The Abdus Salam
International Centre
for Theoretical Physics

Code Optimization I

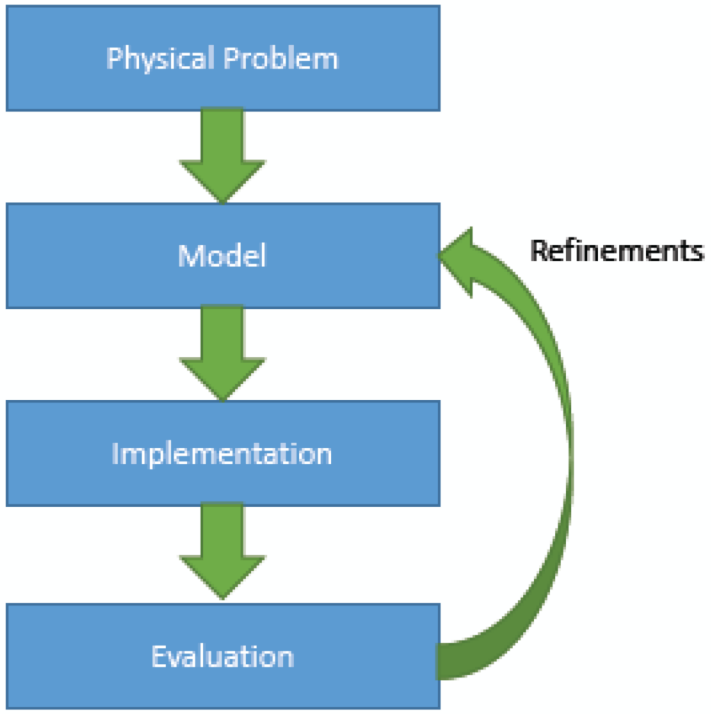
Ivan Girotto – igirotto@ictp.it

Information & Communication Technology Section (ICTS)

International Centre for Theoretical Physics (ICTP)

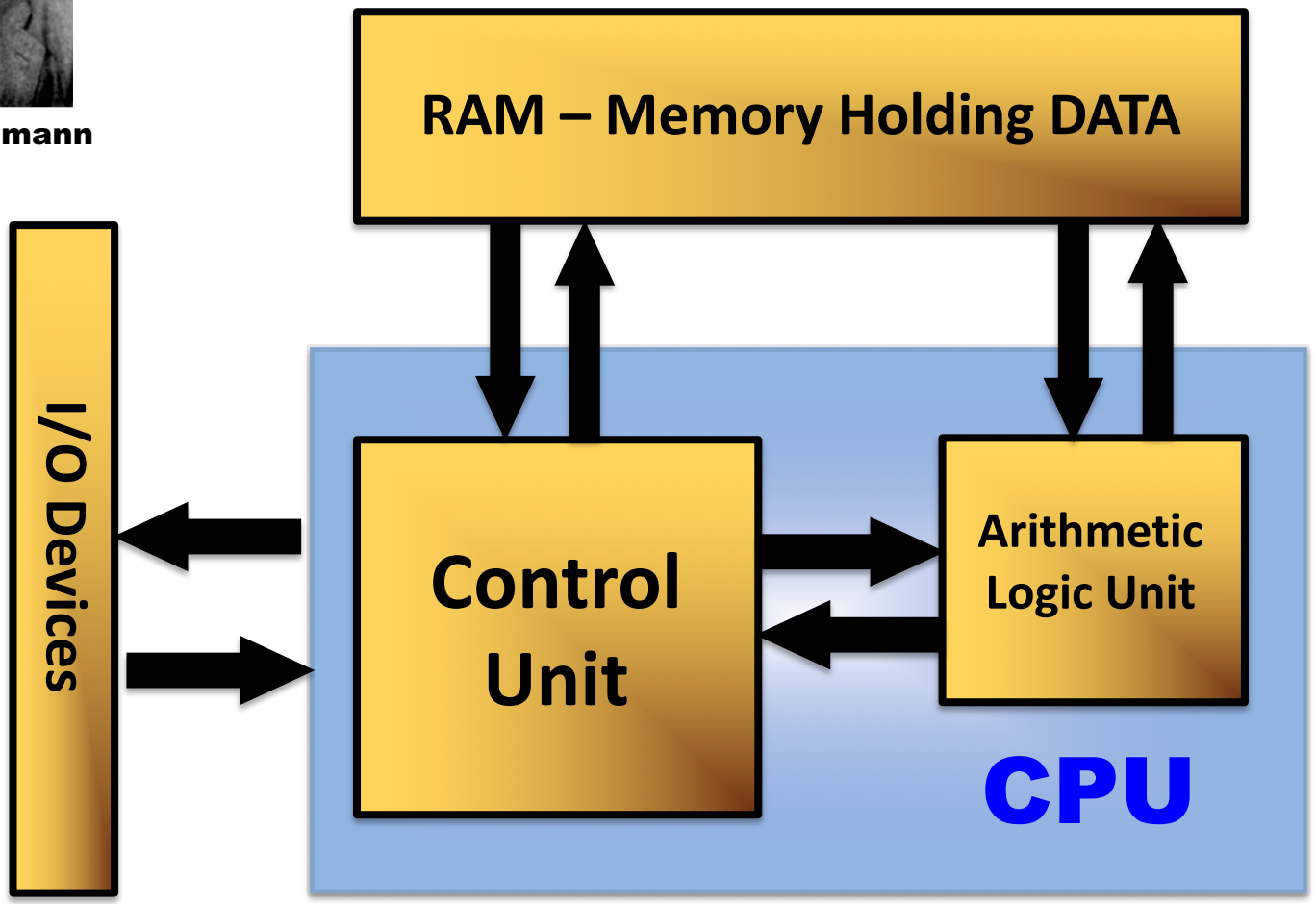
Scientific Development

- Correctness is the main concern
- Coding with no planning
- First version that seems to work is kept
- Sub-optimal choices are noticed only later on (if at all)



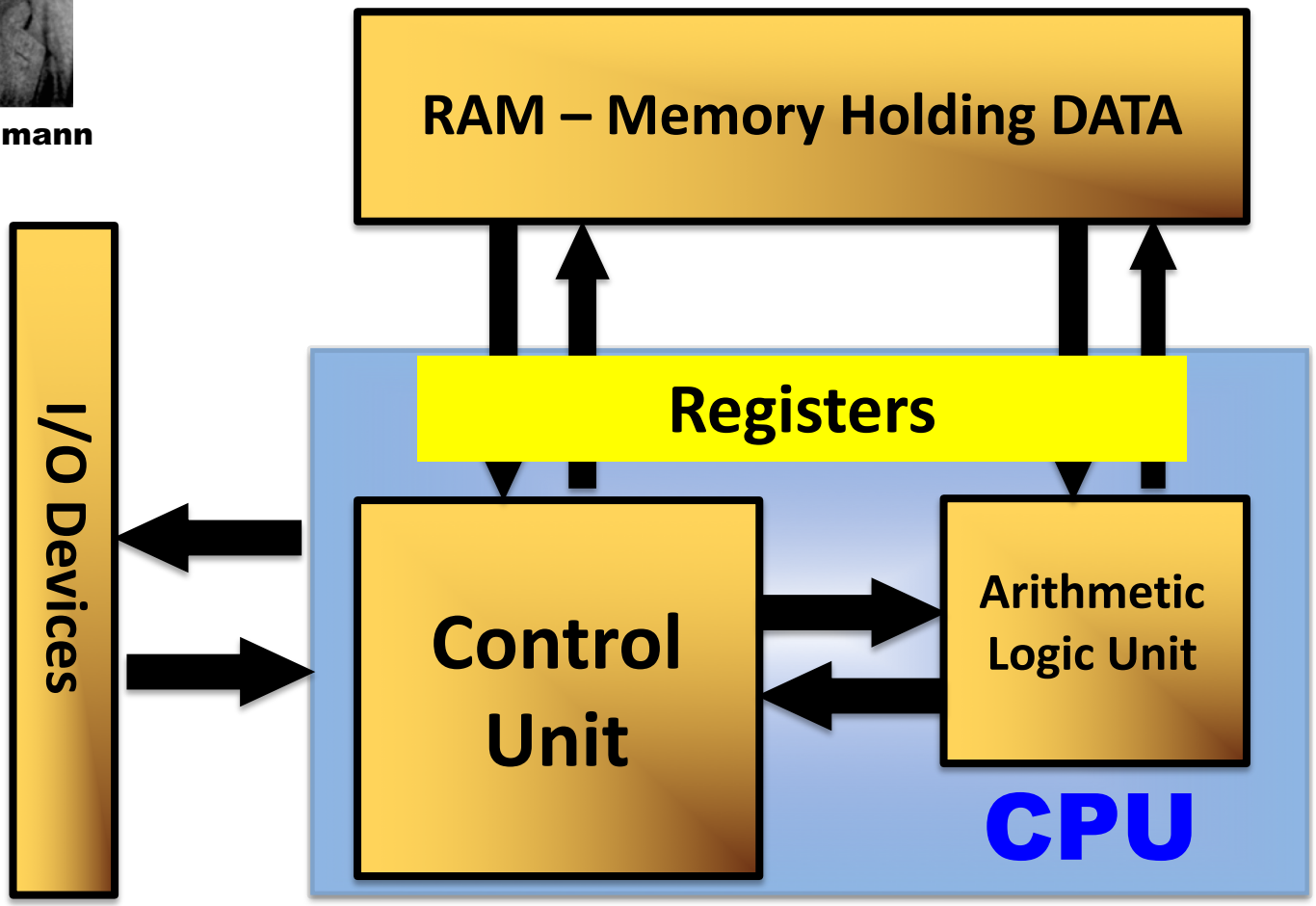
The Classical Model

John Von Neumann

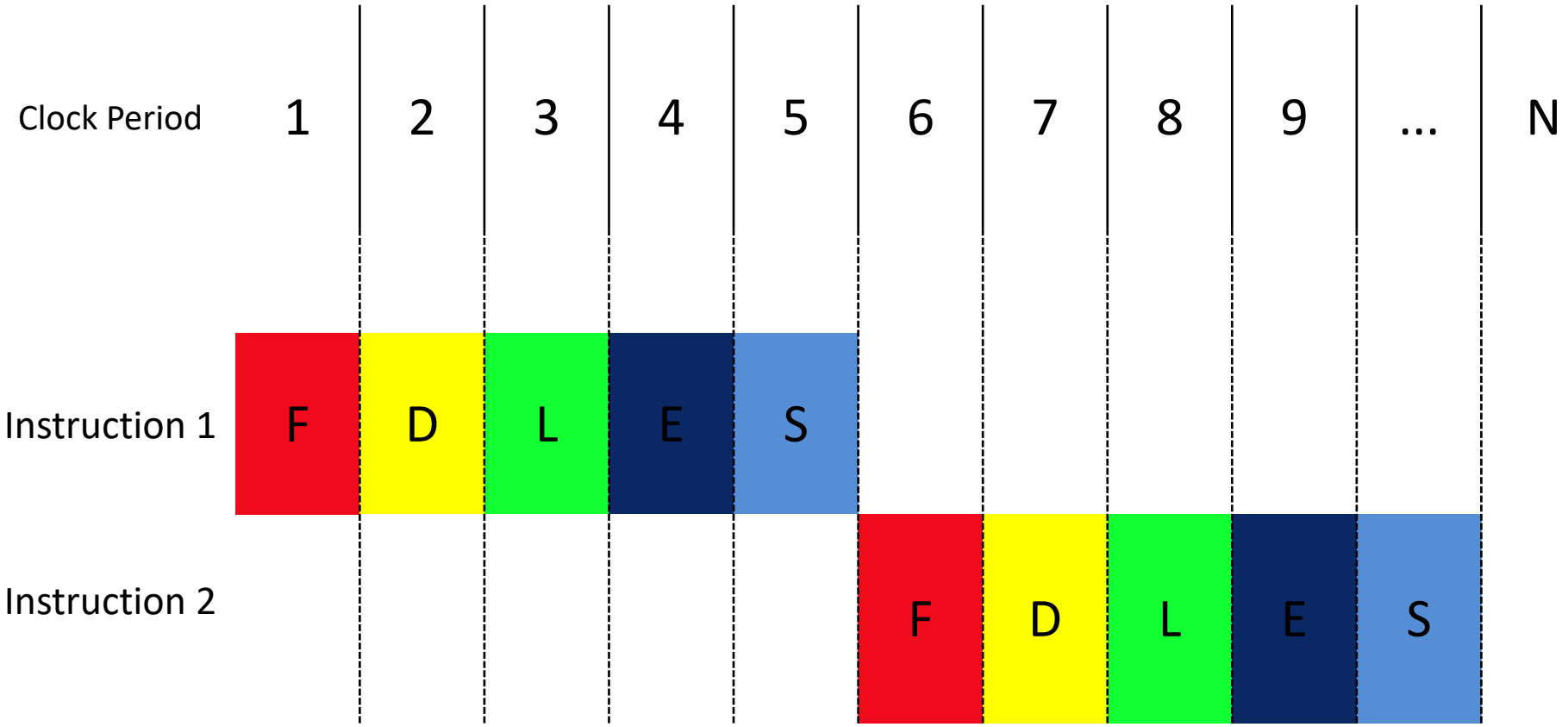


The Classical Model

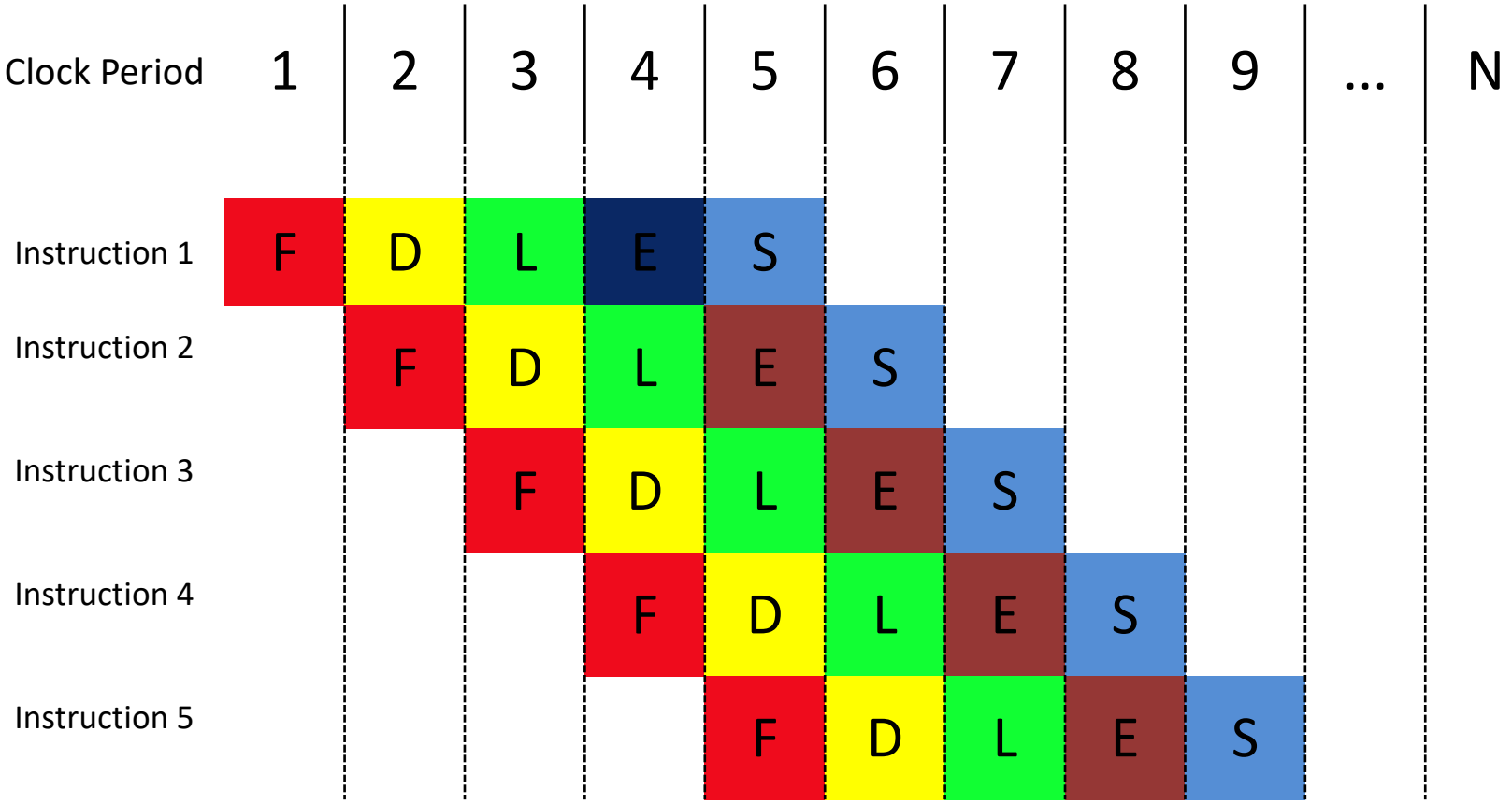
John Von Neumann



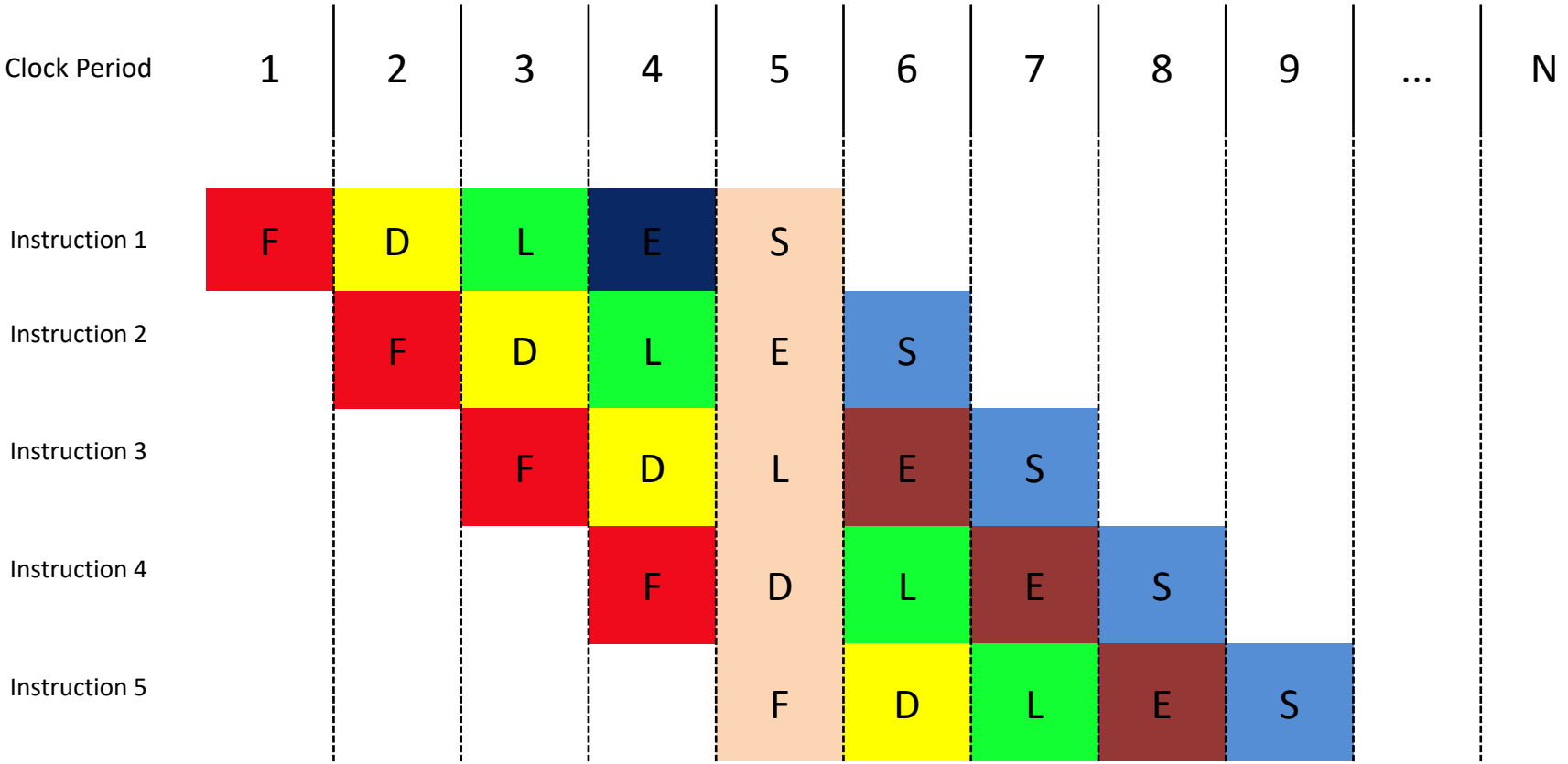
Sequential Processing



Pipelining


Clock Period
1
2
3
4
5
6
7
8
9
...
N
Instruction 1
Instruction 2
Instruction 3
Instruction 4
Instruction 5
F
D
L
E
S

Pipelining


Clock Period
1
2
3
4
5
6
7
8
9
...
N
Instruction 1
Instruction 2
Instruction 3
Instruction 4
Instruction 5
F
D
L
E
S

Superscalar Processors

Best case scenario

```
for( i = 0; i < N; i++ )
  a[ i ] = c[ i ] * b[ i ];
```

Worst case scenario

```
for( i = 1; i < N; i++ )
  a[ i ] = a[ i - 1 ] * b[ i ];
```

Worst case developer

```
for( i = 0; i < N; i++ )
  for( j = 0; j < N; j++ )
    {

      if( i == 0 ) a[ i ][ j ] = 0.0;
      else if( j == 0 ) a[ i ][ j ] = 0.0;
      else if( i == ( N - 1 ) ) a[ i ][ j ] = 0.0;
      else if( j == ( N - 1 ) ) a[ i ][ j ] = 0.0;
      else a[ i ][ j ] = func();
    }
```

Best case developer

```
for( i = 0; i < N; i++ )
  {
      a[ i ][ 0 ] = 0.0;
      a[ i ][ N - 1 ] = 0.0;
  }

for( j = 0; j < N; j++ )
  {
      a[ 0 ][ j ] = 0.0;
      a[ N - 1 ][ j ] = 0.0;
  }

for( i = 1; i < N - ; i++ )
  for( j = 1; j < N - 1; j++ )
    {
      a[ i ][ j ] = func();
    }
```

Copy Propagation

Before

```
x = y
z = 1 + x
```

Has data dependency

Compile

After

```
x = y
z = 1 + y
```

No data dependency

Constant Folding

Before

```
add = 100;
aug = 200;
sum = add + aug;
```

After

```
sum = 300;
```

sum is the sum of two constants. The compiler can precalculate the result (once) at compile time and eliminate code that would otherwise need to be executed at (every) run time.

Cost of operations

- Fast (0.5x-1x): add, subtract, multiply
- Medium (5-10x): divide, modulus, sqrt()
- Slow (20-50x): most transcendental functions
- Very slow (>100x): power (xy for real *x* and *y)*
- Often **only** the fastest operations are pipelined, so code will be the fastest when using only multiply/add
  - BLAS (= Basic Linear Algebra Subroutines)
  - LAPACK (Linear Algebra Package)

Reduction of costing operations

Before

```
x = pow(y, 2);
a = c / 2.0;
```

After

```
x = y * y;
a = c * 0.5;
```

Raising one value to the power of another, or dividing, is more expensive than multiplying.

If the compiler can tell that the power is a small integer, or that the denominator is a constant, it will use multiplication instead.

Common Subexpression Elimination

Before

```
d = c * (a / b);
e = (a / b) * 2.0;
```

After

```
adivb = a / b;
d = c * adivb;
e = adivb * 2.0;
```

The subexpression `(a / b)` occurs in both assignment statements, so there's no point in calculating it twice.

This is typically only worth doing if the common subexpression is expensive to calculate, or the resulting code requires the use of less registers.

Variable renaming

Before

```
x = y * z;
q = r + x * 2;
x = a + b;
```

After

```
x0 = y * z;
q = r + x0 * 2;
x = a + b;
```

The original code has an **output dependency**, while the new code **doesn't** – but the final value of `x` is still correct.

Hoisting Loop Invariant Code

Code that doesn't change inside the loop is known as *loop invariant*. It doesn't need to be calculated over and over.

Before

```
DO i = 1, n
   a(i) = b(i) + c * d
   e = g(n)
END DO
```

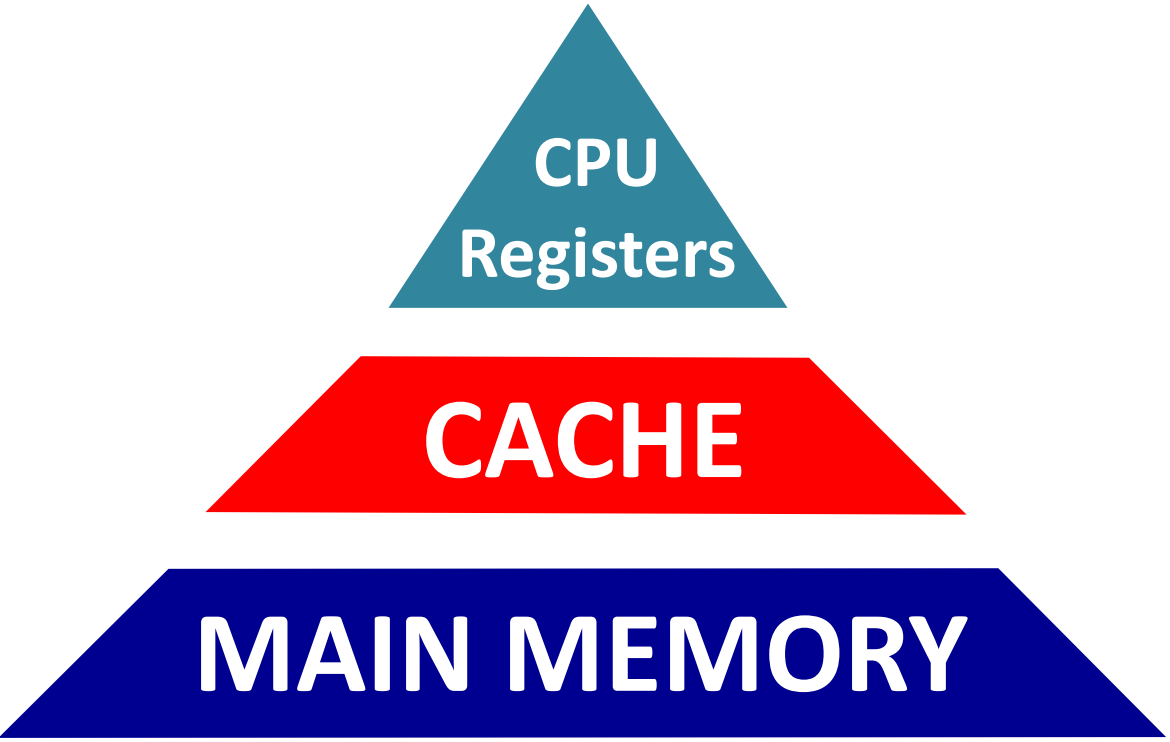
After

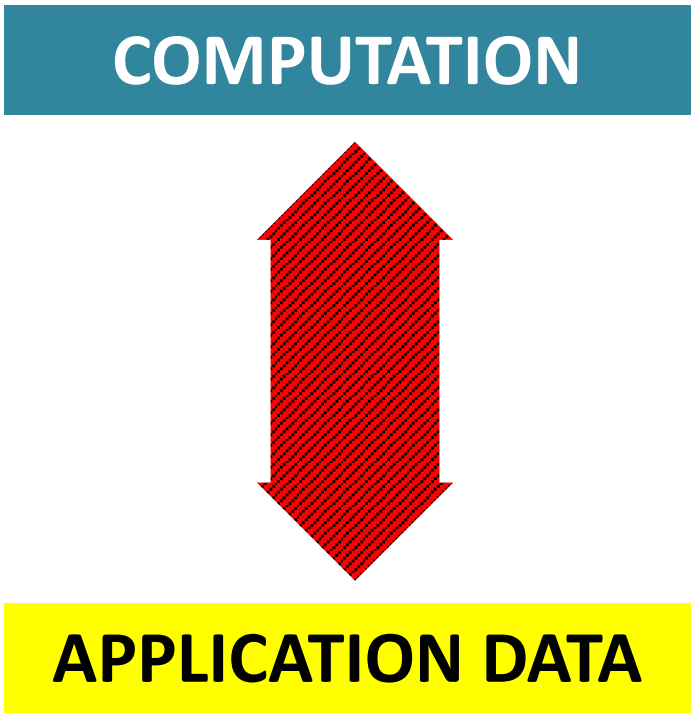
```
temp = c * d
DO i = 1, n
   a(i) = b(i) + temp
END DO
e = g(n)
```

Few basic rules for optimized codes

- Do less work!!
  - Elimination of common sub-expressions
- Avoid expensive operations
  - Reduce your math to cheap operations
  - Avoid branches
- Think as a the compiler works
  - Enhance the compiler

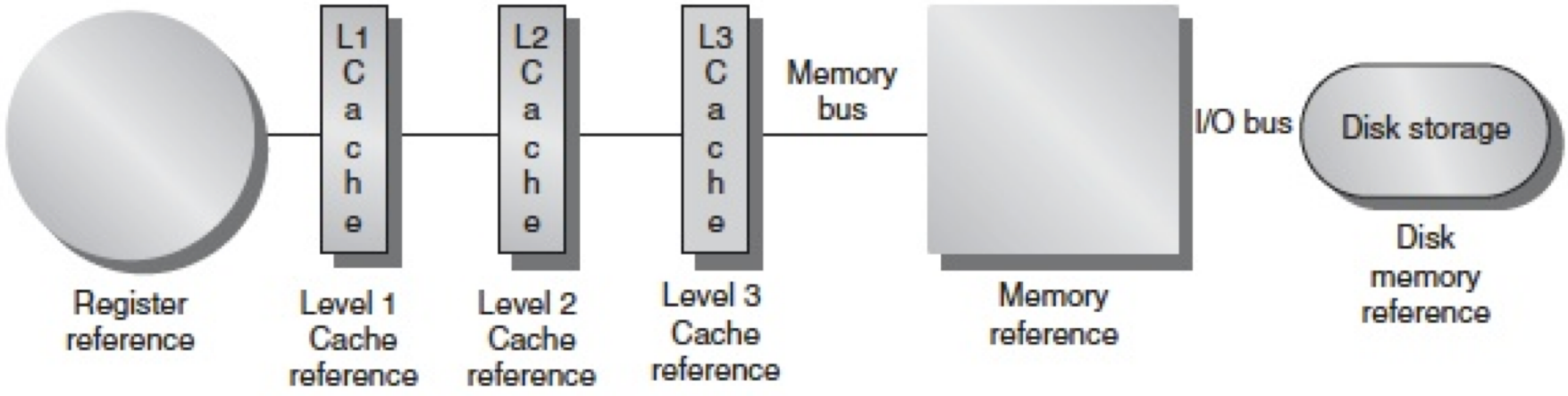
The CPU Memory Hierarchy





- **Cache is designed for temporal/spatial locality**

The CPU Memory Hierarchy



(a) Memory hierarchy for server

Blocking for Cache

- Blocking for cache is
  - An optimization that applies for datasets that do not fit entirely into cache
  - A way to increase spatial locality of reference i.e. exploit full cache lines
  - A way to increase temporal locality of reference i.e. improves data reuse
- Example, the transposing of a matrix

```
do i=1,n
  do j=1,n
     a(i,j)=b(j,i)
  end do
end do
```

Transposing Data

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16

Transposing Data – Step 1

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

1	2
5	6

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

- Copy the data on the buffer block

Transposing Data – Step 2

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

1	5
2	6

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

- Transpose the block

Transposing Data – Step 3

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

1	5
2	6

1	5	0	0
2	6	0	0
0	0	0	0
0	0	0	0

- Copy the transposed block from the buffer block to the destination matrix

Transposing Data – Step 4

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

9	10
13	14

1	5	0	0
2	6	0	0
0	0	0	0
0	0	0	0

- Iterates over blocks

Transposing Data – Step 5

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

9	13
10	14

1	5	0	0
2	6	0	0
0	0	0	0
0	0	0	0

- Iterates over blocks

Transposing Data – Step 6

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

9	13
10	14

1	5	9	13
2	6	10	14
0	0	0	0
0	0	0	0

- Iterates over blocks

Blocking for Cache

- block data size = bsize
  - mb = n/bsize
  - nb = n/bsize
- These sizes can be manipulated to coincide with actual cache sizes on individual architectures

```
do ib = 1, nb
 ioff = (ib-1) * bsiz
 do jb = 1, mb
   joff = (jb-1) * bsiz
    do j = 1, bsiz
      do i = 1, bsiz
        buf(i,j) = x(i+ioff, j+joff)
     enddo
    enddo
    do j = 1, bsiz
      do i = 1, j-1
       bswp = buf(i,j)
       buf(i,j) = buf(j,i)
       buf(j,i) = bswp
      enddo
     enddo
    do i=1,bsiz
      do j=1,bsiz
       y(j+joff, i+ioff) = buf(j,i)
      enddo
    enddo
  enddo
enddo
```

Data Memory Access

- Data ordering
- Reduce at minimum the data transfers
- Avoid complex data structure within computational intensive kernels
- Define constants and help the compiler
- Exploit the memory hierarchy