {U} means the 128-bit operand is unaligned in memory; {H} means move the high half of the 128-bit operand; and {L} means move the low half of the 128-bit operand.

In 2011 Intel doubled the width of the registers again, now called YMM, with *Advanced Vector Extensions (AVX)*. Thus, a single operation can now specify eight 32-bit floating-point operations or four 64-bit floating-point operations. The legacy SSE and SSE2 instructions now operate on the lower 128 bits of the YMM registers. Thus, to go from 128-bit and 256-bit operations, you prepend the letter "v" (for vector) in front of the SSE2 assembly language operations and then use the YMM register names instead of the XMM register name. For example, the SSE2 instruction to perform two 64-bit floating-point multiplies

```
addpd  %xmm0, %xmm4
```

It becomes

```
vaddpd  %ymm0, %ymm4
```

which now produces four 64-bit floating-point multiplies.

Elaboration: AVX also added three address instructions to x86. For example, `vaddpd` can now specify

```
vaddpd %ymm0, %ymm1, %ymm4 # %ymm4 = %ymm1 + %ymm2
```

instead of the standard two address version

```
addpd  %xmm0, %xmm4 # %xmm4 = %xmm4 + %xmm0
```

(Unlike MIPS, the destination is on the right in x86.) Three addresses can reduce the number of registers and instructions needed for a computation.

3.8 Going Faster: Subword Parallelism and Matrix Multiply

To demonstrate the performance impact of subword parallelism, we'll run the same code on the Intel Core i7 first without AVX and then with it. Figure 3.21 shows an unoptimized version of a matrix-matrix multiply written in C. As we saw in Section 3.5, this program is commonly called *DGEMM*, which stands for Double precision GEneral Matrix Multiply. Starting with this edition, we have added a new section entitled "Going Faster" to demonstrate the performance benefit of adapting software to the underlying hardware, in this case the Sandy Bridge version of the Intel Core i7 microprocessor. This new section in Chapters 3, 4, 5, and 6 will incrementally improve DGEMM performance using the ideas that each chapter introduces.

Figure 3.22 shows the x86 assembly language output for the inner loop of Figure 3.21. The five floating point-instructions start with a `v` like the AVX instructions, but note that they use the XMM registers instead of YMM, and they include `sd` in the name, which stands for scalar double precision. We'll define the subword parallel instructions shortly.

```
void dgemm (int n, double* A, double* B, double* C)
{
   for (int i = 0; i < n; ++i)
     for (int j = 0; j < n; ++j)
     {
       double cij = C[i+j*n]; /* cij = C[i][j] */
       for( int k = 0; k < n; k++ )
         cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
       C[i+j*n] = cij; /* C[i][j] = cij */
     }
}
```

FIGURE 3.21 Unoptimized C version of a double precision matrix multiply, widely known as DGEMM for Double-precision GEneral Matrix Multiply (GEMM). Because we are passing the matrix dimension as the parameter n, this version of DGEMM uses single dimensional versions of matrices C, A, and B and address arithmetic to get better performance instead of using the more intuitive two-dimensional arrays that we saw in Section 3.5. The comments remind us of this more intuitive notation.

```
vmovsd (%r10),%xmm0           # Load 1 element of C into %xmm0
mov    %rsi,%rcx              # register %rcx = %rsi
xor    %eax,%eax              # register %eax = 0
vmovsd (%rcx),%xmm1           # Load 1 element of B into %xmm1
add    %r9,%rcx               # register %rcx = %rcx + %r9
vmulsd (%r8,%rax,8),%xmm1,%xmm1 # Multiply %xmm1, element of A
add    $0x1,%rax              # register %rax = %rax + 1
cmp    %eax,%edi              # compare %eax to %edi
vaddsd %xmm1,%xmm0,%xmm0      # Add %xmm1, %xmm0
jg     30 <dgemm+0x30>        # jump if %eax > %edi
add    $0x1,%r11d             # register %r11 = %r11 + 1
vmovsd %xmm0,(%r10)           # Store %xmm0 into C element
```

FIGURE 3.22 The x86 assembly language for the body of the nested loops generated by compiling the optimized C code in Figure 3.21. Although it is dealing with just 64-bits of data, the compiler uses the AVX version of the instructions instead of SSE2 presumably so that it can use three address per instruction instead of two (see the Elaboration in Section 3.7).

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

FIGURE 3.23 Optimized C version of DGEMM using C intrinsics to generate the AVX subword-parallel instructions for the x86. Figure 3.24 shows the assembly language produced by the compiler for the inner loop.

While compiler writers may eventually be able to routinely produce high-quality code that uses the AVX instructions of the x86, for now we must "cheat" by using C intrinsics that more or less tell the compiler exactly how to produce good code. Figure 3.23 shows the enhanced version of Figure 3.21 for which the Gnu C compiler produces AVX code. Figure 3.24 shows annotated x86 code that is the output of compiling using gcc with the –O3 level of optimization.

The declaration on line 6 of Figure 3.23 uses the __m256d data type, which tells the compiler the variable will hold 4 double-precision floating-point values. The intrinsic _mm256_load_pd() also on line 6 uses AVX instructions to load 4 double-precision floating-point numbers in parallel (_pd) from the matrix C into c0. The address calculation C+i+j*n on line 6 represents element C[i+j*n]. Symmetrically, the final step on line 11 uses the intrinsic _mm256_store_pd() to store 4 double-precision floating-point numbers from c0 into the matrix C. As we're going through 4 elements each iteration, the outer *for* loop on line 4 increments i by 4 instead of by 1 as on line 3 of Figure 3.21.

Inside the loops, on line 9 we first load 4 elements of A again using _mm256_load_pd(). To multiply these elements by one element of B, on line 10 we first use the intrinsic _mm256_broadcast_sd(), which makes 4 identical copies of the scalar double precision number—in this case an element of B—in one of the YMM registers. We then use _mm256_mul_pd() on line 9 to multiply the four double-precision results in parallel. Finally, _mm256_add_pd() on line 8 adds the 4 products to the 4 sums in c0.

Figure 3.24 shows resulting x86 code for the body of the inner loops produced by the compiler. You can see the five AVX instructions—they all start with v and

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

FIGURE 3.24 The x86 assembly language for the body of the nested loops generated by compiling the optimized C code in Figure 3.23. Note the similarities to Figure 3.22, with the primary difference being that the five floating-point operations are now using YMM registers and using the pd versions of the instructions for parallel double precision instead of the sd version for scalar double precision.

four of the five use `pd` for parallel double precision—that correspond to the C intrinsics mentioned above. The code is very similar to that in Figure 3.22 above: both use 12 instructions, the integer instructions are nearly identical (but different registers), and the floating-point instruction differences are generally just going from *scalar double* (`sd`) using XMM registers to *parallel double* (`pd`) with YMM registers. The one exception is line 4 of Figure 3.24. Every element of A must be multiplied by one element of B. One solution is to place four identical copies of the 64-bit B element side-by-side into the 256-bit YMM register, which is just what the instruction `vbroadcastsd` does.

For matrices of dimensions of 32 by 32, the unoptimized DGEMM in Figure 3.21 runs at 1.7 GigaFLOPS (FLoating point Operations Per Second) on one core of a 2.6 GHz Intel Core i7 (Sandy Bridge). The optimized code in Figure 3.23 performs at 6.4 GigaFLOPS. The AVX version is 3.85 times as fast, which is very close to the factor of 4.0 increase that you might hope for from performing 4 times as many operations at a time by using **subword parallelism**.

PARALLELISM

Elaboration: As mentioned in the Elaboration in Section 1.6, Intel offers Turbo mode that temporarily runs at a higher clock rate until the chip gets too hot. This Intel Core i7 (Sandy Bridge) can increase from 2.6 GHz to 3.3 GHz in Turbo mode. The results above are with Turbo mode turned off. If we turn it on, we improve all the results by the increase in the clock rate of 3.3/2.6 = 1.27 to 2.1 GFLOPS for unoptimized DGEMM and 8.1 GFLOPS with AVX. Turbo mode works particularly well when using only a single core of an eight-core chip, as in this case, as it lets that single core use much more than its fair share of power since the other cores are idle.