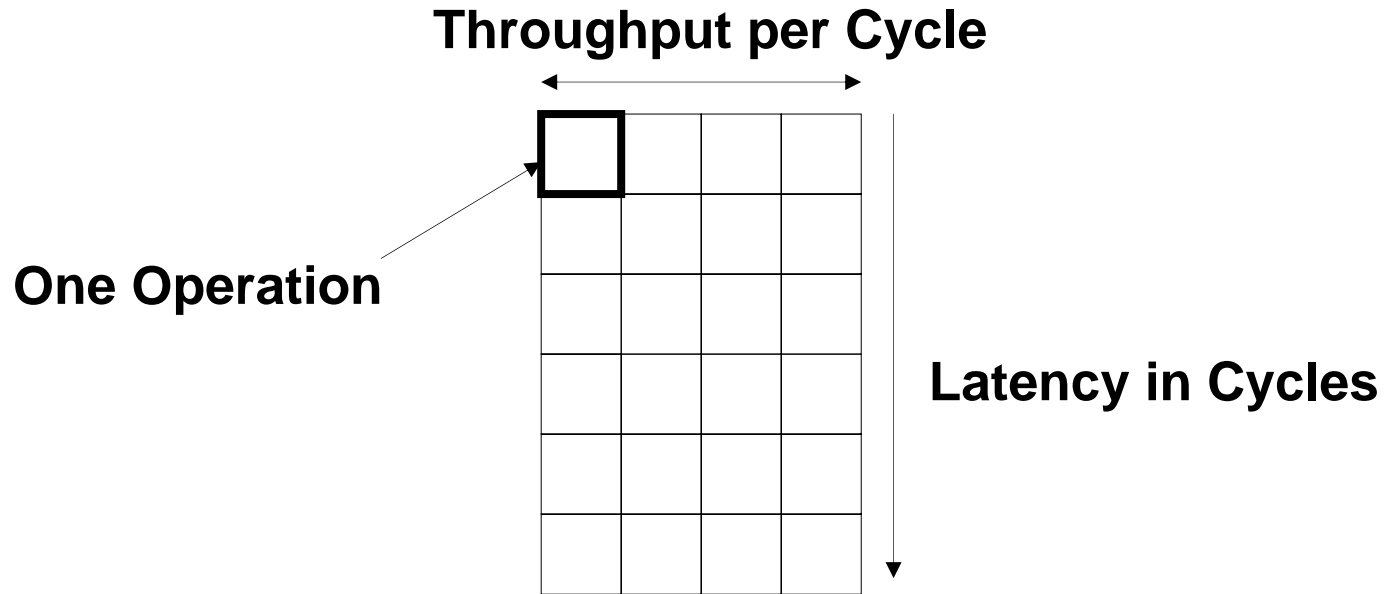

Exploiting Parallelisms

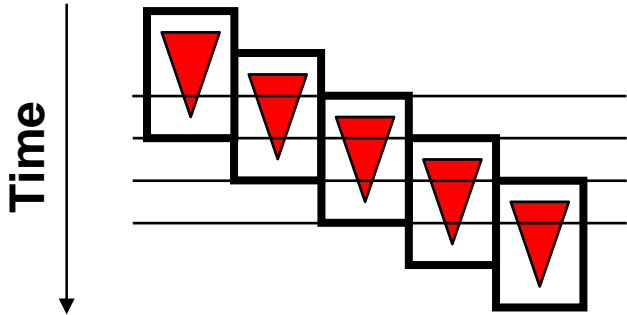
Little's Law



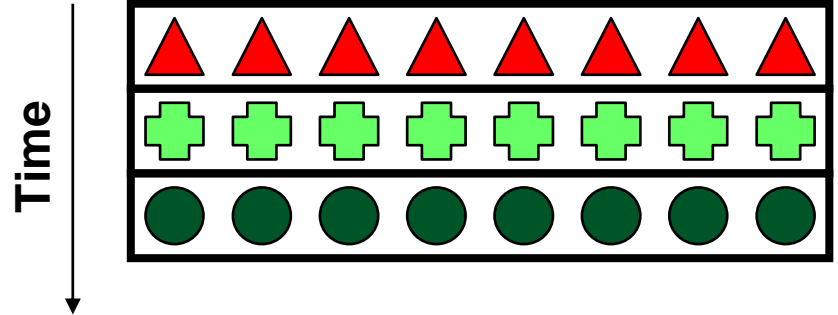
$$\text{Parallelism} = \text{Throughput} * \text{Latency}$$

- ❑ To maintain throughput T/cycle when each operation has latency L cycles, need $T*L$ *independent* operations
- ❑ For fixed parallelism:
 - decreased latency allows increased throughput
 - decreased throughput allows increased latency tolerance

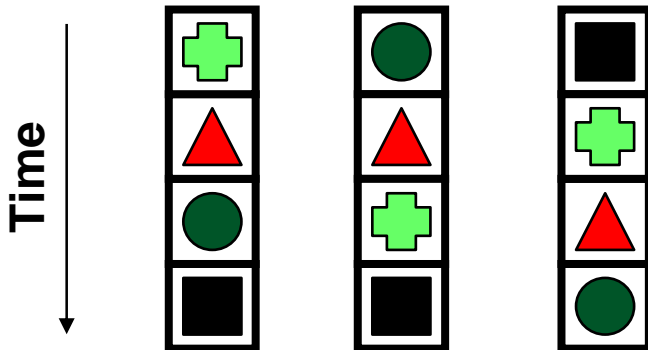
Types of Parallelism



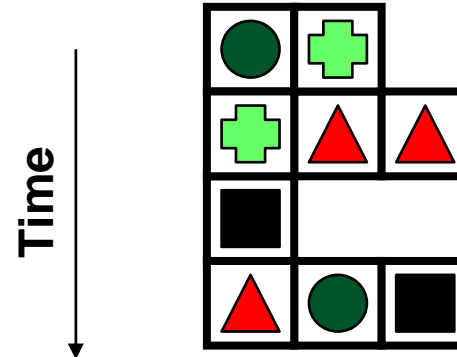
Pipelining



Data-Level Parallelism (DLP)



Thread-Level Parallelism (TLP)



Instruction-Level Parallelism (ILP)

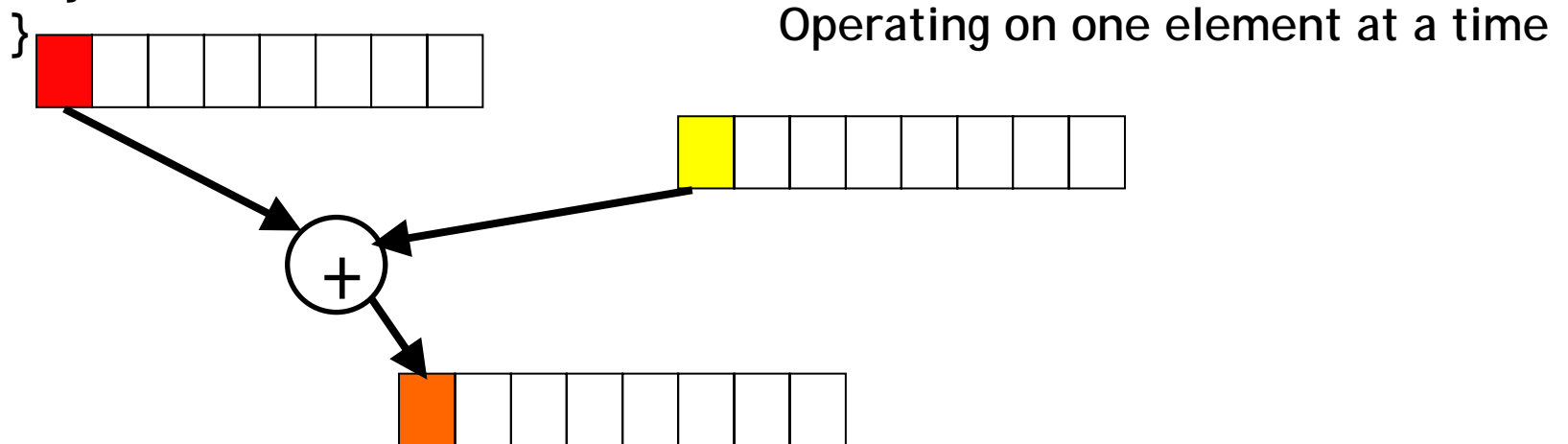
Flynn's Classification (1966)

- ❑ **Broad classification of parallel computing systems based on number of instruction and data streams**
- ❑ **SISD: Single Instruction, Single Data**
 - conventional uniprocessor
- ❑ **SIMD: Single Instruction, Multiple Data**
 - one instruction stream, multiple data paths
- ❑ **MIMD: Multiple Instruction, Multiple Data**
 - message passing machines (Transputers, nCube, CM-5)
 - non-cache-coherent shared memory machines (BBN Butterfly, T3D)
 - cache-coherent shared memory machines (Sequent, Sun Starfire, SGI Origin)
- ❑ **MISD: Multiple Instruction, Single Data**
 - no commercial examples

SIMD

- ❑ Consider adding together two arrays:

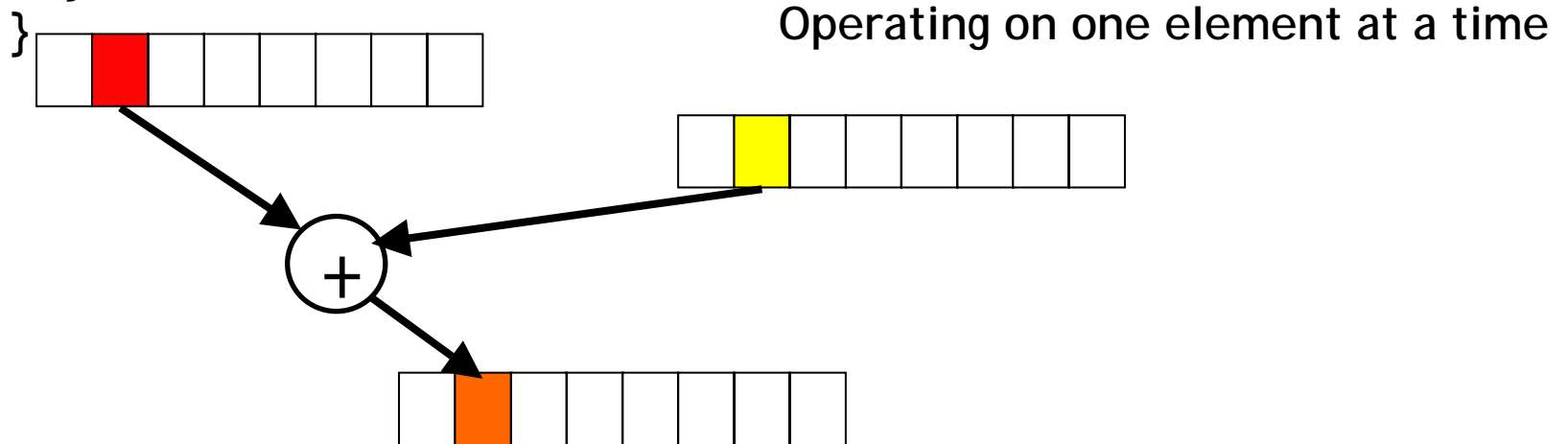
```
void  
array_add(int A[], int B[], int C[], int length) {  
    int i;  
    for (i = 0 ; i < length ; ++ i) {  
        C[i] = A[i] + B[i];  
    }  
}
```



SIMD

- ❑ Consider adding together two arrays:

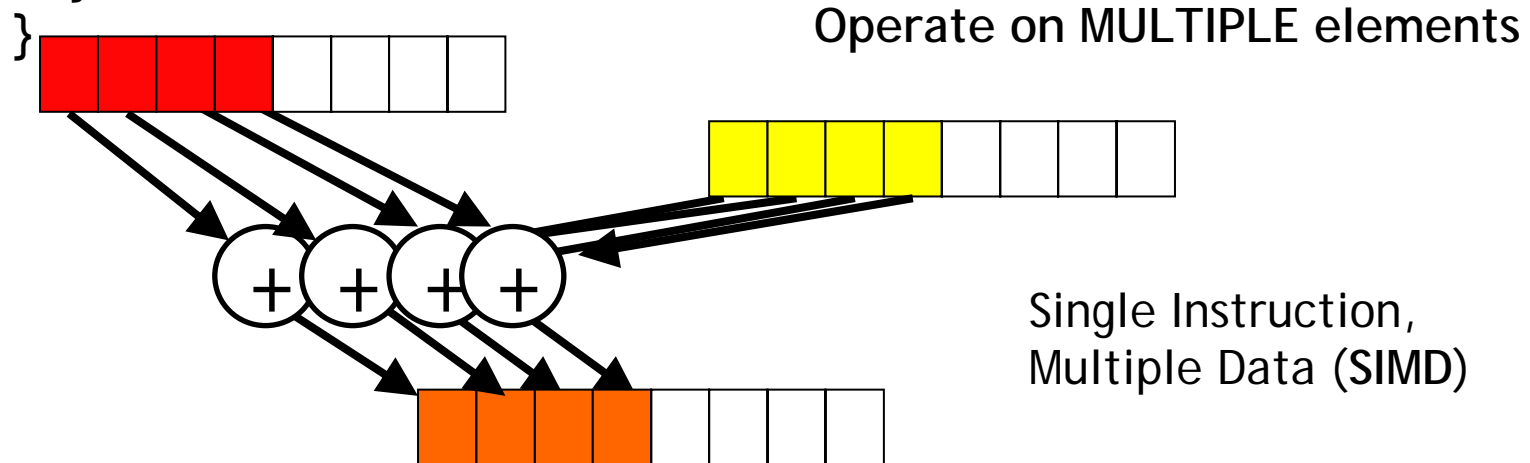
```
void  
array_add(int A[], int B[], int C[], int length) {  
    int i;  
    for (i = 0 ; i < length ; ++ i) {  
        C[i] = A[i] + B[i];  
    }  
}
```



SIMD

- ❑ Consider adding together two arrays:

```
void  
array_add(int A[], int B[], int C[], int length) {  
    int i;  
    for (i = 0 ; i < length ; ++ i) {  
        C[i] = A[i] + B[i];  
    }  
}
```



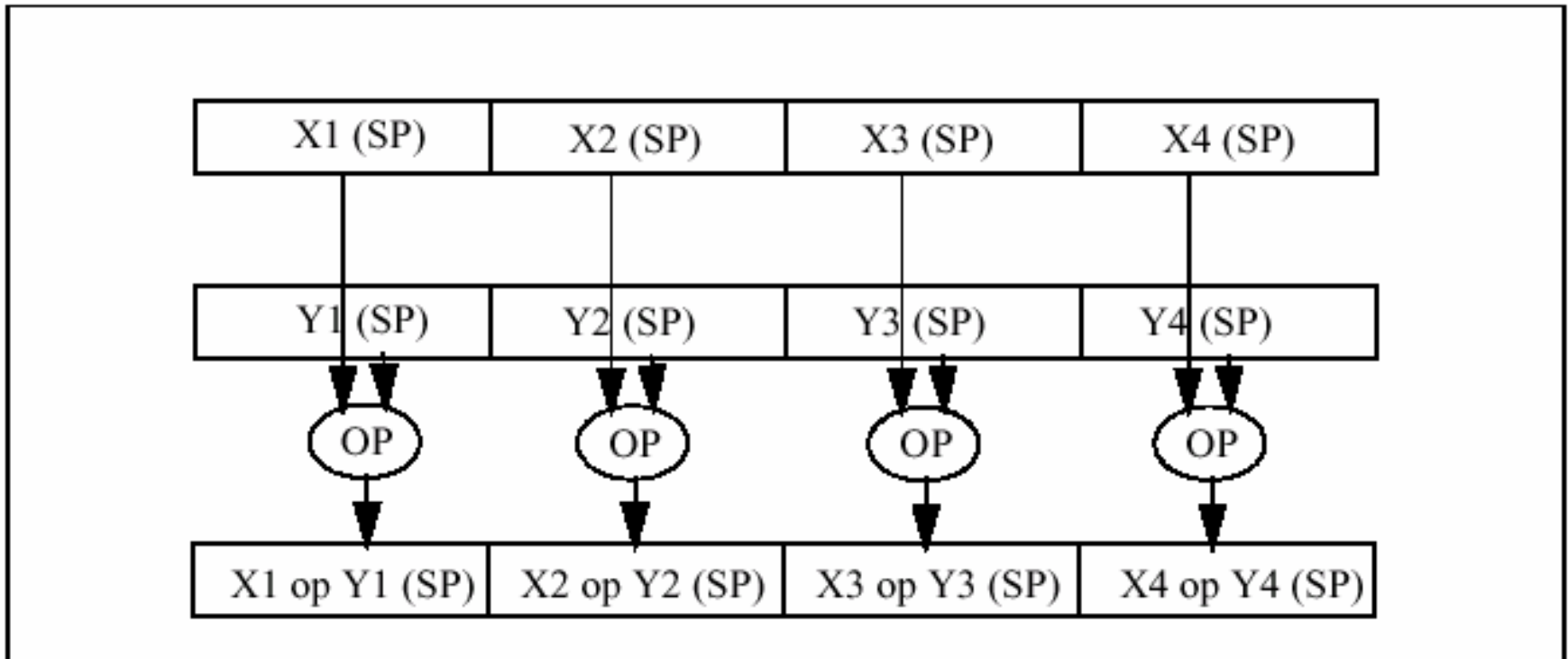
Intel SSE/SSE2 as an example of SIMD

Added new 128 bit registers (XMM0 - XMM7), each can store

- 4 single precision FP values (SSE) $4 * 32b$
- 2 double precision FP values (SSE2) $2 * 64b$
- 16 byte values (SSE2) $16 * 8b$
- 8 word values (SSE2) $8 * 16b$
- 4 double word values (SSE2) $4 * 32b$
- 1 128-bit integer value (SSE2) $1 * 128b$

	4.0 (32 bits)	4.0 (32 bits)	3.5 (32 bits)	-2.0 (32 bits)
+	-1.5 (32 bits)	2.0 (32 bits)	1.7 (32 bits)	2.3 (32 bits)
<hr/>				
	2.5 (32 bits)	6.0 (32 bits)	5.2 (32 bits)	0.3 (32 bits)

SIMD Extensions



Packed Operations

More than 70 instructions. Arithmetic Operations supported: Addition, Subtraction, Mult, Division, Square Root, Maximum, Minimum. Can operate on Floating point or Integer data.

Is it always that easy?

- ❑ No. Not always. Let's look at a little more challenging one.

```
unsigned
sum_array(unsigned *array, int length) {
    int total = 0;
    for (int i = 0 ; i < length ; ++ i) {
        total += array[i];
    }
    return total;
}
```

- ❑ Is there parallelism here?

Exposing the parallelism

```
unsigned
sum_array(unsigned *array, int length) {
    int total = 0;

    for (int i = 0 ; i < length ; ++ i) {
        total += array[i];
    }

    return total;
}
```

We first need to restructure the code

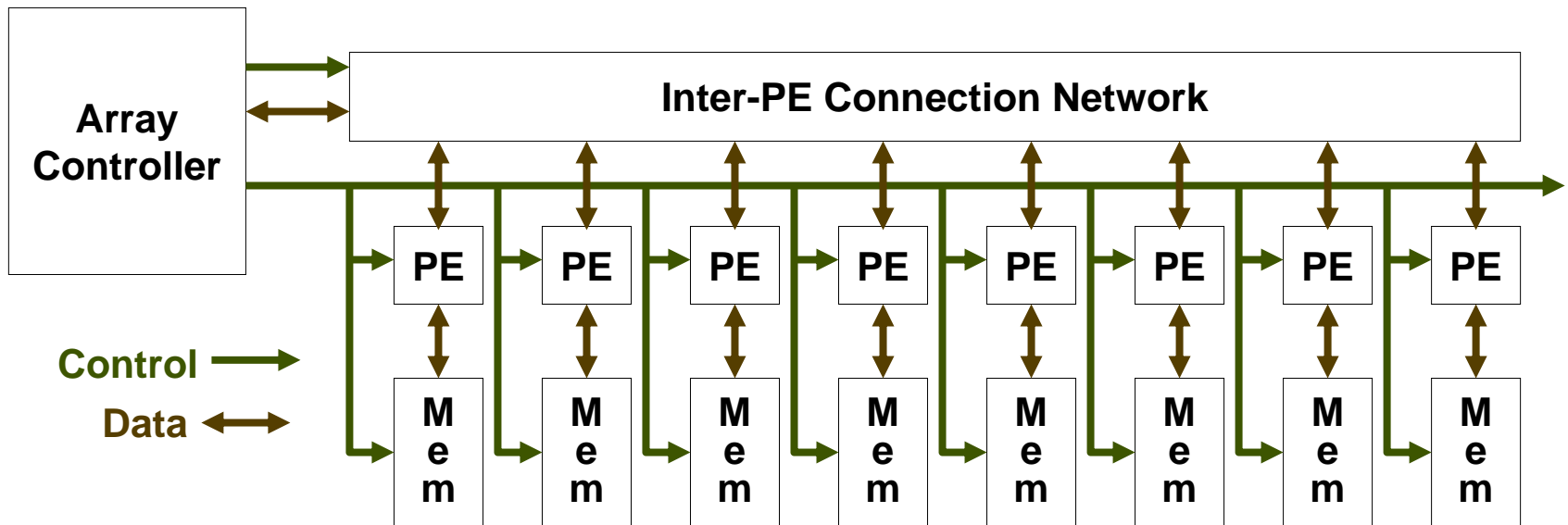
```
unsigned
sum_array2(unsigned *array, int length) {
    unsigned total, i;
    unsigned temp[4] = {0, 0, 0, 0};
    for (i = 0 ; i < length & ~0x3 ; i += 4) {
        temp[0] += array[i];
        temp[1] += array[i+1];
        temp[2] += array[i+2];
        temp[3] += array[i+3];
    }
    total = temp[0] + temp[1] + temp[2] + temp[3];
    for ( ; i < length ; ++ i) {
        total += array[i];
    }
    return total;
}
```

Then we can write SIMD code for the hot part

```
unsigned
sum_array2(unsigned *array, int length) {
    unsigned total, i;
    unsigned temp[4] = {0, 0, 0, 0};
    for (i = 0 ; i < length & ~0x3 ; i += 4) {
        temp[0] += array[i];
        temp[1] += array[i+1];
        temp[2] += array[i+2];
        temp[3] += array[i+3];
    }
    total = temp[0] + temp[1] + temp[2] + temp[3];
    for ( ; i < length ; ++ i) {
        total += array[i];
    }
    return total;
}
```

SIMD Architecture

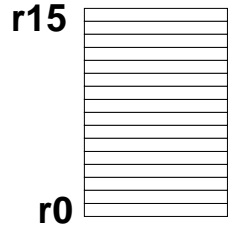
- ❑ Central controller broadcasts instructions to multiple processing elements (PEs)



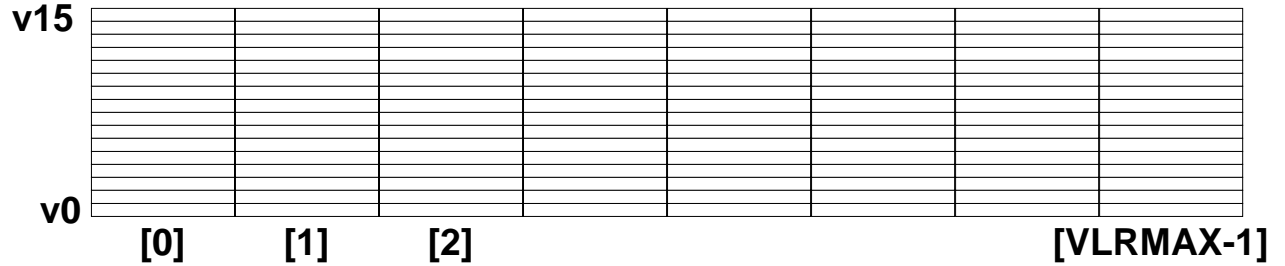
- Only requires one controller for whole array
- Only requires storage for one copy of program
- All computations fully synchronized

Vector Register Machine

Scalar Registers



Vector Registers

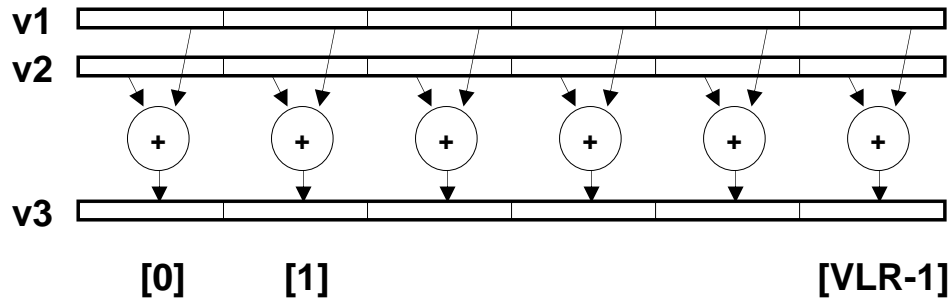


Vector Length Register



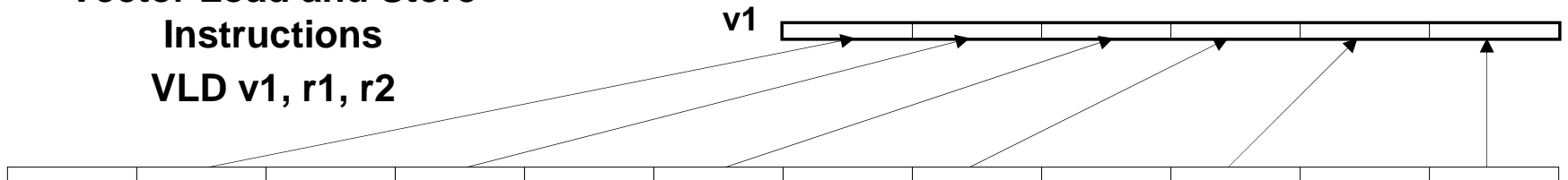
Vector Arithmetic Instructions

VADD v3, v1, v2

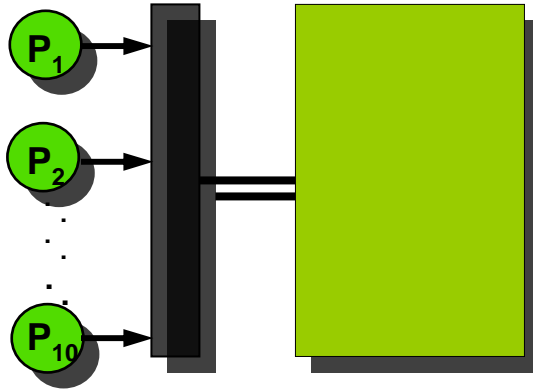


Vector Load and Store Instructions

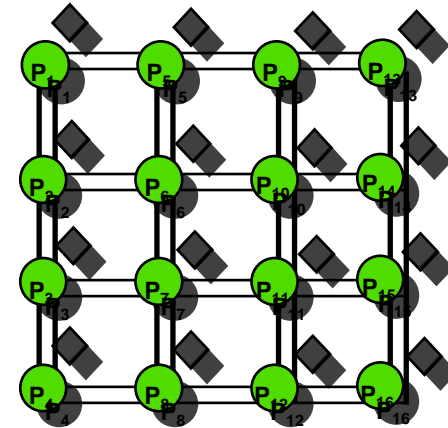
VLD v1, r1, r2



Traditional MIMD Architectures



Shared Bus



Distributed

❑ Or, a multiprocessor system

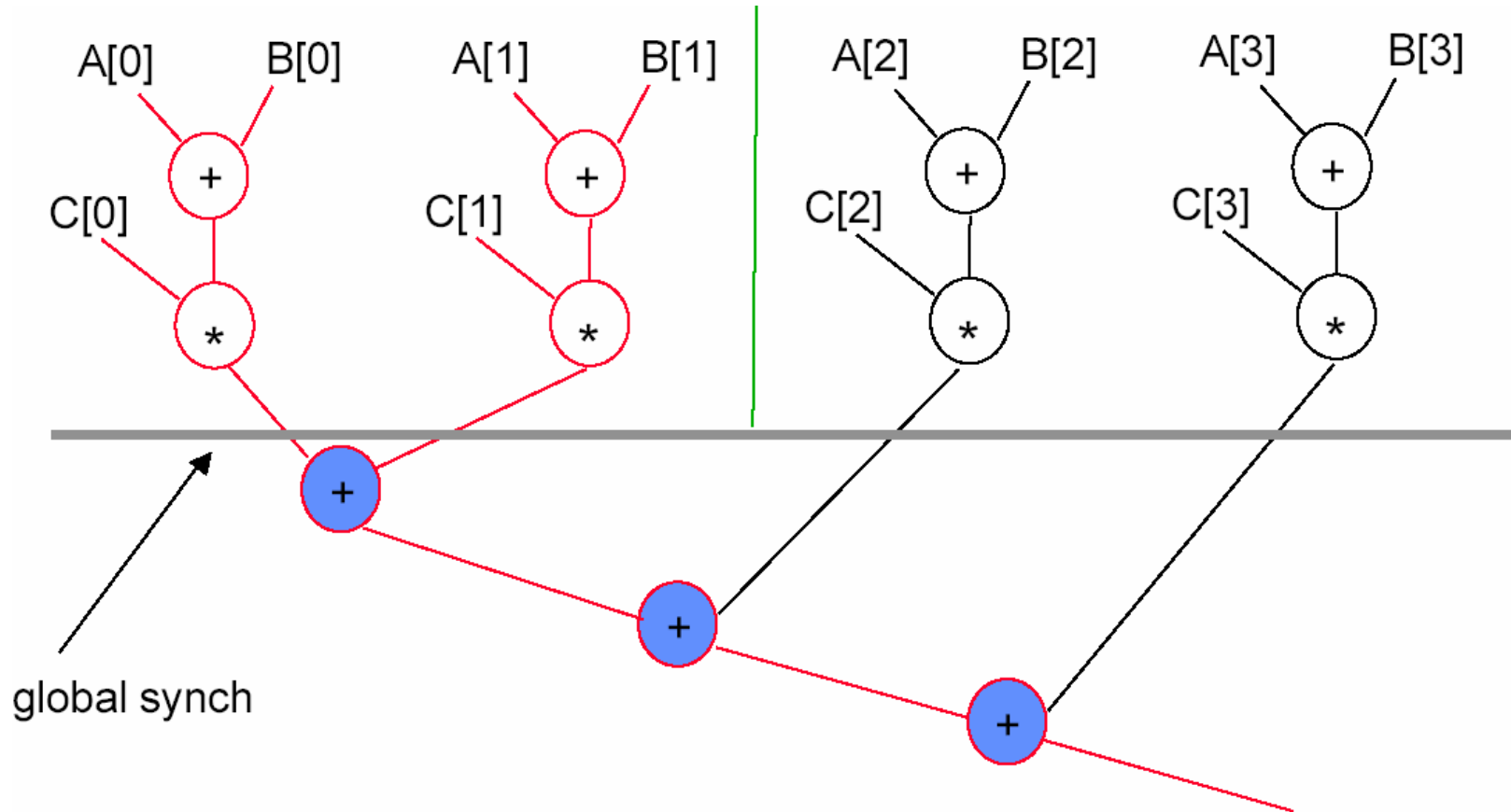
Simple Problem

```
for i = 1 to N
    A[i] = (A[i] + B[i]) * C[i]
sum = sum + A[i]
```

- Split the loops
 - Independent iterations

```
for i = 1 to N
    A[i] = (A[i] + B[i]) * C[i]
for i = 1 to N
    sum = sum + A[i]
```

Partitioning of Data Flow Graph



Simple Problem: Shared Memory

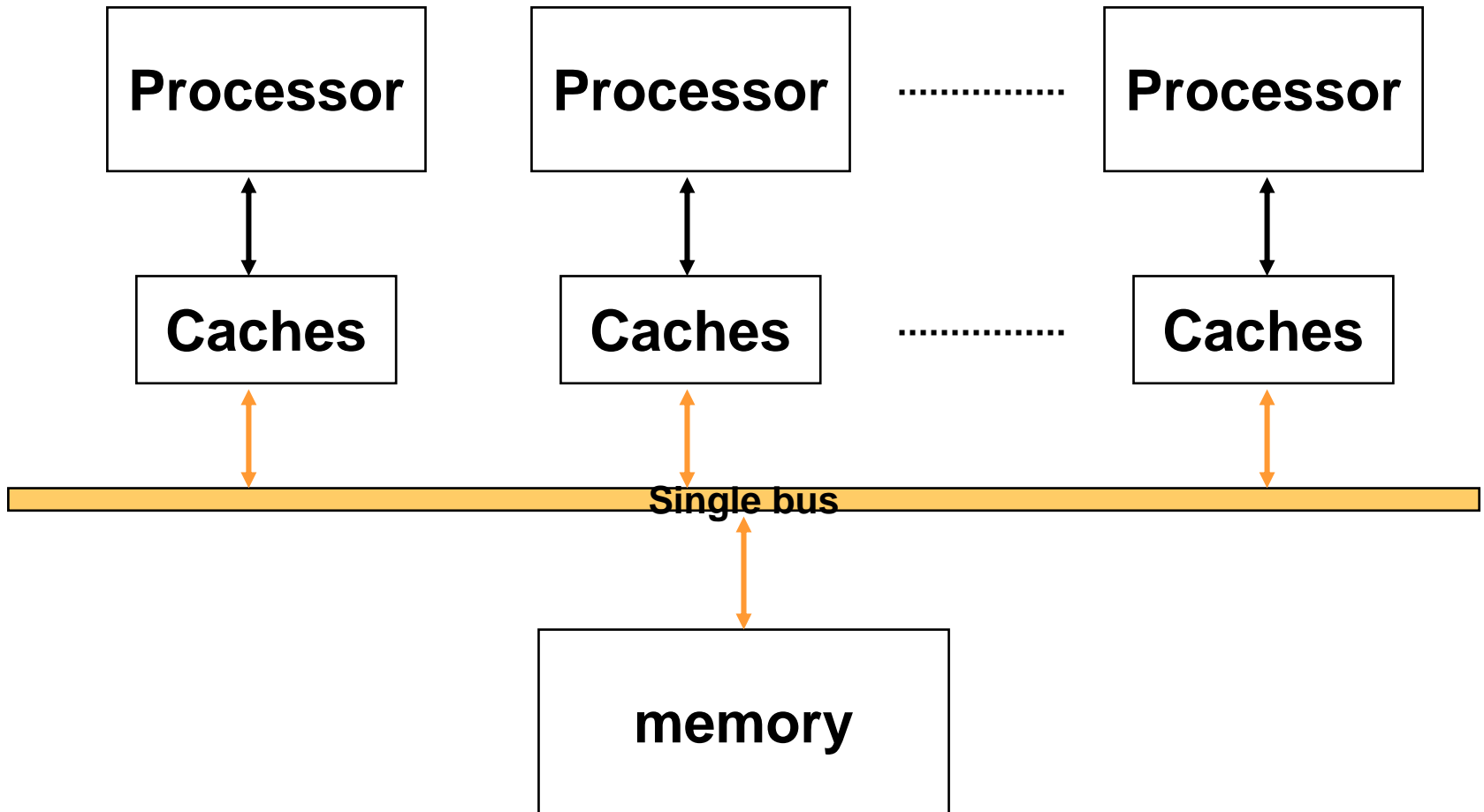
```
private int i, my_start, my_end, mynode;
shared float A[N], B[N], C[N], sum;
for i = my_start to my_end
    A[i] = (A[i] + B[i]) * C[i]
GLOBAL_SYNC;
if (mynode == 0)
    for i = 1 to N
        sum = sum + A[i]
```

- Can run this on any shared memory machine

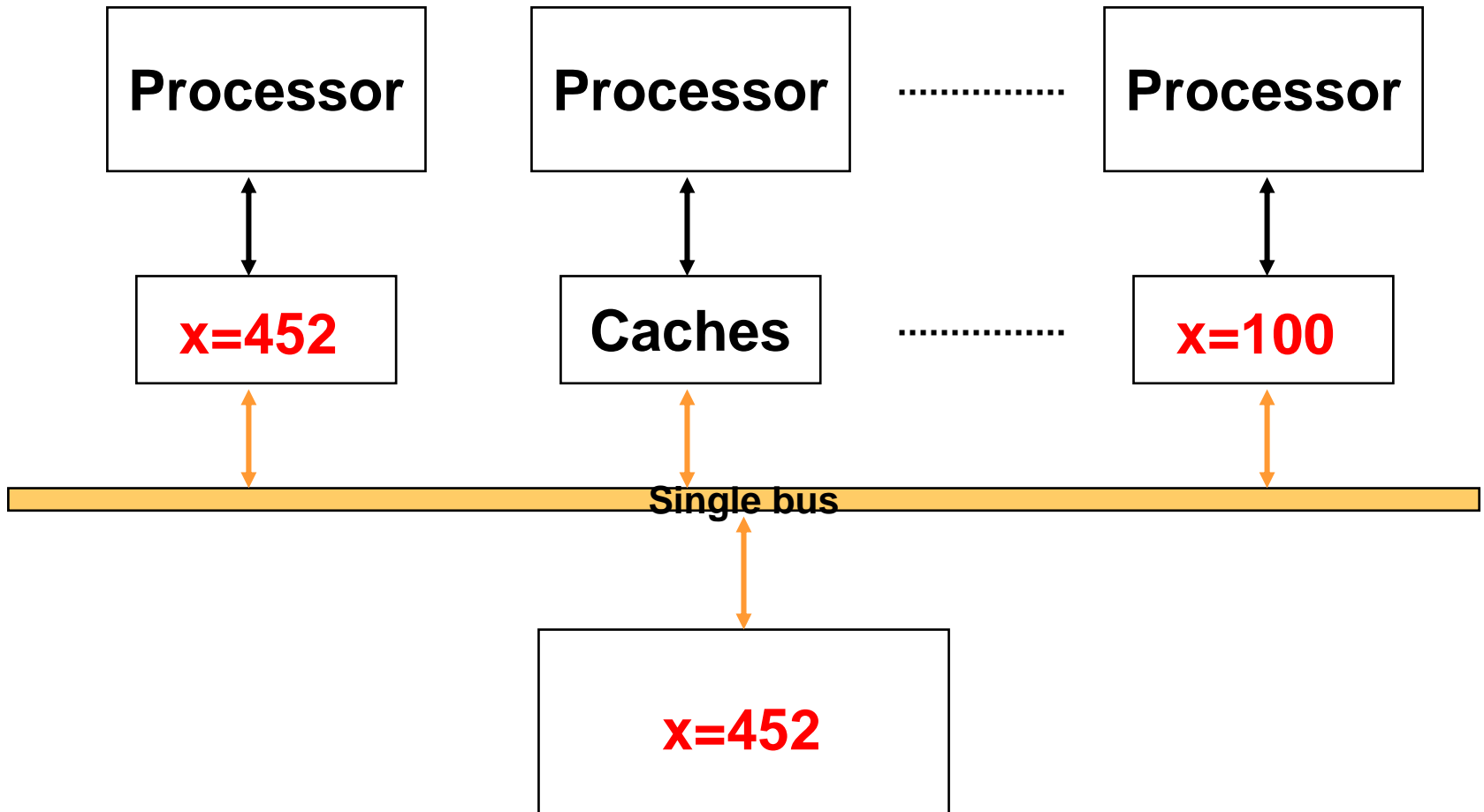
Multiprocessor Questions

- ❑ **How do parallel processors share data?**
 - single address space (SMP, NUMA)
 - message passing (clusters, massively parallel processors (MPP))
- ❑ **How do parallel processors coordinate?**
 - software synchronization (locks, semaphores)
 - built into send / receive primitives
 - OS primitives (sockets)
- ❑ **How are parallel processors interconnected?**
 - connected by a single bus
 - connected by a network

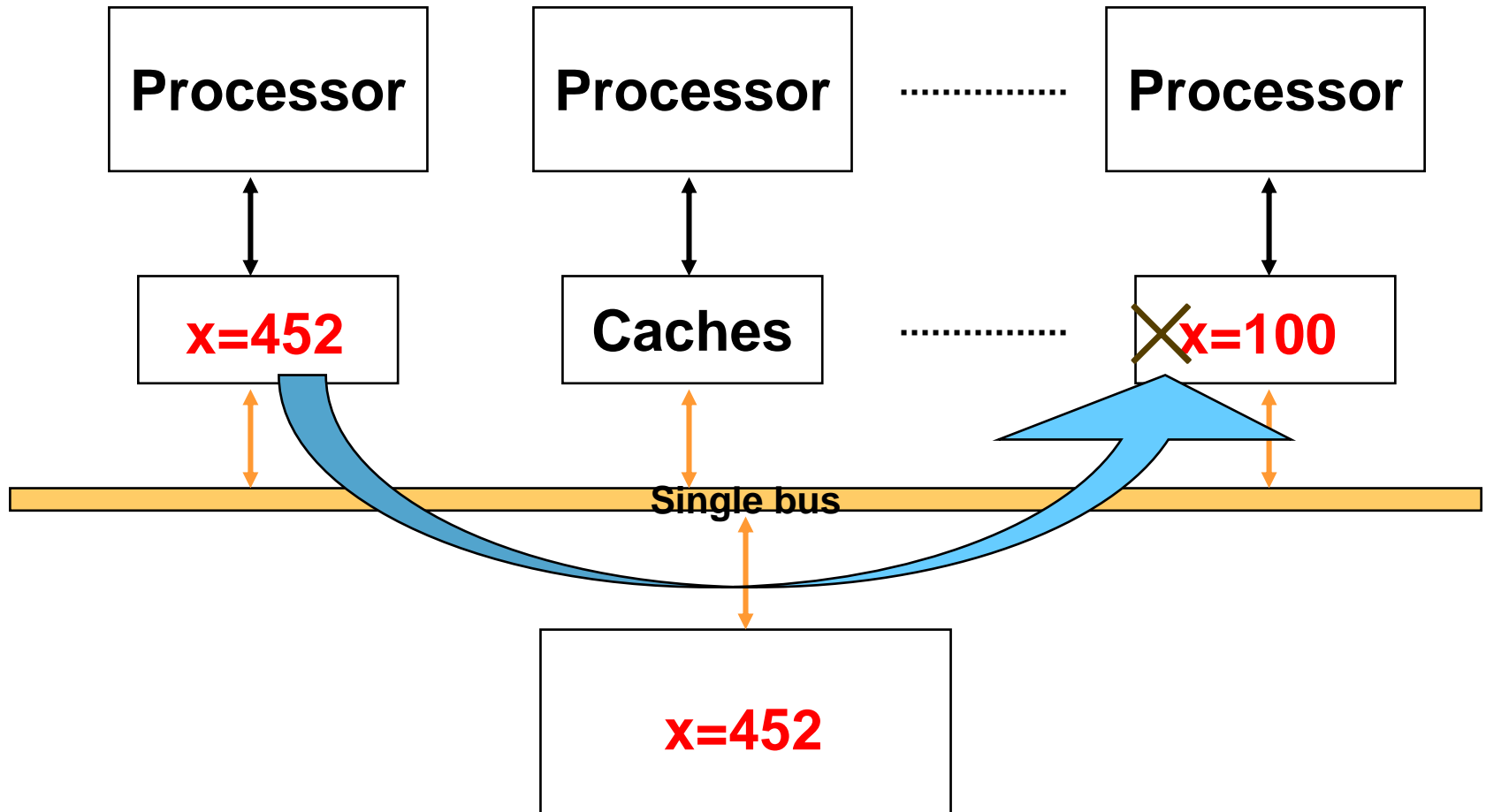
SMP



Challenges



Snoopy Cache Coherence



Snoopy Cache Coherence

□ Solution

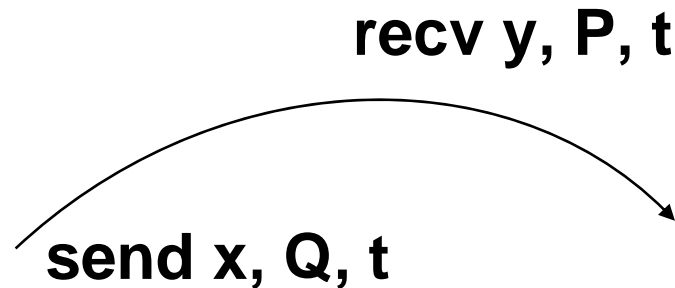
- Have caches “snoop” the contents of other caches on reads and writes
- Writes: Invalidate or update other caches’ copies
- Reads: Get latest version from other cache, if it has a “dirty” copy
- Need extra tags on caches to handle numerous requests

Message Passing Programming Model

local process **P**
address space



local process **Q**
address space



- User level send/receive abstraction
 - local buffer (x,y), process (Q,P) and tag (t)
 - Provides naming *and* synchronization

Simple Problem: Message Passing

```
int i, my_start, my_end, mynode;
float A[N/P], B[N/P], C[N/P], sum;
for i = 1 to N/P
    A[i] = (A[i] + B[i]) * C[i]
    sum = sum + A[i]
if (mynode != 0)
    send (sum, 0);
else // mynode == 0
    for i = 1 to P-1
        recv(tmp, i)
        sum = sum + tmp
```

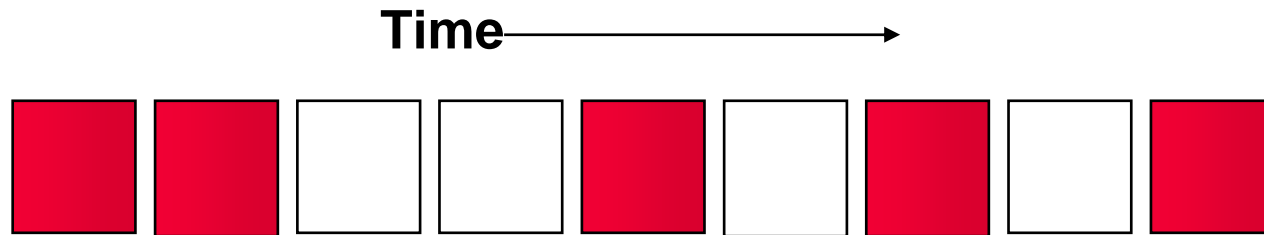
- ❑ P processors
- ❑ Send/Recv communicates *and* synchronizes

MIMD on-chip: SMT and CMP

- ❑ **Squeeze multiple parallel processing units on-chip**
 - **Minimize communication/synchronization/coherence cost**

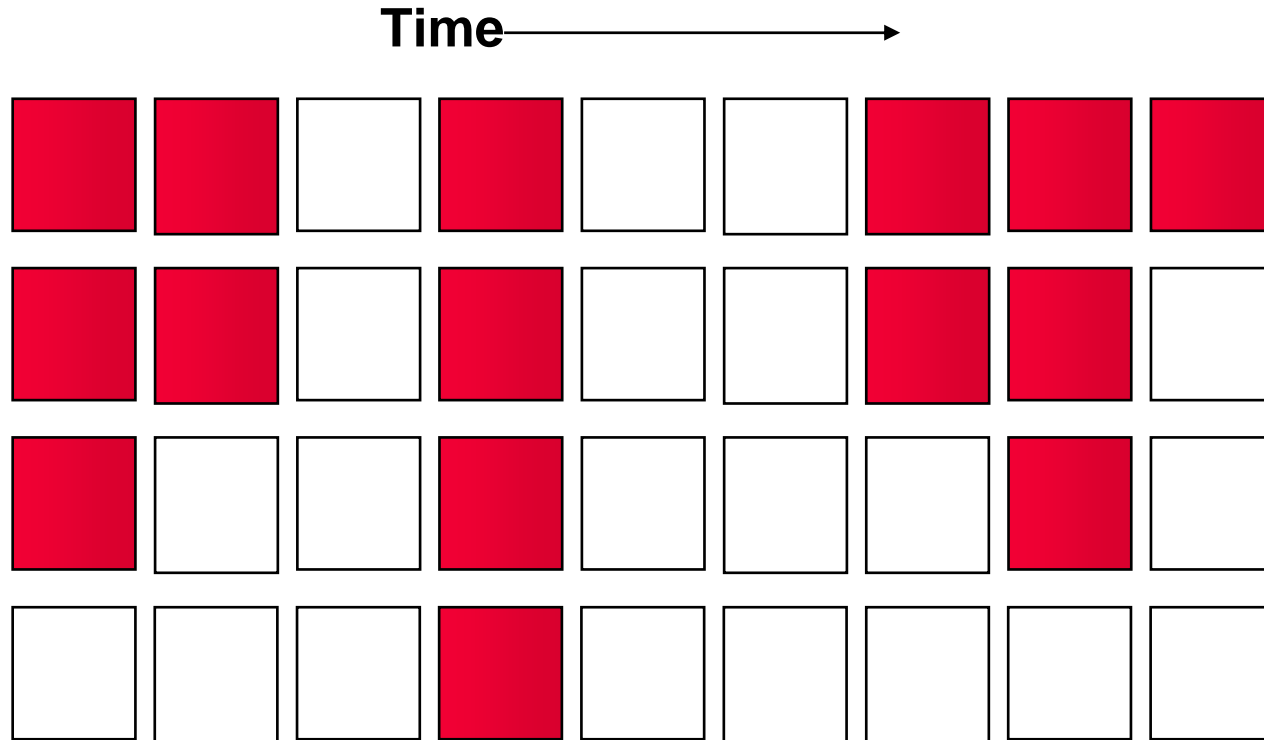
- ❑ **Share resources**
 - **Maximize the resource utilization at finer granularity**

Single Instruction Issue



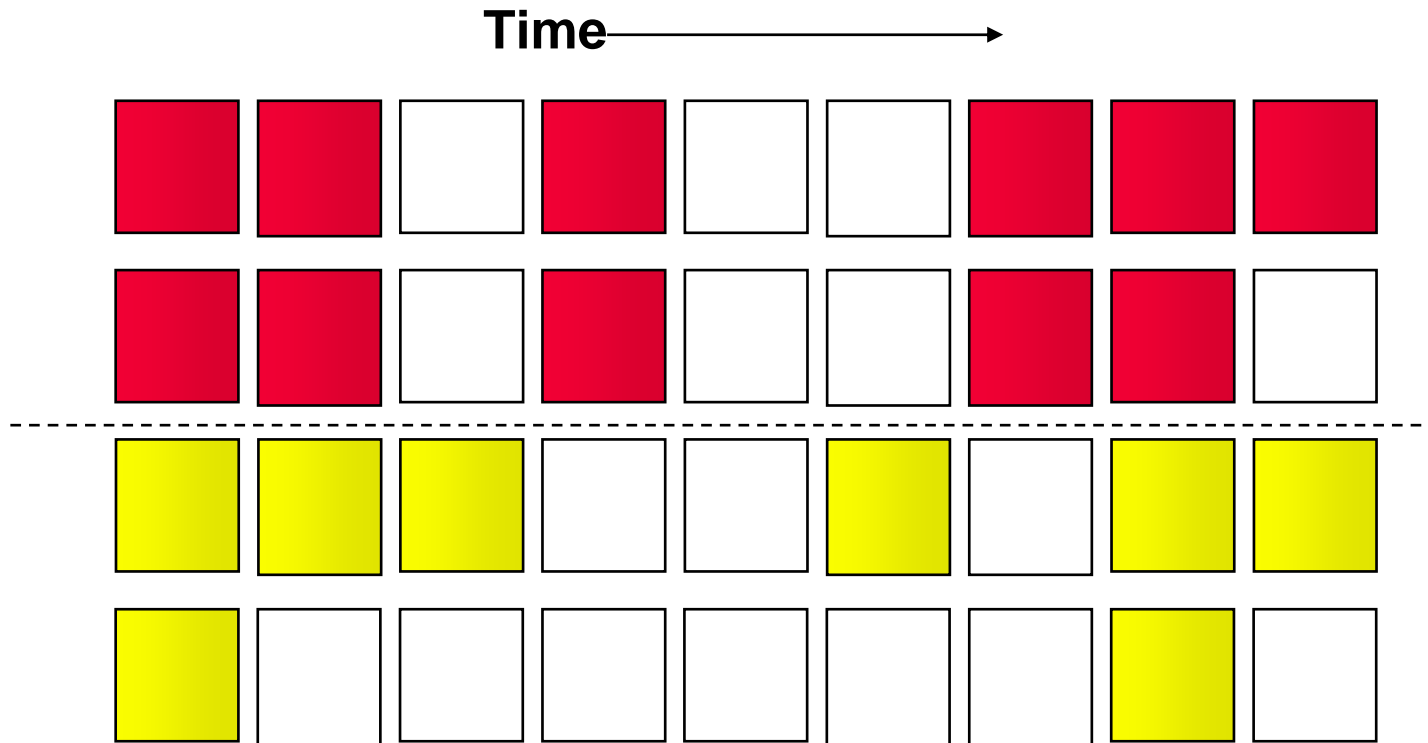
Reduced function unit utilization due to dependencies

Superscalar Issue



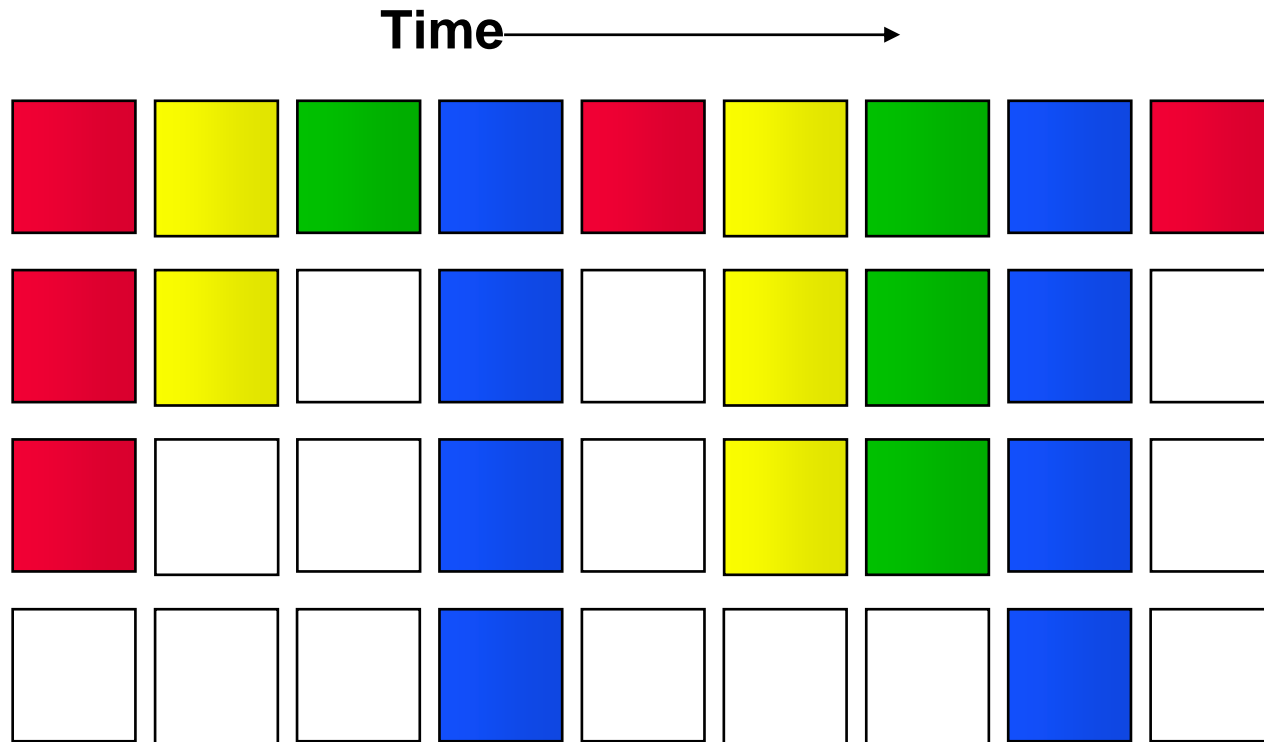
Superscalar leads to more performance, but lower utilization

Chip Multiprocessor (CMP)



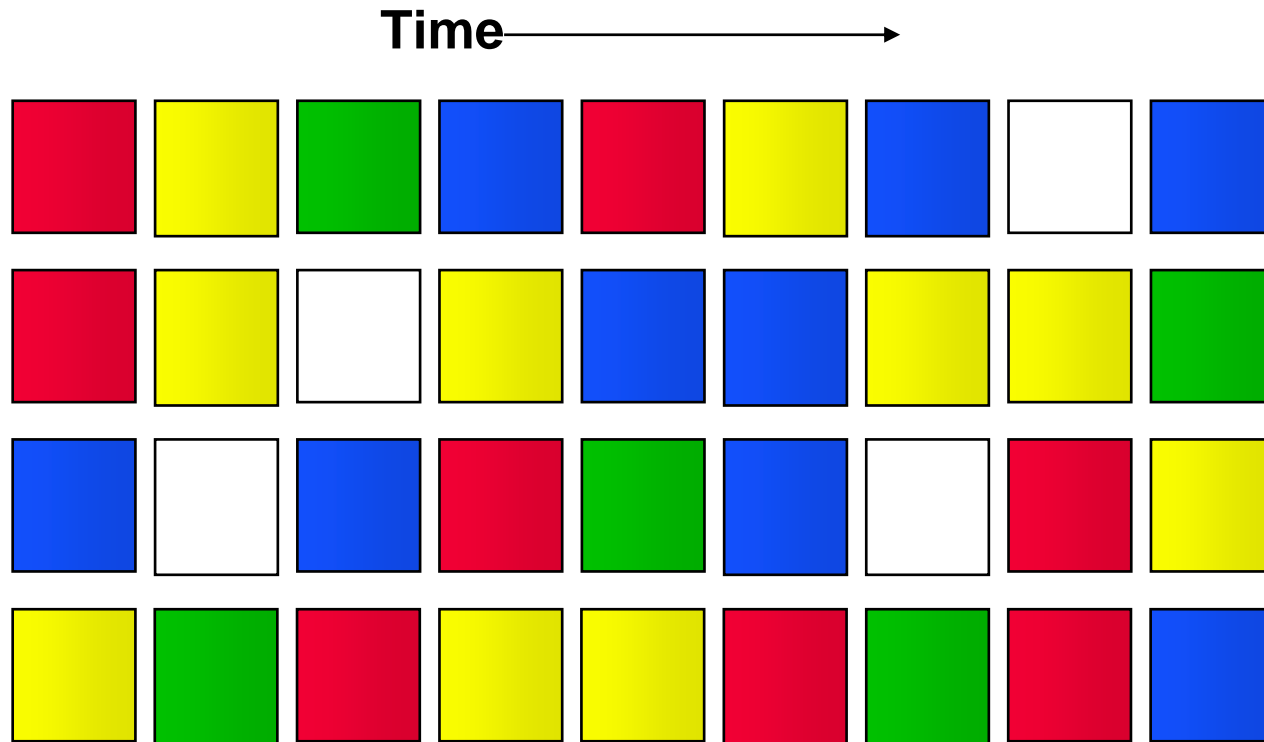
Limited utilization when only running one thread

Fine Grained Multithreading



Intra-thread dependencies still limit performance

Simultaneous Multithreading

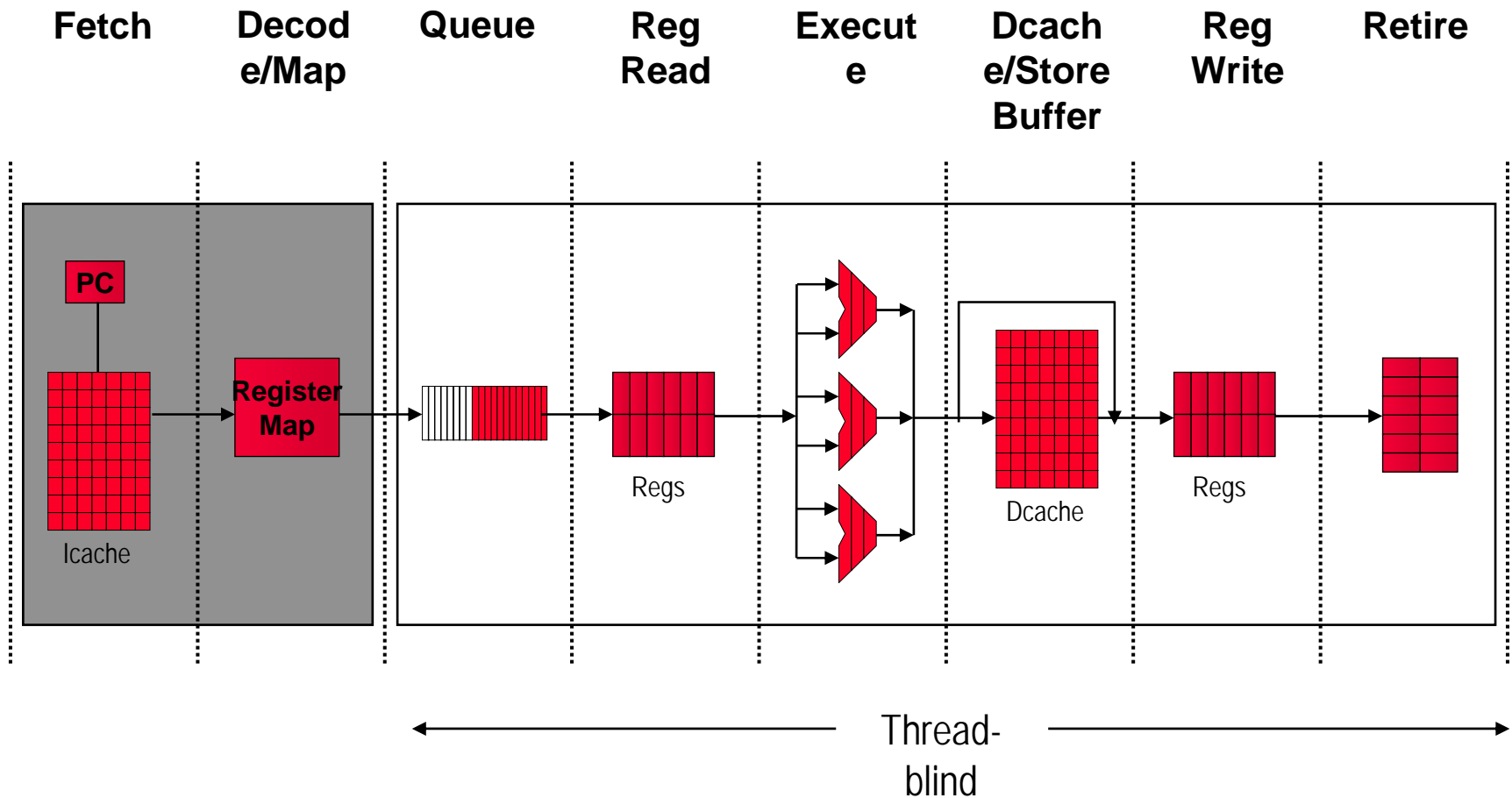


Maximum utilization of function units by independent operations

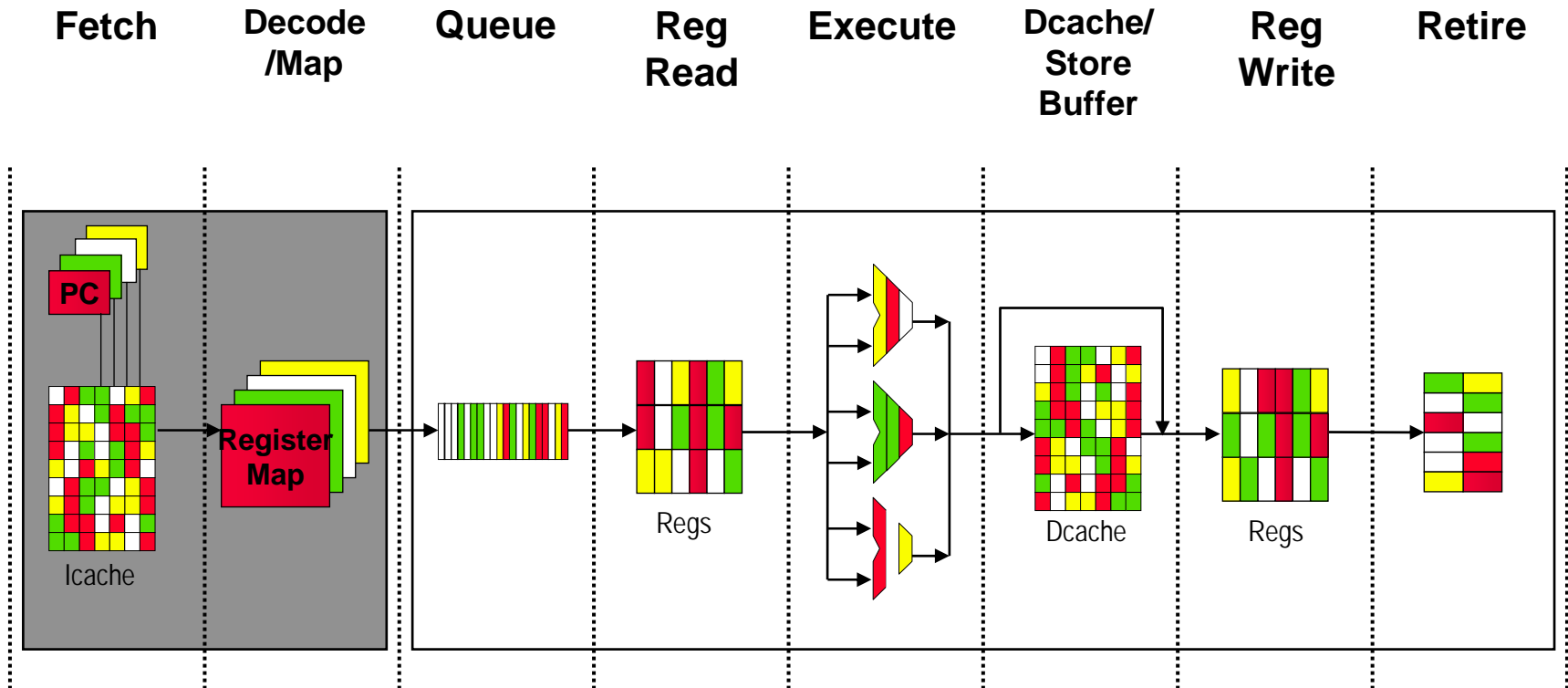
SMT Microarchitecture Changes

- ❑ **Multiple PCs**
- ❑ **Control to decide how to fetch from**
- ❑ **Careful handling with branches**
 - **Thread id with BTB**
 - **Thread flushing on mis-prediction**
- ❑ **Larger register file**
 - **More things outstanding**

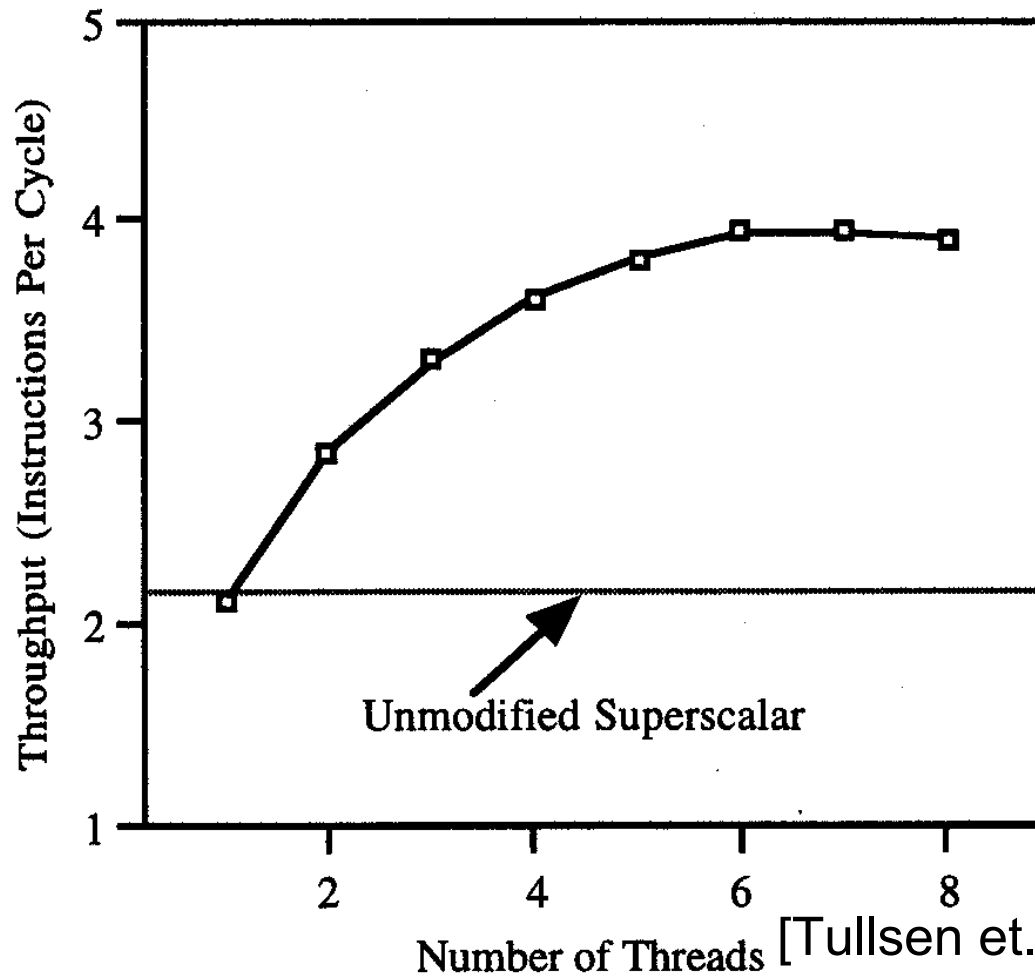
Basic Out-of-order Pipeline



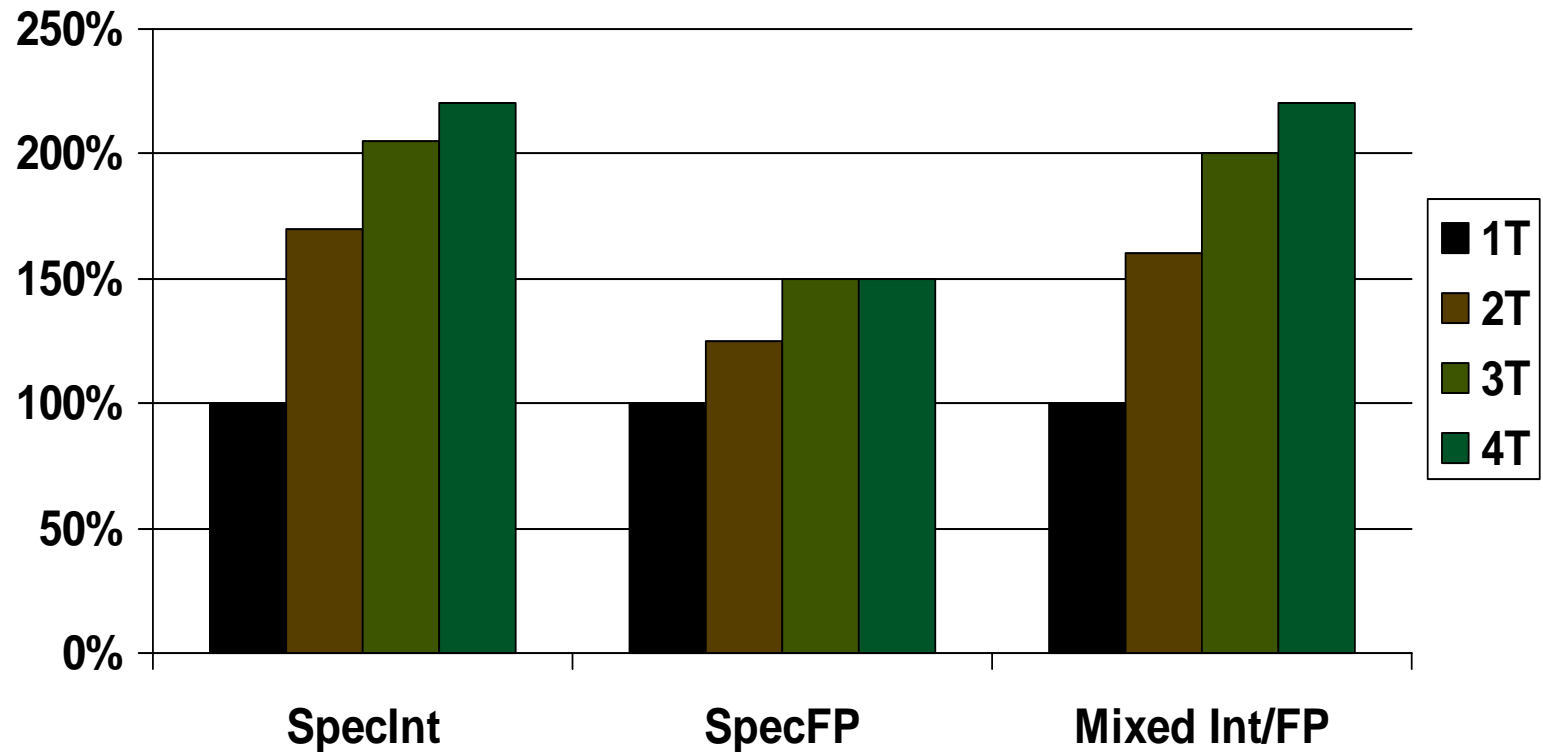
SMT Pipeline



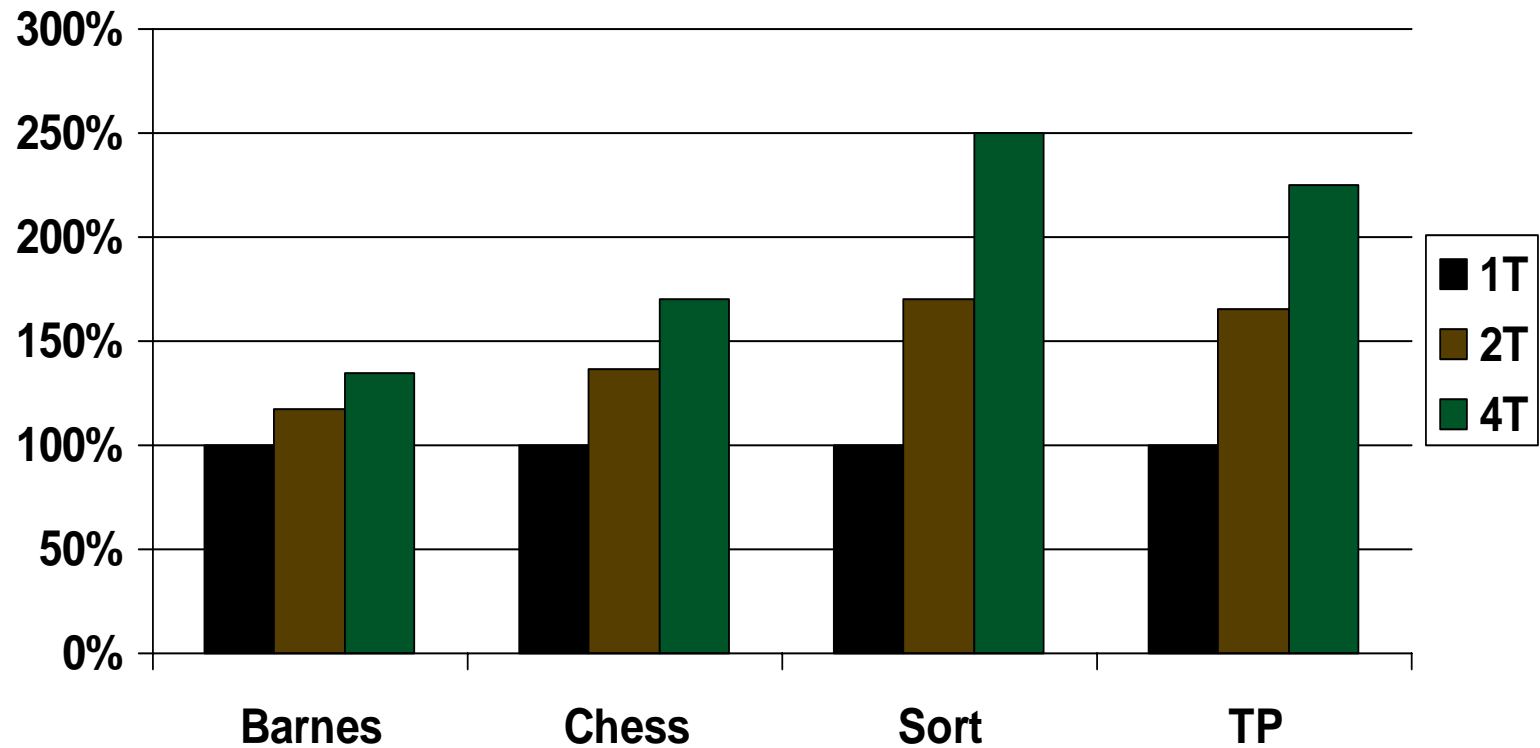
Performance



Multiprogrammed workload



Multithreaded Applications



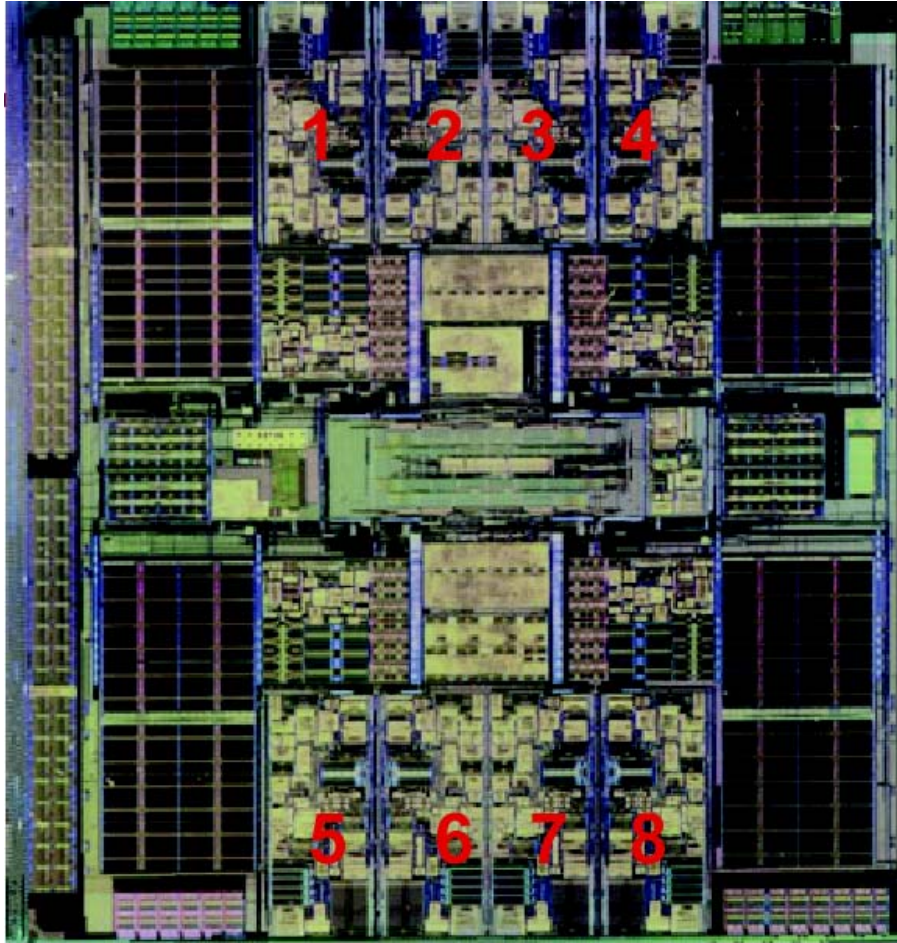
Inter-thread Cache Interference

- ❑ **Threads share the cache, so more threads, lower hit-rate.**
- ❑ **Two reasons why this is not a significant problem:**
 - **The L1 Cache miss can almost be entirely covered by the set-associative and larger L2 cache.**
 - **Out-of-order execution, write buffering and the use of multiple threads allow SMT to hide the small increases of additional memory latency.**
- ❑ **0.1% speed up without inter-thread cache miss.**

Interference in Branch Prediction Hardware

- ❑ Since all threads share the prediction hardware, it will experience inter-thread interference.
- ❑ This effect is ignorable since: the speedup out-weighted the additional latencies
- ❑ From 1 to 8 threads, branch and jump misprediction rates range from 2.0%-2.8% (branch) 0.0%-0.1% (jump)

Sun Niagara



- ❑ Instruction cache
 - 16kB, 4-way set associative, 32B line size
- ❑ L1 Data cache
 - 8kB, 4-way set associative, 16B line size
 - Write-through cache, write-around on miss
- ❑ L2 cache
 - 3 MB, 12-way set associative, 64B line size
 - Write-through
 - 4-way banked by line
- ❑ Coherency
 - Data cache lines have 2 state: valid or invalid
 - Data cache is write through → no modified state
 - Caches kept coherent by tracking lines in directories in L2