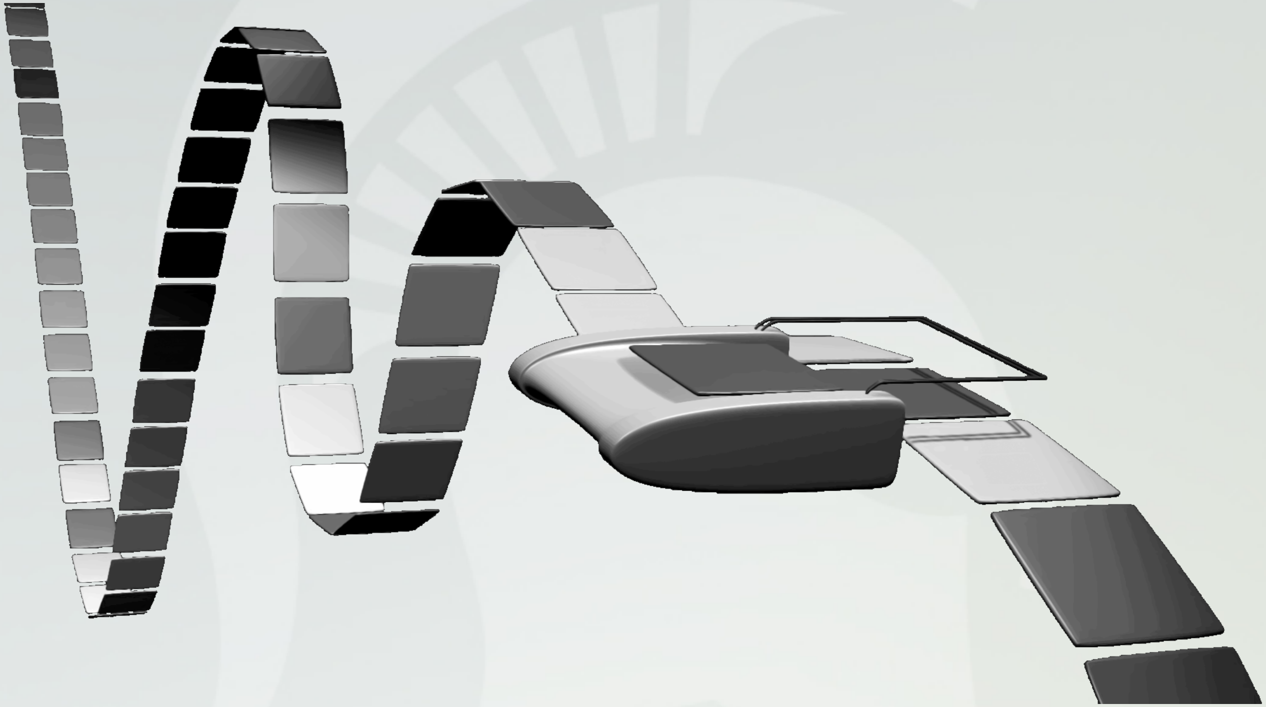
CSE 891 - Section 1: Parallel Computing - Fundamentals and Applications

Fall 2014 - Lecture 2: Parallel Computer Architectures (1)

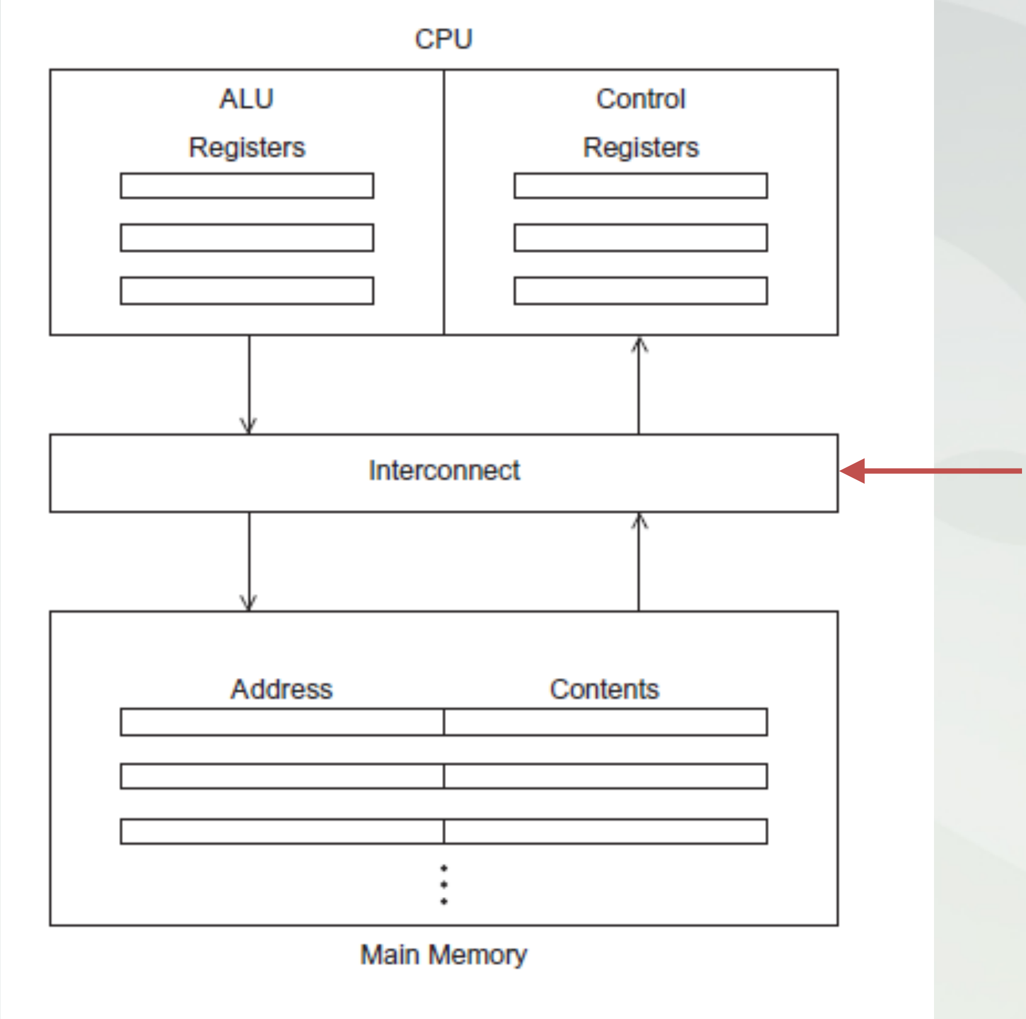
H. Metin Aktulga
hma@cse.msu.edu
517-355-1646

Turing Machine (1936)



- A hypothetical device that manipulates symbols on a strip of tape according to a table of rules [Wikipedia].

The von Neumann Architecture (1945)



von Neumann Bottleneck
The rate at which one can access instructions and data over the interconnect (memory latency & bandwidth) is a limiting factor.

in 2010:
latency ~ 100 CPU inst.
bytes/flop ~ 1/6

Modifications to the von Neumann Architecture: Caches

- **Cache:** a collection of memory addresses that is physically close to the CPU and can be accessed quickly.
- Limited space on silicon die, so limited cache size
- Exploit two kinds of locality to make best use:
  - Spatial locality
  - Temporal locality
- After accessing one memory location, a program will typically access a nearby location in the near future.

Caches: An example

```
float z[1000];
…
sum = 0.0;
for (i = 0; i < 1000; i++)
   sum += z[i];
```

- z[0] through z[999] are stored contiguously in memory
- *float* is 4 bytes
- typical cache line length today is 64 bytes
- when z[0] is accessed, z[1] through z[15] are also fetched into L1 cache

Caches: A slightly more complex one…

```
float z[1000][1000];
…
sum = 0.0;
for (i = 0; i < 1000; i++)
    for (j = 0; j < 1000; j++)
        sum += z[i][j];
```

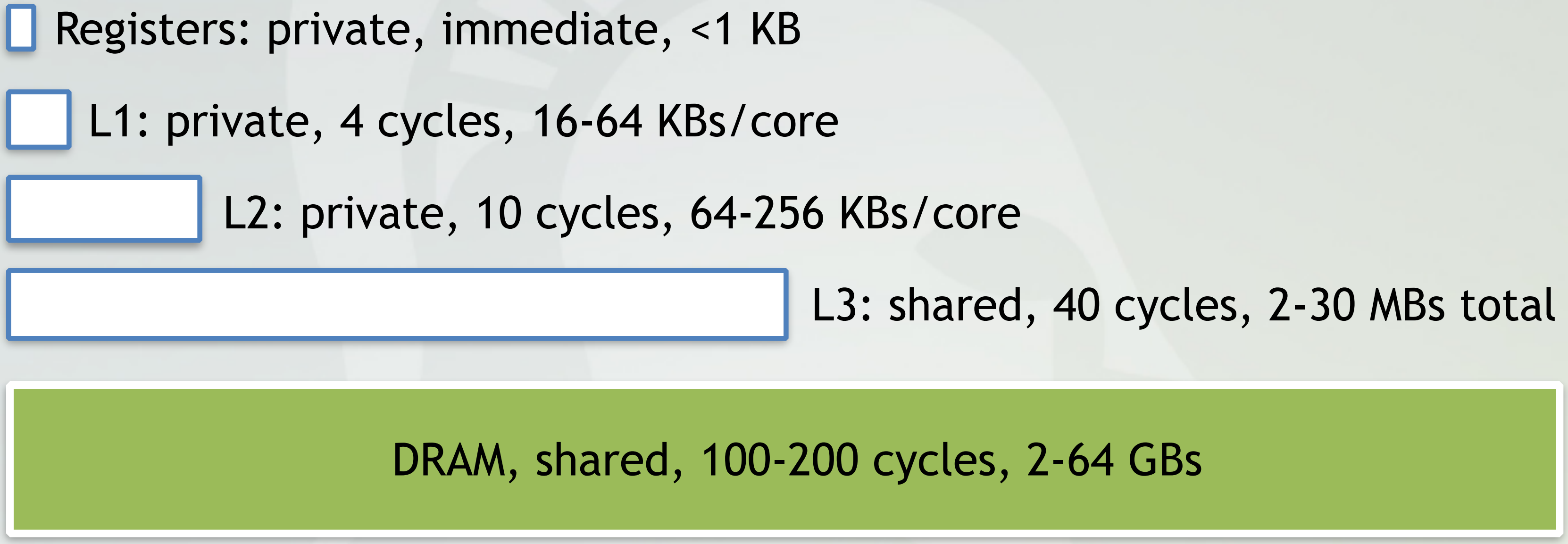
OR?

```
float z[1000][1000];
…
sum = 0.0;
for (j = 0; j < 1000; j++)
    for (i = 0; i < 1000; i++)
        sum += z[i][j];
```

- Do both codes do the same thing?
- Would they be any different in terms of performance?
- Which one would you choose? Why?
- What if you were programming in Fortran? or Matlab?

Different cache types

- Cache sizes & access times: some numbers



Registers: private, immediate, <1 KB

L1: private, 4 cycles, 16-64 KBs/core

L2: private, 10 cycles, 64-256 KBs/core

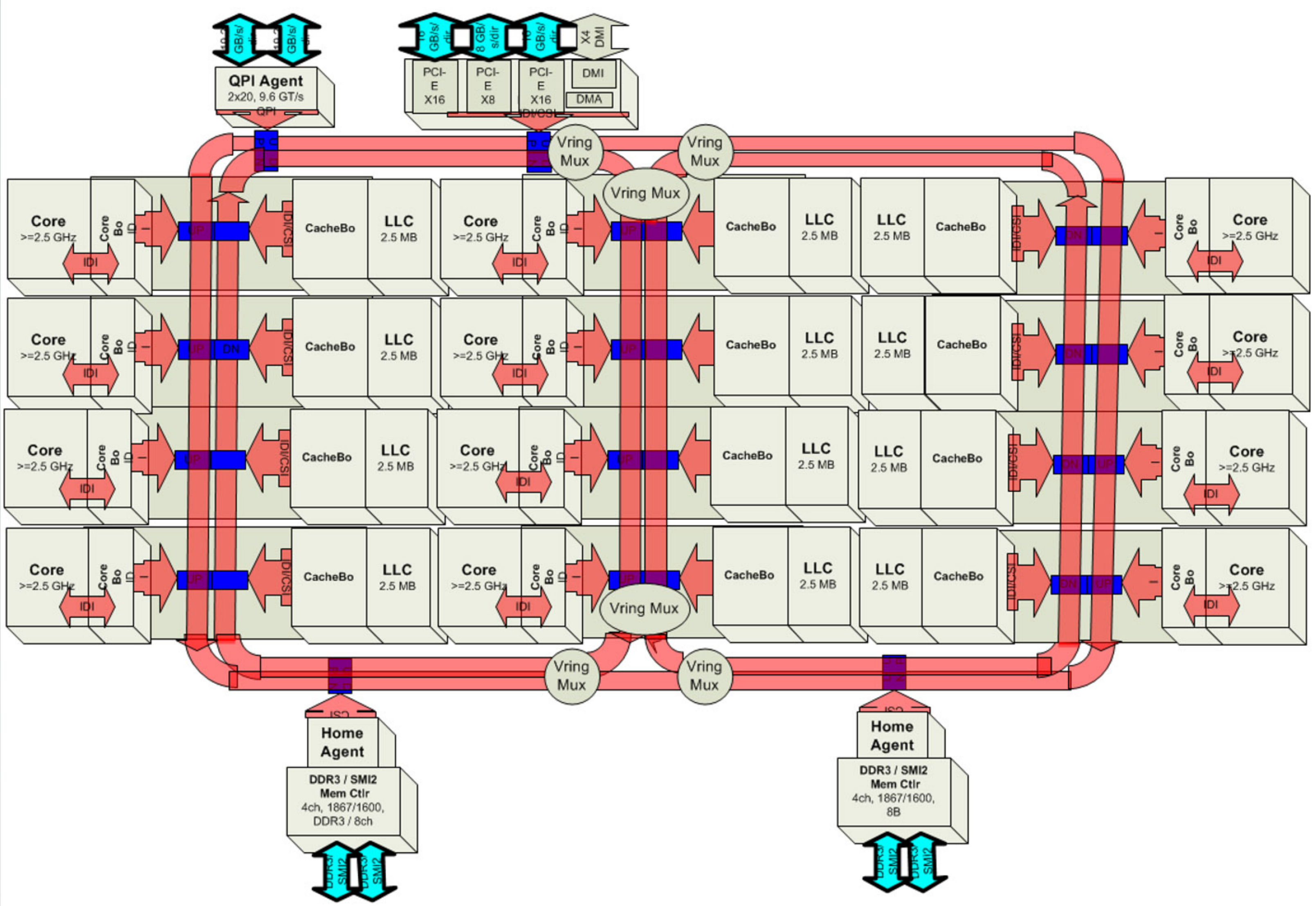
L3: shared, 40 cycles, 2-30 MBs total

DRAM, shared, 100-200 cycles, 2-64 GBs

Caches: Other considerations

- Cache mappings: full, direct, n-way set associative
- Cache coherency protocols to ensure data consistency among different cache locations
- Cache-memory consistency: write-through vs. write-back

How does a modern CPU look today?



Seamless parallelism mechanisms

- **Bit-level parallelism:** Instruction set architectures (ISA) has evolved from 4 bits to 64 bits

- **Instruction Level Parallelism (ILP)**
  - Pipelining, Multiple issue, Speculation

Instruction Level Parallelism (ILP)

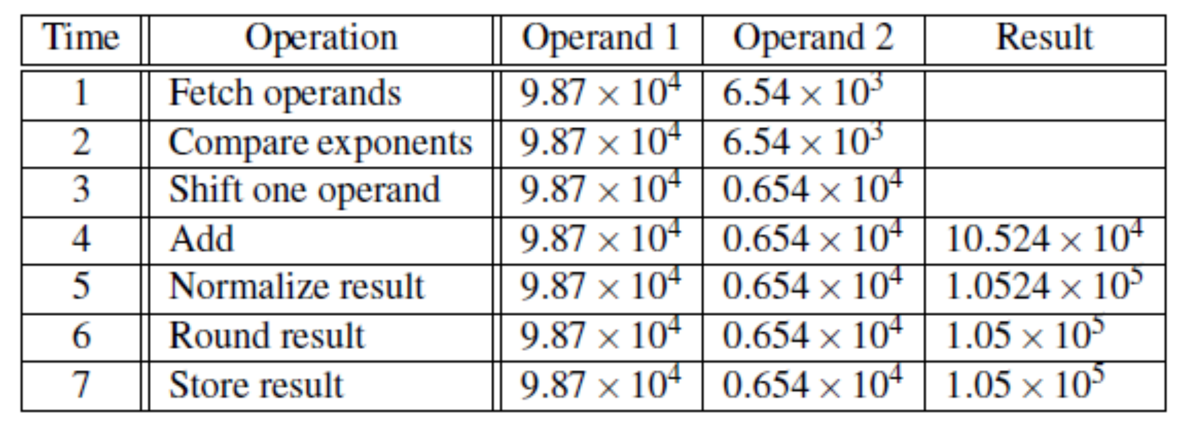
- Attempts to improve performance by having multiple functional units simultaneously execute instructions.
- Two main approaches used in all modern CPUs
  - Pipelining
  - Multiple Issue
- Seamless to the programmer
  - Hardware automatically exploits available parallelism

Pipelining

- Break instructions into smaller units and overlap their executions

For example:

Add the floating point numbers 9.87×10^4 and 6.54×10^3



Time	Operation	Operand 1	Operand 2	Result
1	Fetch operands	9.87×10^{4}	6.54×10^{3}	
2	Compare exponents	9.87×10^{4}	6.54×10^{3}	
3	Shift one operand	9.87×10^{4}	0.654×10^{4}	
4	Add	9.87×10^{4}	0.654×10^{4}	10.524×10^{4}
5	Normalize result	9.87×10^{4}	0.654×10^{4}	1.0524×10^{5}
6	Round result	9.87×10^{4}	0.654×10^{4}	1.05×10^{5}
7	Store result	9.87×10^{4}	0.654×10^{4}	1.05×10^{5}

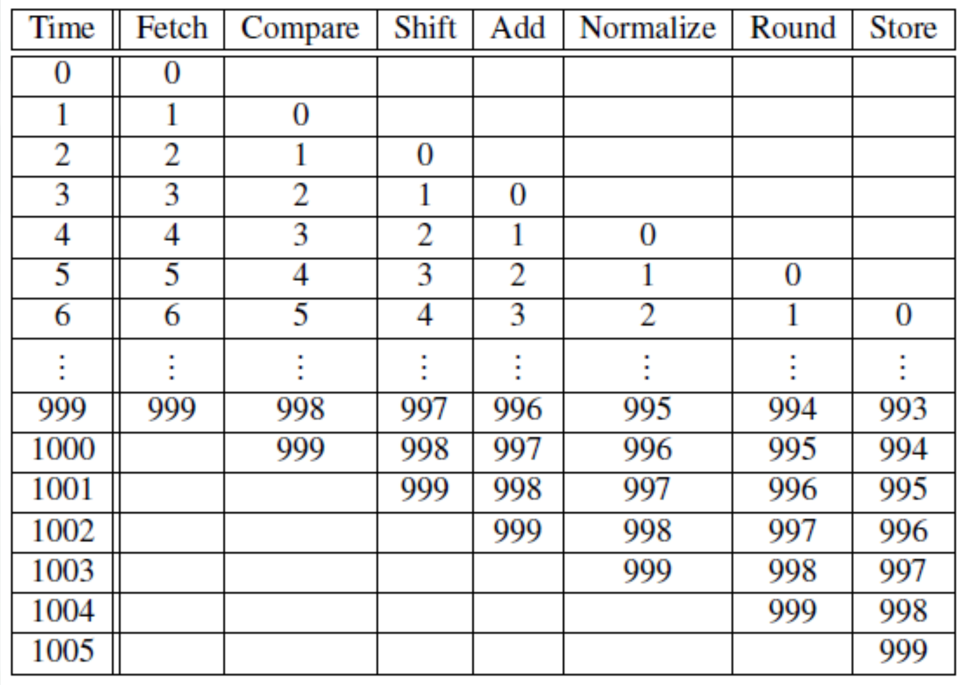
Pipelining: Example

```
float x[1000], y[1000], z[1000];
…
for (i = 0; i < 1000; i++)
   z[i] = x[i] + y[i];
```

- Assume x, y and z arrays reside in L1 cache already.
- Assume each operation takes 10^{-9} seconds - then what is the GHz rate of this CPU?
- Without pipelining, how long would the above loop take?

Pipelining: Example

Table 2.3: Pipelined Addition. Numbers in the table are subscripts of operands/ results.



Time	Fetch	Compare	Shift	Add	Normalize	Round	Store
0	0						
1	1	0					
2	2	1	0				
3	3	2	1	0			
4	4	3	2	1	0		
5	5	4	3	2	1	0	
6	6	5	4	3	2	1	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
999	999	998	997	996	995	994	993
1000		999	998	997	996	995	994
1001			999	998	997	996	995
1002				999	998	997	996
1003					999	998	997
1004						999	998
1005							999

- With pipelining, the same loop would take 1006 ns!

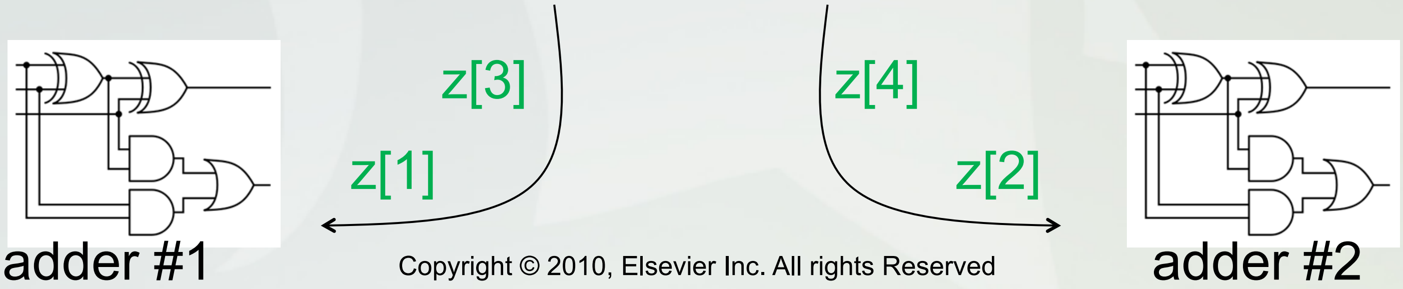
Pipelining in the Intel Core Microarchitecture

- Has 14- to 19-stage instruction pipeline
- Not all instructions fill the entire pipeline
- Cache hits are shorter, misses are longer

Multiple issue

- Multiple issue processors replicate functional units (multiple pipelines) and try to execute different instructions in parallel.
- **Example:**

```
for (i = 0; i < 1000; i++)
    z[i] = x[i] + y[i];
```



- Static vs. dynamic (superscalar) multiple issue

Speculation

- A certain type of instruction causes the pipeline to be stalled. Which one and how to solve this issue?

Branches:

```
z = x + y ;
if ( z > 0)
    w = x ;
else
    w = y ;
```

- Make a guess, execute along the guessed branch.
  - Commit when it is certain that the guess was correct
  - Rollback if the guess turns out to be wrong
  - Branch prediction table to make better guesses