Performance Engineering

Apart from increasing the operating frequency, there are several ways how a manufacturer can improve the processor performance:

Registers More registers, vector registers.

Special Operations For frequent, time-consuming tasks (e.g. special "multimedia" operations).

Parallelism Duplication of operation units ("superscalar").

Pipelining Parts of the processor can work in parallel: While (say) a result is calculated, the decoding unit already works on the next command. Depending on the processor such a command pipeline may have several stages.

Good performance might depend on arranging processor commands in the right order.

Problem: Branches will empty the pipeline ("branch prediction").

Compiler Optimization

Typically, a (good) compiler will try to optimize the produced machine code.

On modern processors, often an optimizing compiler will produce better code than (most) human programmers working in machine code.

Sometimes optimizations trande space for runtime (and thus one might not want to do them always.)

Some optimization techniques that are frequently used are:

Constant expressions $5 + 3 * 7$ is replaced by 26.

Common subexpressions are computed once and stored

```
a=a+(b*c+2)*d+(b*c+2);
```

is replaced by:

```
temp=b*c+2;
a=a+temp*d+temp;
```

Unused/overwritten variables are eliminated

```
a=2*b+c;
a=3*c+d;
```

is replaced by:

```
a=3*c+d;
```

Loop optimization Loop independent commands are taken out of the loop

```
for (i=0;i<10000;i++) {
  a=b+c; z=z+z*a; }
```

is replaced by:

```
a=b+c;
for (i=0;i<10000;i++) { z=z+z*a; }
```

Loop unrolling Replace a loop by a series of commands

```
for (i=1;i<3;i++){ a=a*b+c; }
```

is replaced by:

```
a=a*b+c;
a=a*b+c;
a=a*b+c;
```

This only works for small multiples. Also: Cache issues!

Inline code Replace functions by explicit code

```
float myfunc(float a) { return 2*a+3; }
...
a=myfunc(b);
c=myfunc(2*d+1);
```

is replaced by:

```
a=2*b+3;
```

```
c=2*(2*d+1)+3;
```

Unswitching A condition is taken out of a loop.

```
for (i=1;i<=10000;i++) {
  if (a<2) { b=b*c; }
  else { b=b*b^2-c; }
}
```

is replaced by:

```
if (a<2) { for (i=1;i<=10000;i++) b=b*c; }
else { for (i=1;i<=10000;i++) b=b*b^2-c; }
```

Divide optimization Use shift commands

```
int a;
...
a=a*2;
```

is replaced by:

```
a=a<<2;
```

Indexing elimination Instead of repeated index access use auxilliary variable.

```
a[t]=0;
for (i=1;i<10000;i++) { a[t]=a[t]+i; }
```

is replaced by:

```
temp=0;
for (i=1;i<10000;i++) { temp=temp+i; }
a[t]=temp;
```

Register Use Frequently used variables are kept in a processor register if possible.

Enable Pipelining Rearrange commands (where rearrangement does not change the way the program works) so that they use different parts of the processor (and thus can be done in parallel):

```
a=a+2;
a=a*5;
b=b<<2;
```

is replaced by:

```
a=a+2;
b=b<<2;
a=a*5;
```

Some optimizations can change the control flow of a program. This is not always desirable:

- Optimization takes time.
- Debugging.
- Critical timing/Hardware access
- Memory tradeoff might not be desirable.

- Potential error source.

Thus optimization can be switched off or be turned on to higher levels.

Typically the `-O` or `-O`*nr* flag turns on optimization, but there are also many compiler-dependent flags.

Code optimization

While modern compilers can do many optimizations, it is often possible do get further improvements by deliberate code tuning.

Good books for this topic include:

- Steve McConnell: Code Complete
- J.L.Bentley: Writing efficient programs

Caveats

- A better algorithm is (almost) always preferrable to code tuning.
- Some optimizations can substantially reduce program readability or block further development or improvements.
- Optimize, what takes significant time, not overall. ("Pareto Principle" 80/20-Rule, Use profiling.)

- Program length does not necessarily correlate to speed.
- Some operations (for example shifts) might be much faster than others.
- There might be efficient libraries available. But not every existing library is optimal for your processor.
- Is Optimization worth the effort?

Time factors

The following tables give explicit timing factors obtained with a (particular) Pascal and C compiler:

		Relative Time Consumed	
Operation	**Example**	**Pascal**	**C**
Integer assignment	i = j	1	1
Integer addition/ subtraction	i = j + k	2	1.3
Integer multiplication	i = j * k	3	2
Integer division	i = j div k	5	4
Access integer array with constant subscript	i = a[5]	3	2
Access integer array with variable subscript	i = a[j]	3	4
Access two-dimensional integer array with constant subscripts	i = a[3, 5]	3	2

Access two-dimensional integer array with variable subscripts	i = a[j, k]	6	4
Access field of structured *variable*	i = rec.num	1	1
Access singly dereferenced pointer	i = rec^.num	2.5	2
Access doubly dereferenced pointer	i = rec^.next^.num	3.8	2.6
Each additional dereference	i = rec^.next^.next^...	1.3	0.6
Floating-point assignment	x = y	5	85
Floating-point addition/ subtraction	x = y + z	300	150
Floating-point multiplication/division	x = y * z	300	150
Floating-point square root	x = sqrt(y)	500	300
Floating-point sine	x = sin(y)	1000	700
Floating-point logarithm	x = log(y)	1800	1200
Floating-point e^x	x = exp(y)	2000	1300
Access floating-point array with constant subscript	x = z[5]	5	100
Access floating-point array with integer-variable subscript	x = z[j]	8	100
Access two-dimensional floating-point array with constant subscripts	x = z[3, 5]	5	100

Access two-dimensional floating-point array with integer-variable subscripts	x = z[j, k]	10	100
Routine call with no parameters	foo()	6	6
Routine call with 1 parameter	foo(i)	6	6
Routine call with 2 parameters	foo(i, j)	8	8

Source: S.McConnell: Code Complete

These numbers should just give an idea, values for a particular compiler might differ.

Overview Optimization Methods

These are methods that assume an idealized computer. Concrete implementations will have to take account of issues such as cache.

Trade space for time

- Extended data structures.
- Store precomputed results.
- Caching.
- Lazy evaluation.

Loops

- Move Code out of loops.
- Unswitching

- Inner loop is busiest.
- Loop unrolling.
- Loop fusion.

Logic/Expressions

- Common Subexpressions.
- Algebraic identities.
- Use simpler commands.
- Precomputation.
- Short-circuit/reordered testing.

Procedures

- Inline.

- Special treatment for common cases.
- Replace recursions.