

Advanced Optimization Techniques

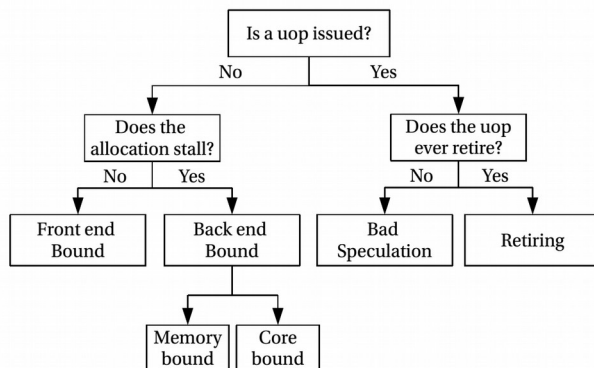
ICTP Trieste 2014

Dr. Christopher Dahnken

Intel GmbH

Outline

Method



Code

```

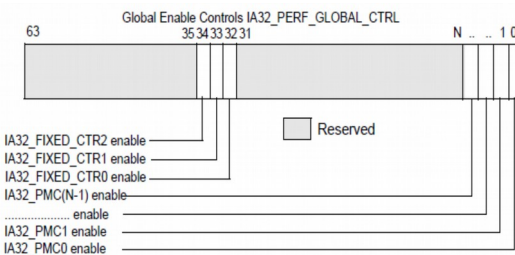
!$OMP SECTION
if(iblock.lt.(nblocks)) then
  nexti=m_of_i(iblock+1)
  nextj=n_of_i(iblock+1)
  nextk=k_of_i(iblock+1)

  next_buffsize_m=bufferize(ms,bm,nexti)
  next_index_m=(nexti-1)*bm+1

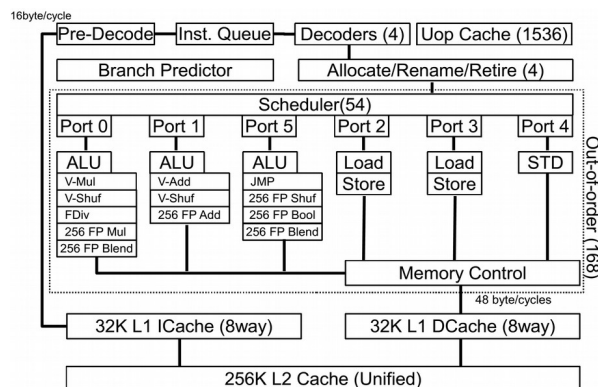
  next_buffsize_n=bufferize(ns,bn,nextj)
  next_index_n=(nextj-1)*bn+1

  next_buffsize_k=bufferize(ks,bk,nextk)
  next_index_k=(nextk-1)*bk+1
  
```

Measurement



CPU



Legal Disclaimer & Optimization Notice

- INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.
- Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.
- Copyright © , Intel Corporation. All rights reserved. Intel, the Intel logo, Xeon, Xeon Phi, Core, VTune, and Cilk are trademarks of Intel Corporation in the U.S. and other countries.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Features of a modern CPU

Modern CPU design

- It is necessary to consider the underlying hardware if one wants to extract a maximum of performance from it
- Strength and also weaknesses need to be understood
- The most important features will not (or rarely) change – we will discuss them here

Some things upfront

- **Instruction set architecture:** Defines all programming related properties of a CPU, e.g. the instructions a processor can execute, their arguments, interrupts, registers, etc.
- **Instruction:** A function of the processor performing a certain task. Usually it has implicit or explicit arguments, such as registers. It is represented by a number (opcode), but normally use a more readable equivalent, a so-called mnemonic.
- **Register:** A very small and very fast piece of memory, directly accessible by the CPU. Actually, the (externally visible) number of registers is a constant of the ISA. On 64bit architecture, the standard registers are 64bit wide, but there are extensions to the ISA that have wider registers, such as AVX.

Some things upfront

- **Assembly language:** A programming “language” that represents a direct mapping of the machine instructions (numbers) to mnemonics, for example:

<u>Opcode:</u>	<u>Assembly:</u>
48 89 c1	mov rcx, rax

On Linux, you can see the assembly language of a program or object file by using

```
objdump -M intel-mnemonic -d <object file>
```

Here, we will often be concerned how the compiler translates some given code, so it is advisable to familiarize oneself with assembly language.

Modern CPU design

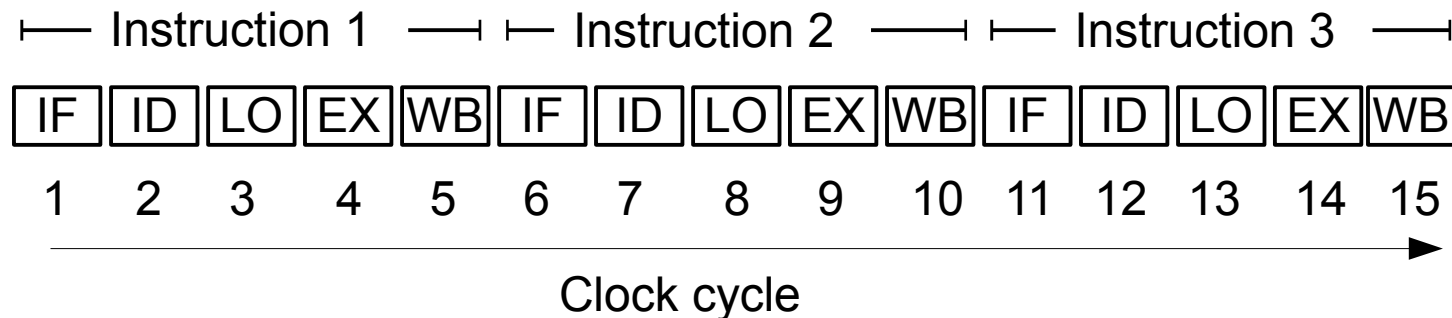
Important building blocks

- Pipelines
- Out-of-order execution
- Superscalarity
- SIMD execution
- Branch prediction
- Caches

Pipeline

- Consider a very simple processor operation:
 - Instruction fetch (IF)
 - Instruction decode (ID)
 - Load operands (LO)
 - Execute (EX)
 - Write back results (WB)

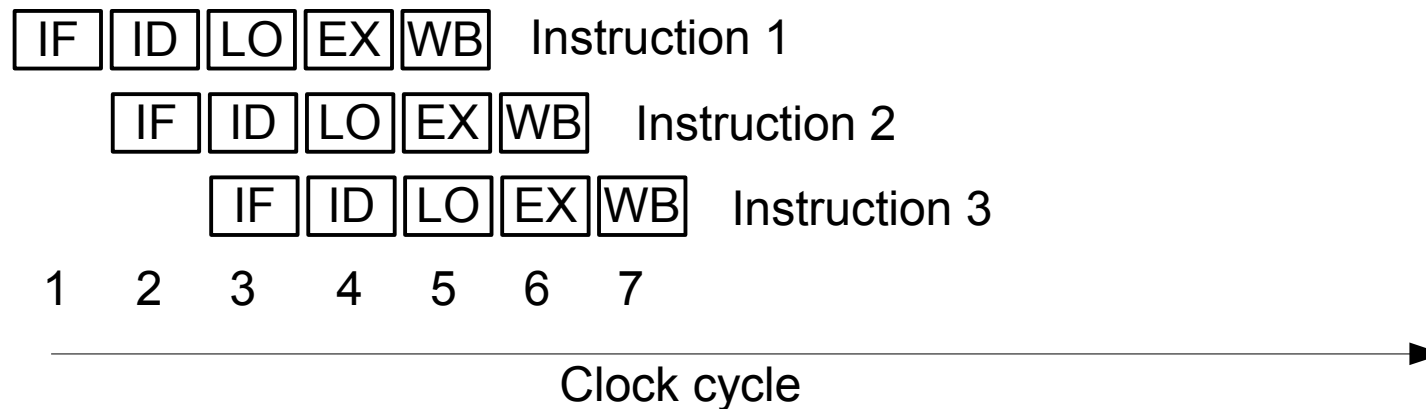
No pipeline



Pipeline

- In this model, 4/5 of the processor are idle while only 1/5 is working at every single time
- If we can overlap (pipeline) the operation, each step is (potentially) busy every cycle

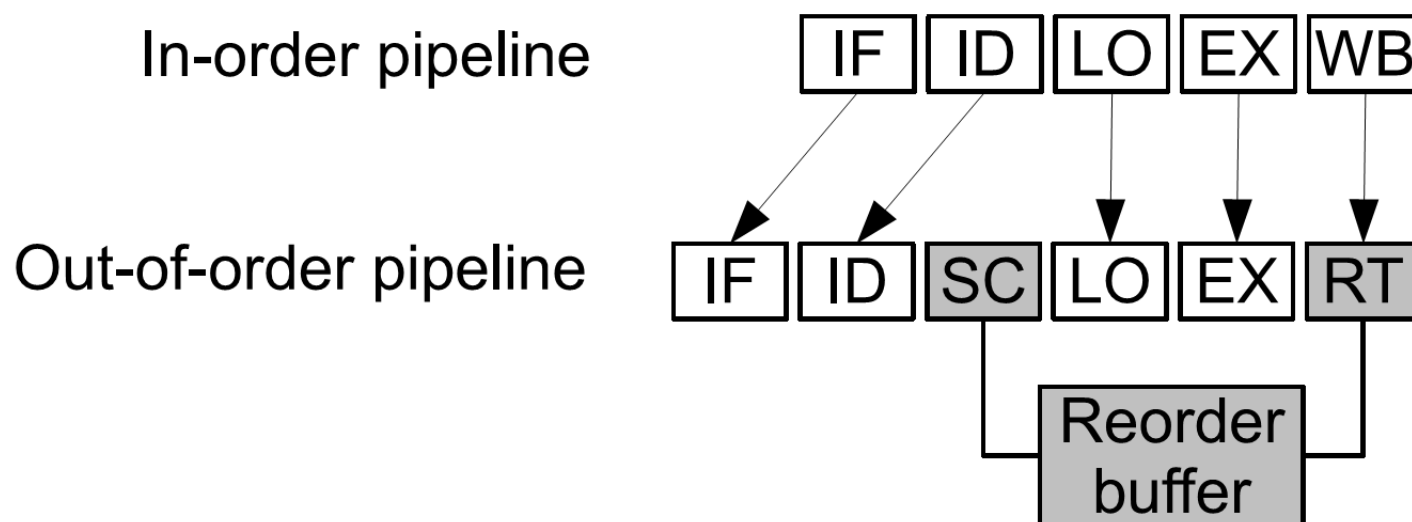
Pipeline



Pipeline

- A pipelined operation gives the processor a speedup of $\times N_{\text{stages}}$
- Today, we assume this performance to be normal
- Every disruption of the pipeline is perceived as a performance loss
- Main goal of tuning is to keep the pipeline busy
- There are many scenarios, that stop the pipeline:
Dependent instruction, conditional execution, etc

Out-of-order Pipeline

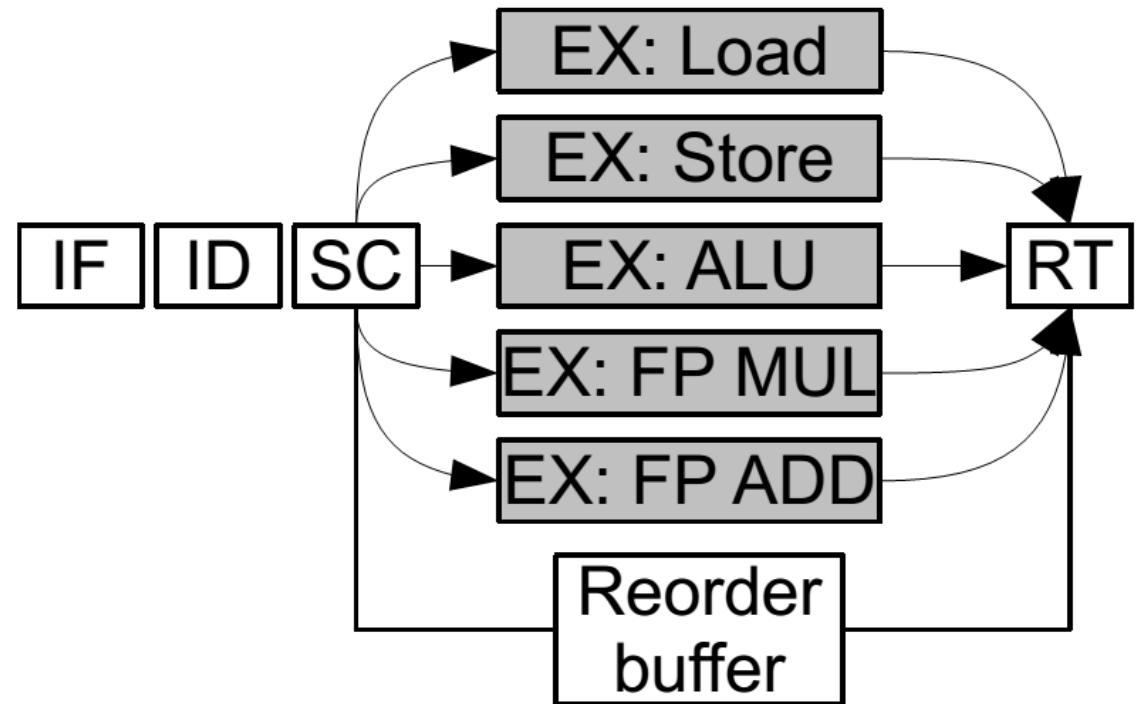


Out-of-order execution exploits the independence of instructions in a linear stream → while one instruction waits for dependencies, another one can execute. In the end, the original order needs to be restored.

We need more units: A “Scheduler” (SC) and a “Retirement unit” (RT)

Superscalarity

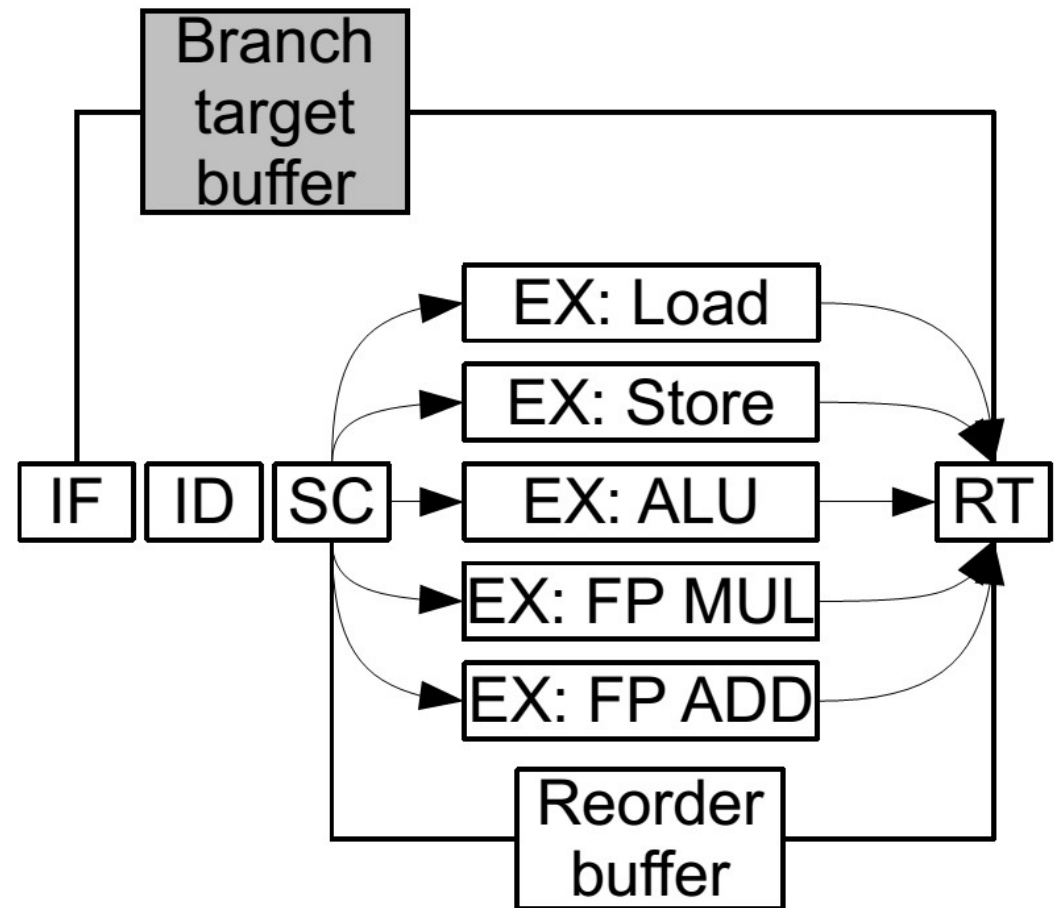
We have a scheduler already that takes care of issuing uops into the execution unit *independently*.



A straight forward development is the implementation of superscalar pipelines that can execute different instructions in parallel

Branch Prediction

A performance hazard of the out-of-order pipeline is the appearance of conditional branches. If the condition is not known at the time of the branch, the execution has to wait.



Branches

```
status=mySubroutine(...);  
if(status!=0) {  
    printf("alert!\n");  
    exit(0);  
}
```

If there is no branch prediction, the execution would halt at the if condition (or rather the corresponding assembly code)

Branches

Some language construction have implicit branches:

```
for(int i=0;i<length;i++) {  
    sum+=a[i];  
}
```

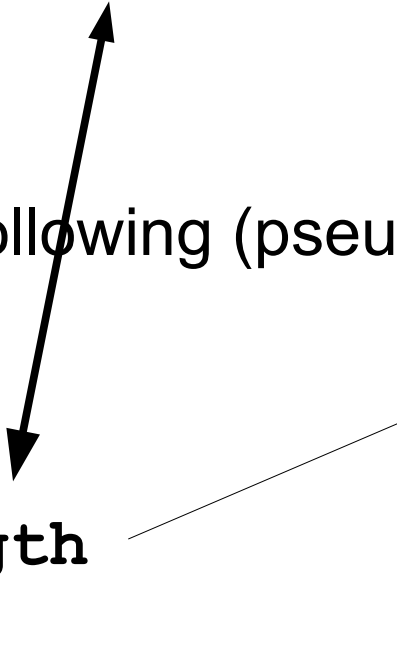
Corresponds to the following (pseudo) assembly:

```
loop1:
```

```
...
```

```
cmp eax, length
```

```
jle <loop1>
```



This will compare a counter register to length and store the status in a flag

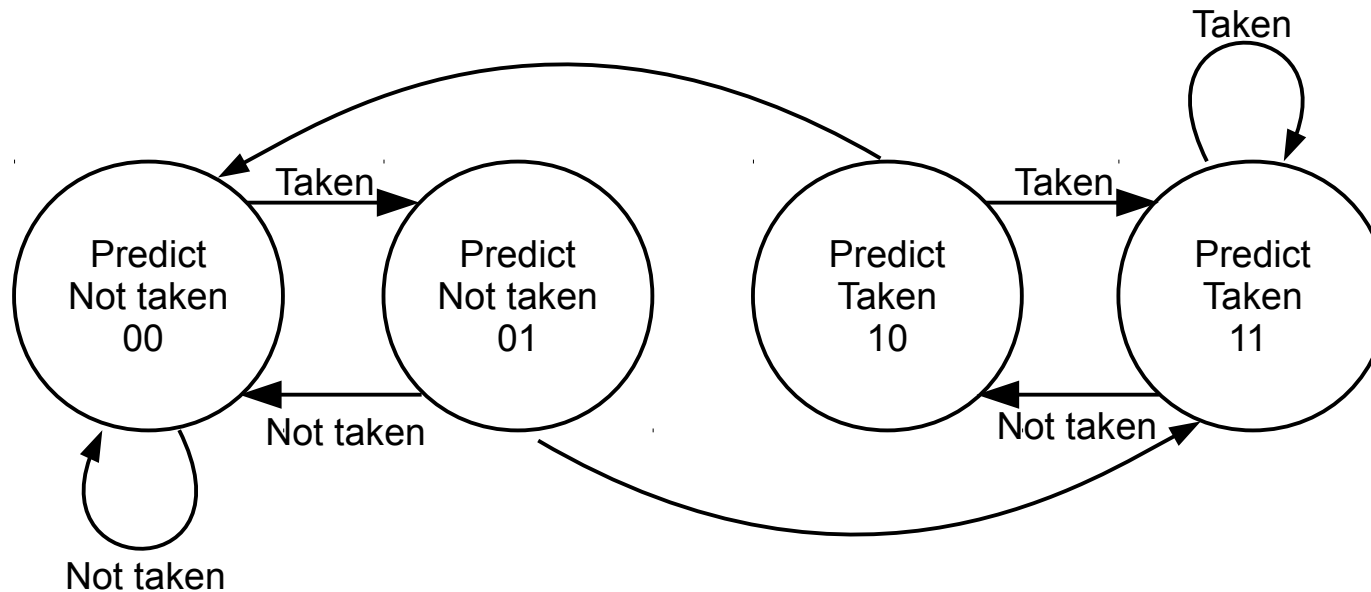
This will cause the instruction pointer to move to <loop1> when the comparison was less or equal

Dynamic Branch Prediction

- The *Branch Prediction Unit* (BPU) tracks the address of each branch along with a status in a structure called the *Branch Target Buffer* (BTB)
- Based on the status in the BTB, the execution will simply continue *speculatively*
- If the speculation we wrong, the compute results after the wrong prediction cannot be committed and the pipeline needs to be *flushed*

Dynamic Branch Prediction

- The most simple branch predictor would be a single bit (if I took the branch last time, I will take it again). A 2bit branch prediction would be:



- A real CPU has a much more complicated BPU

Static Branch Prediction

- Above branch prediction schemes are **dynamic**, i.e. they change based upon previously recorded data
- If we encounter a branch for the first time, we need a set of rules which define the default behavior
- These rules are the so-called **static branch prediction**

Static Branch Prediction

- A **forward** conditional branch (an if-statement) is predicted *not to be taken*
- A **backward** conditional branch (a loop) is predicted *to be taken*
- An **unconditional** branch (a call to or return from a subroutine) is predicted *to be taken*

SIMD Execution

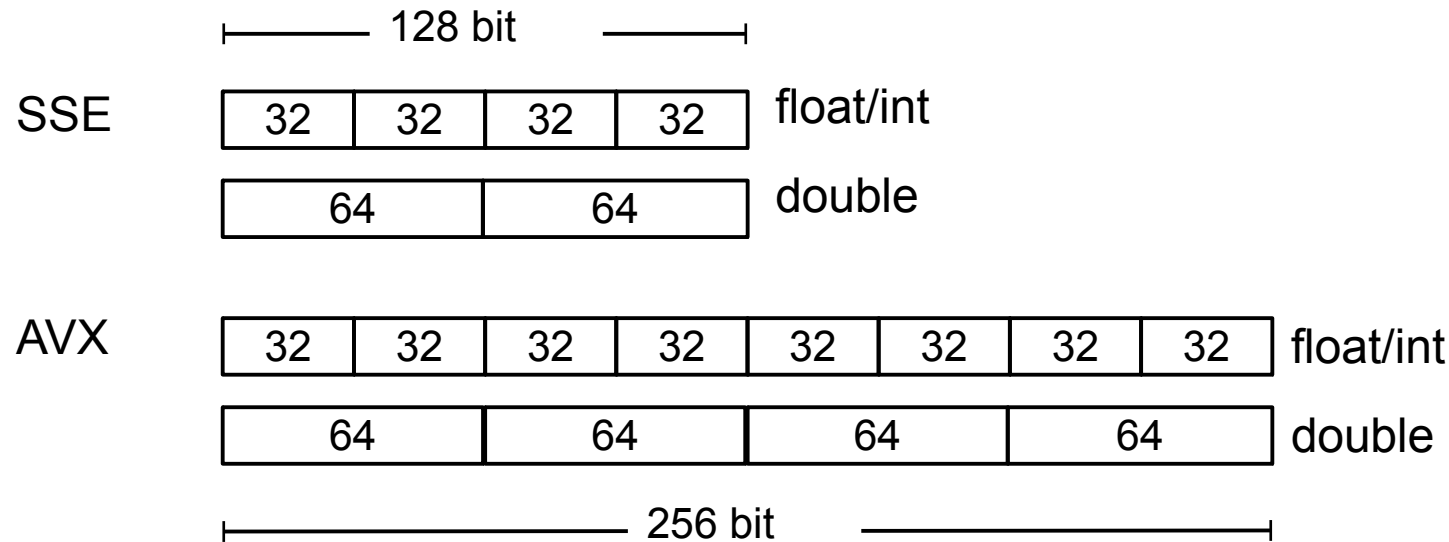
- Very often, code constructs execute the same operation on a large number of elements, exposing data level parallelism (DLP)
- It is a straight forward idea to exploit this parallelism by implementing the DLP in hardware
- Quite old, but very effective idea – CDC, Thinking Machines

```
for(int i=0;i<length;i++)  
    c[i]=a[i]+b[i];
```



```
for(int i=0;i<length;i+=4)  
    c[i]=add4atime(a[i],b[i]);
```

SIMD Execution



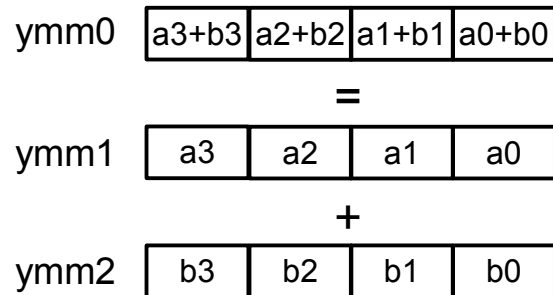
Hardware needs to implement:

- Registers (storage units) of required width
- Instructions that accept those registers as arguments

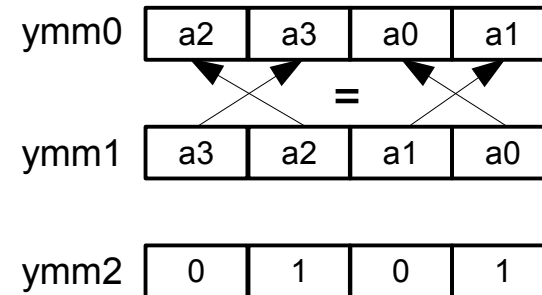
SIMD Execution

- AVX SIMD examples (we will look at the very details later)

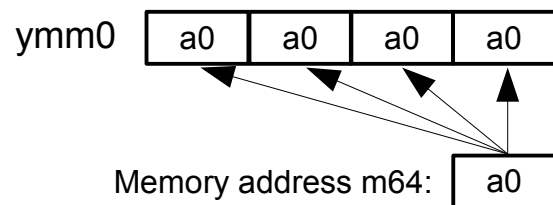
vaddpd ymm0,ymm1,ymm2



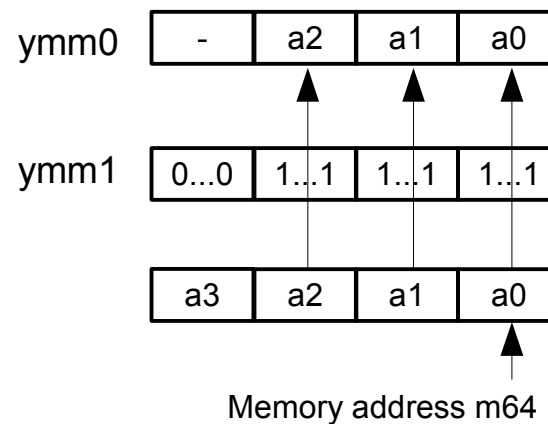
vpermilpd ymm0,ymm1,ymm2



vbroadcastsd ymm0,m64



vmaskmovpd ymm0,ymm1, m64



Caches and Cache Coherency

- Caches and cache coherency is a topic not *really* related to the core micro-architecture, but still important in the context of advanced performance tuning, so we'll discuss it here
- We assume that standard OS memory management has been introduced (pages, TLB, etc.)

Caches

- Performance of DRAM grows much slower than the performance of CPUs
- Cache: A small, fast and expensive bieve of storage built into the CPU
- Caches base on the principle of temporal locality: data you have used you are likely to use again in close future
- Often, more than one cache is present, which we then call a cache hierarchy

Caches

- A cache can only store *cache lines* which are 64byte of contiguous memory in our case
- There are three types of caches
 - Direct mapped caches
 - Fully associative caches
 - Set associative caches

Caches

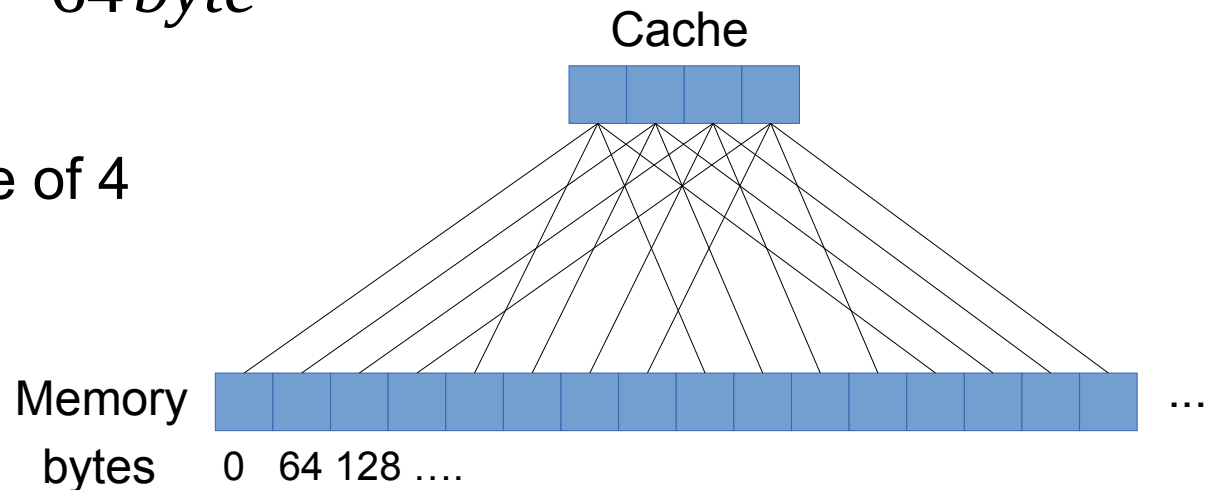
Direct mapped caches

- In a direct mapped cache, each memory address can only be stored in a particular cache line. One possible strategy would be

$$N_{line} = \frac{Address}{64\text{ byte}} \bmod N_{lines}$$

Example:

A direct mapped cache of 4 cache lines



Caches

Direct mapped caches

- Direct mapped caches are fast – there is no logic in them!
- But they are disposed to *cache thrashing*: Cache lines are evicted from the cache with high frequency. Think about a stride that is a multiple of a cache line!

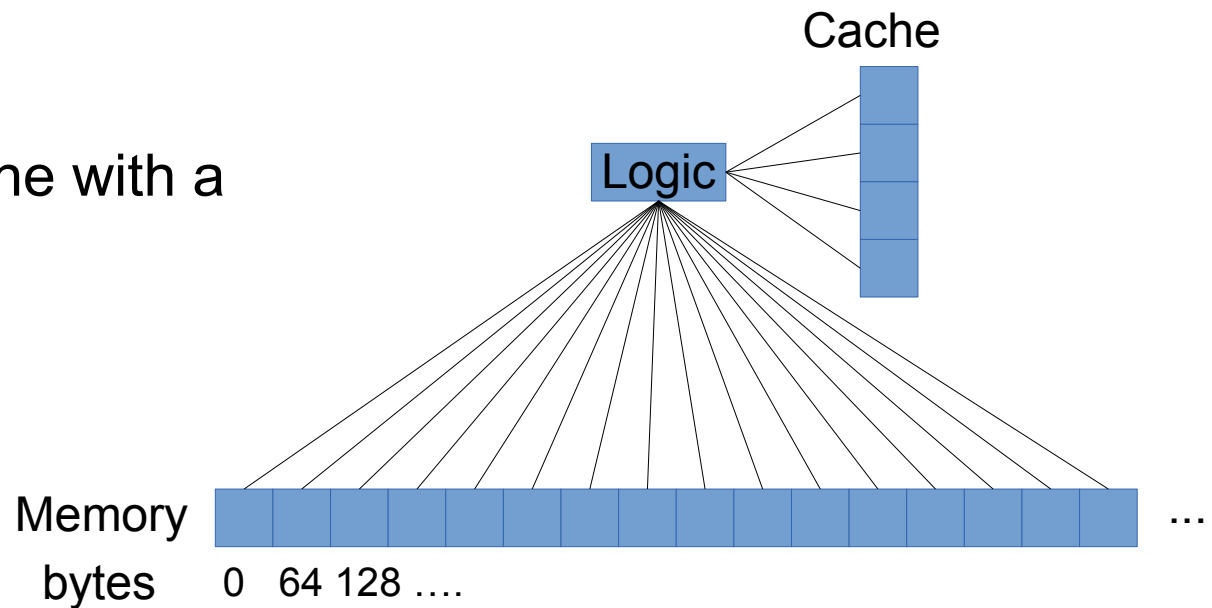
Caches

Fully associative caches

- A fully associative cache has a pool of cache lines (a so-called *set*) and a logic that stores requested cache lines not present in the cache. A common logic strategy is to evict the line that was Last Recently Used (LRU).

Example:

A fully associative cache with a set of 4 cache lines



Caches

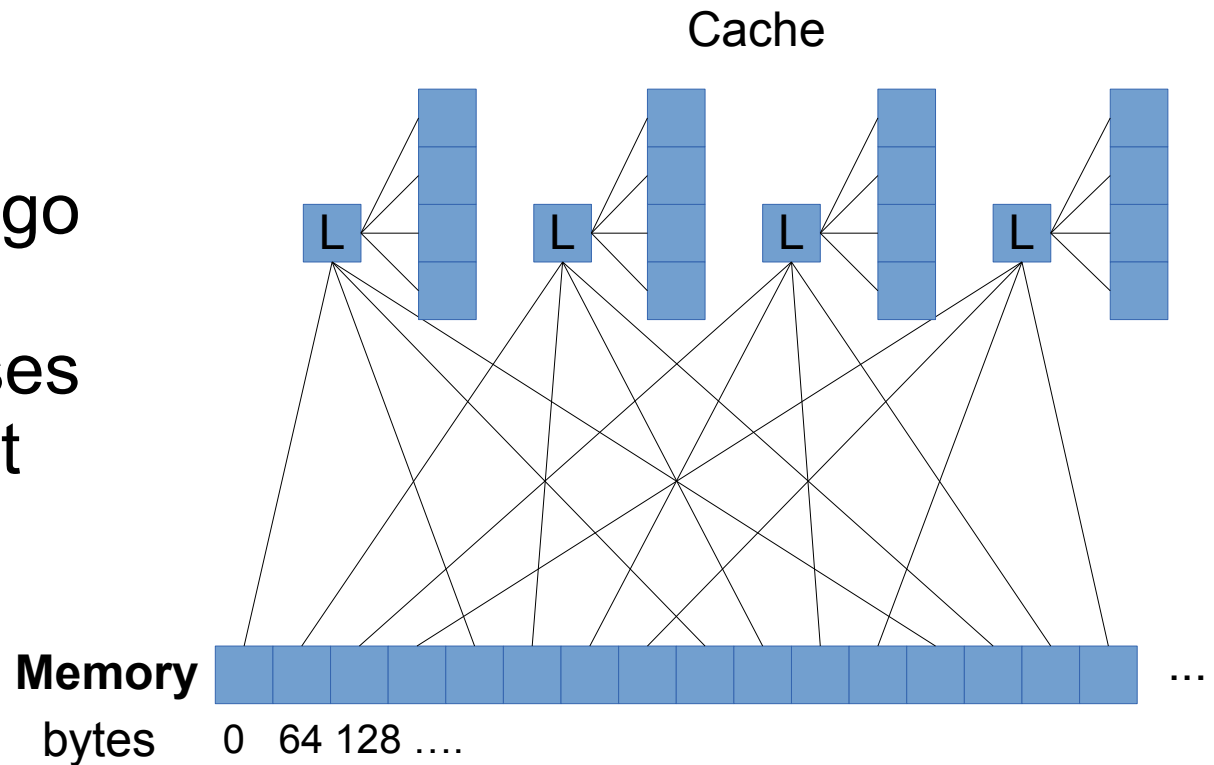
Fully associative caches

- Fully associative caches are comparatively slow since the logic needs to
 - Look up if a request cache line is present
 - If yes, deliver the data to the requesting instance
 - if not find the LRU entry, load the data from memory and deliver the data to the requesting instance.
- This also makes them expensive since the logic takes quite some space.
- The longer the set, the more expensive and slower it becomes

Caches

Set associative caches

- A set associative cache is a combination of direct mapped and fully associative cache strategies
- Each address in memory can only go to a particular set, but for all addresses mapped to this set the cache is fully associative

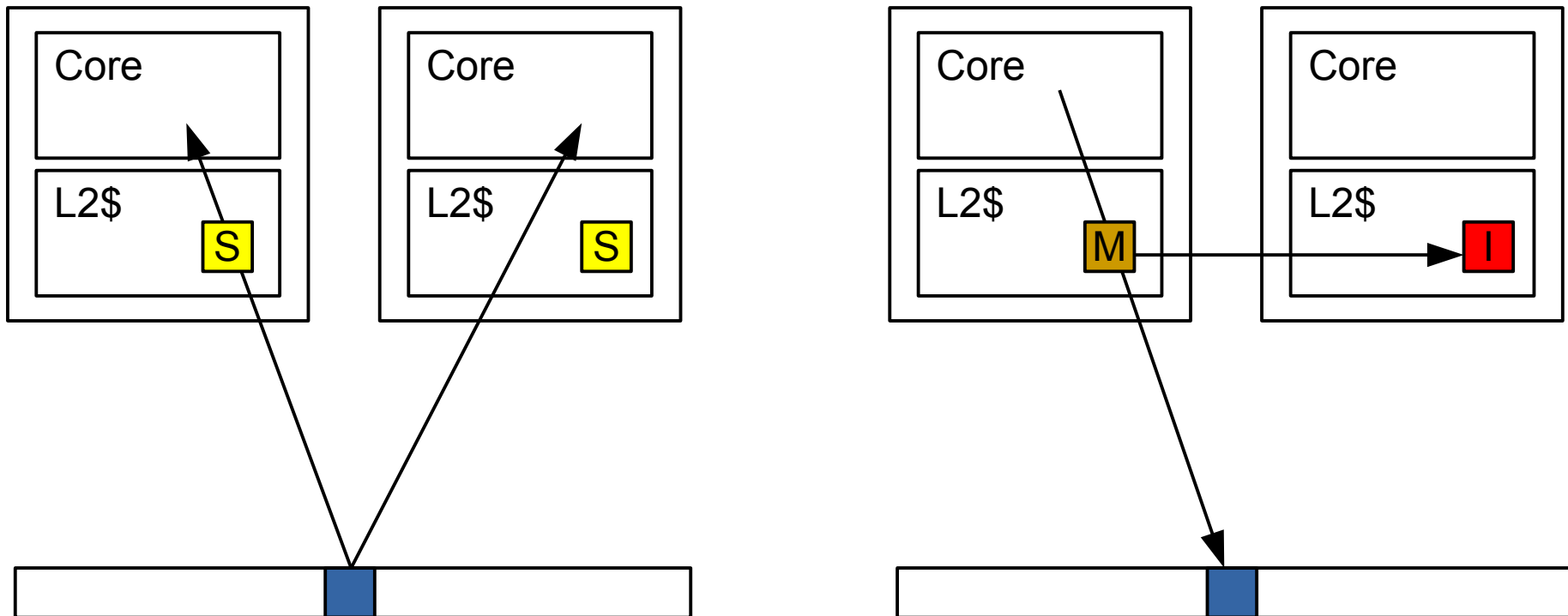


Caches

Set associative caches

- Set associative caches represent a reasonable compromise between fast direct mapped and slow fully associative caches
- In Sandy Bridge, we have
 - 32Kbyte Level1 cache, 8 way set associative
 - 256Kbyte Level2 cache, 8 way set associative

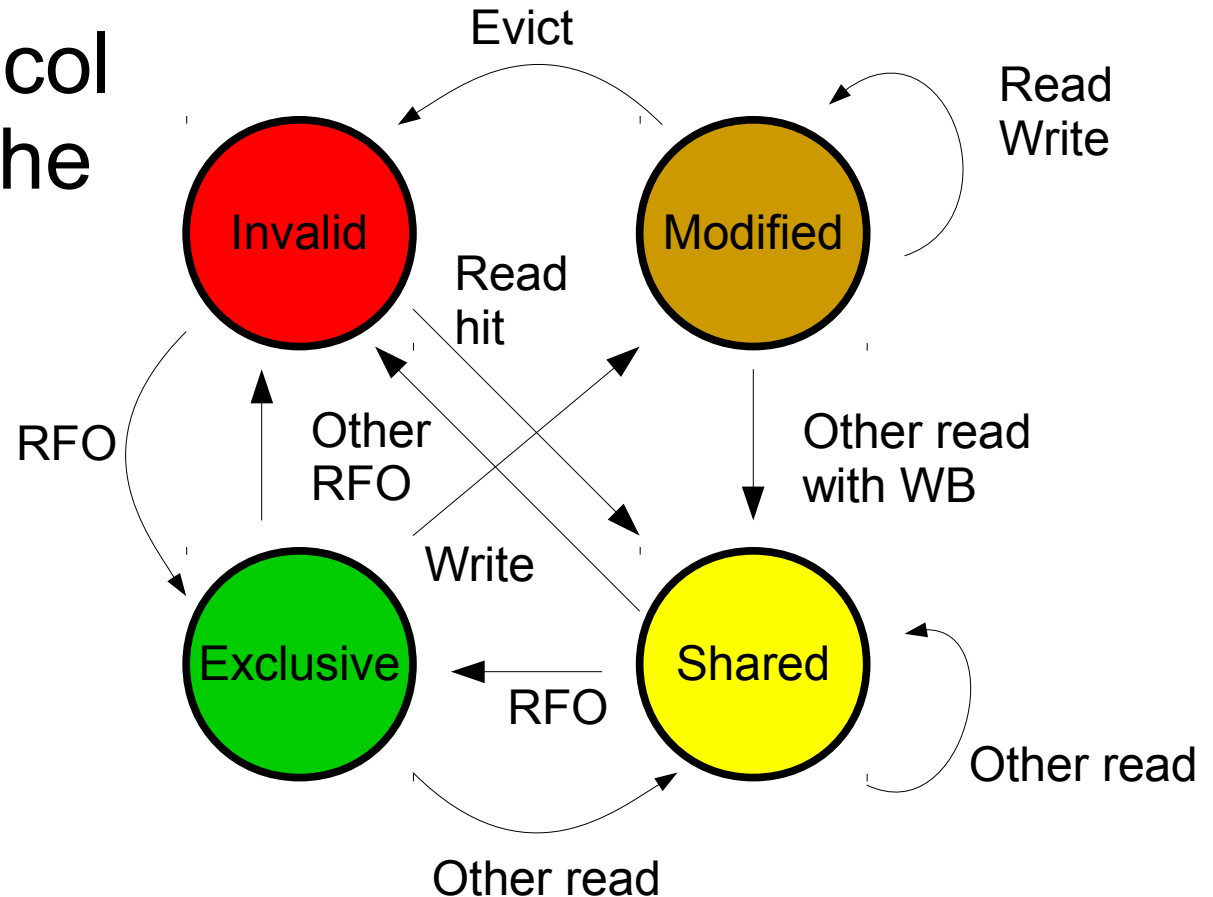
Cache Coherency



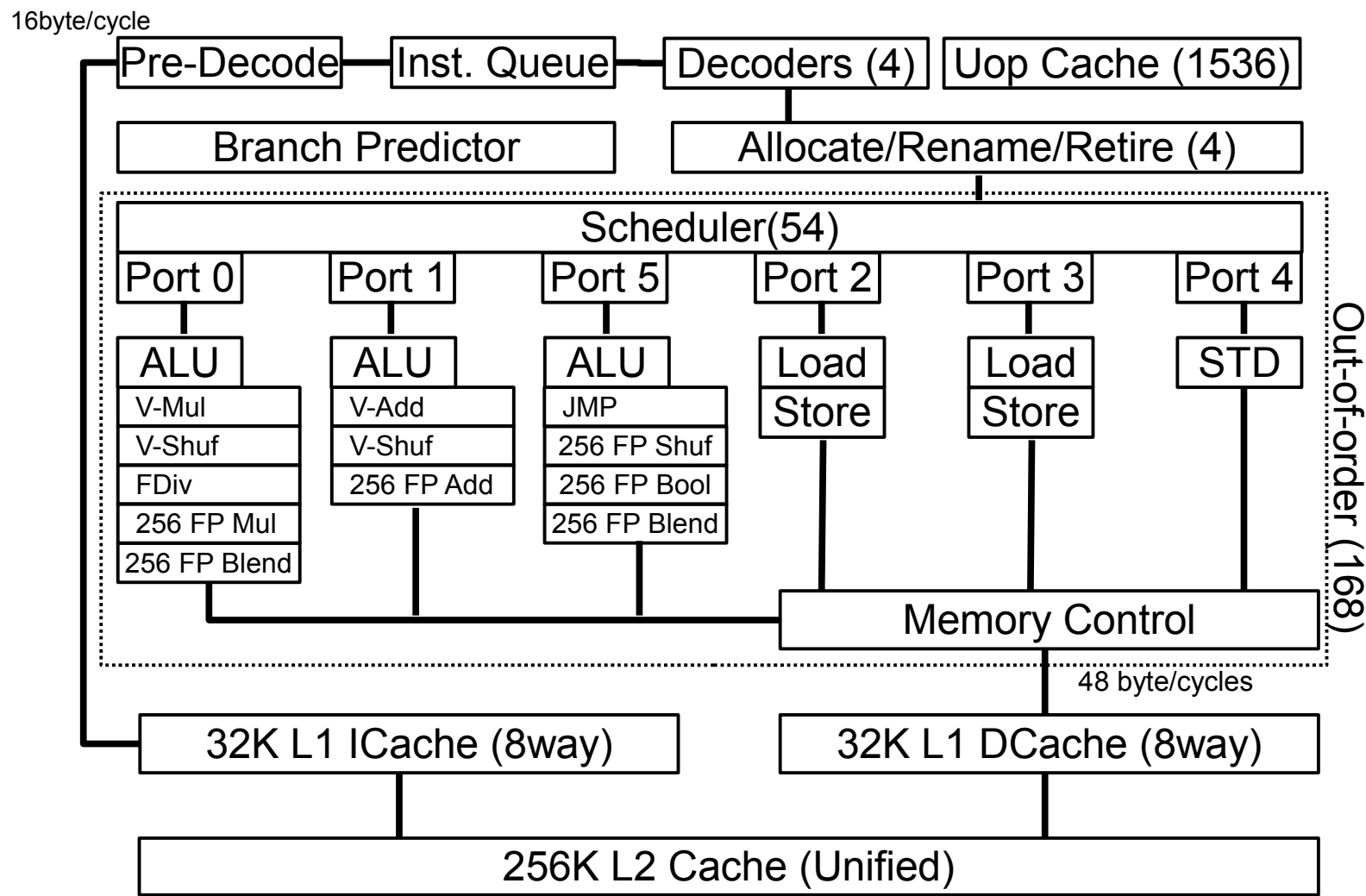
When a memory address being present in two caches is changed in one cache, the other cache needs to be notified that the data is now invalid.

Cache Coherency - MESI

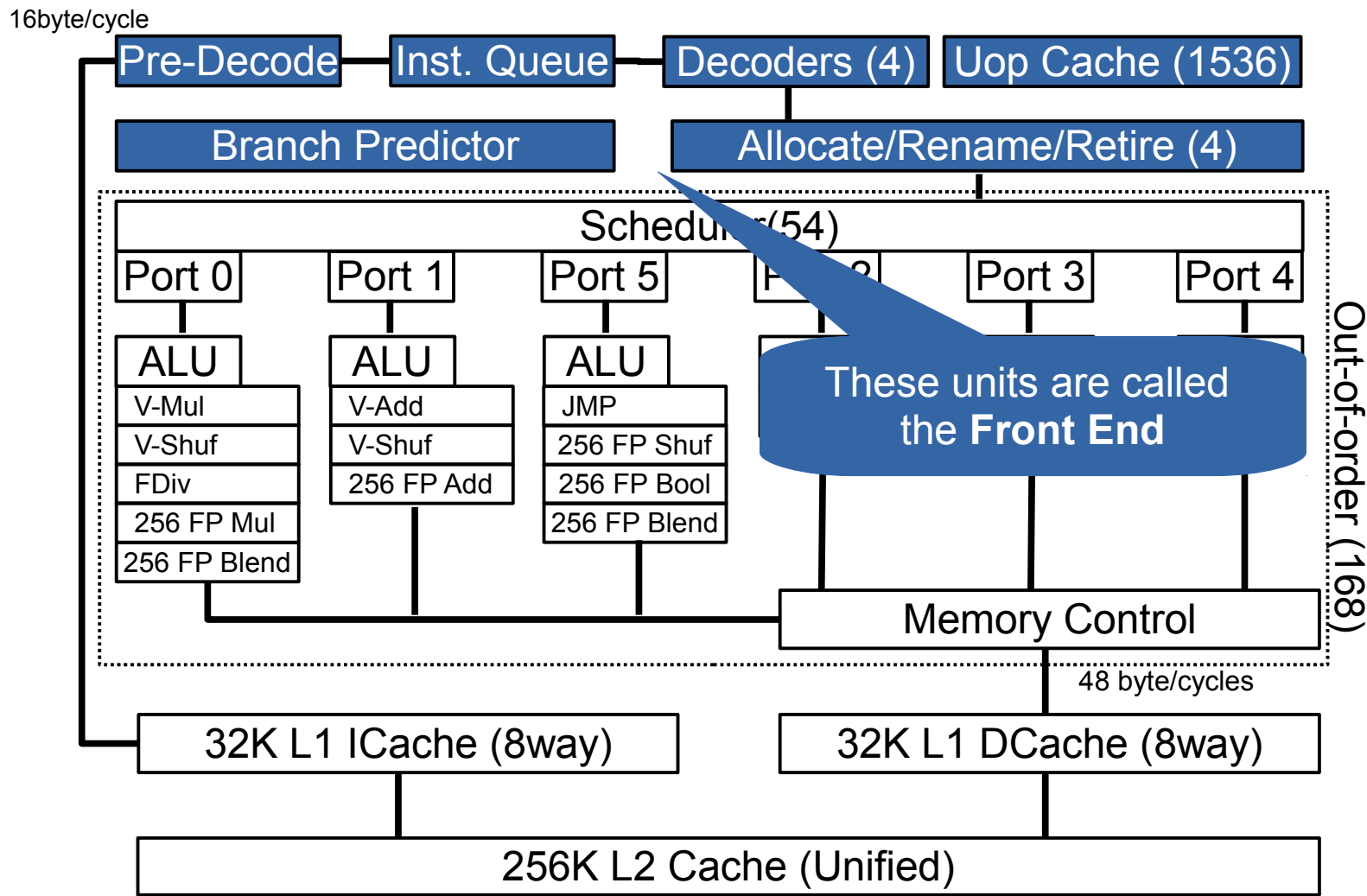
- MESI is the protocol employed for cache coherency:
- Modified
- Exclusive
- Shared
- Invalid



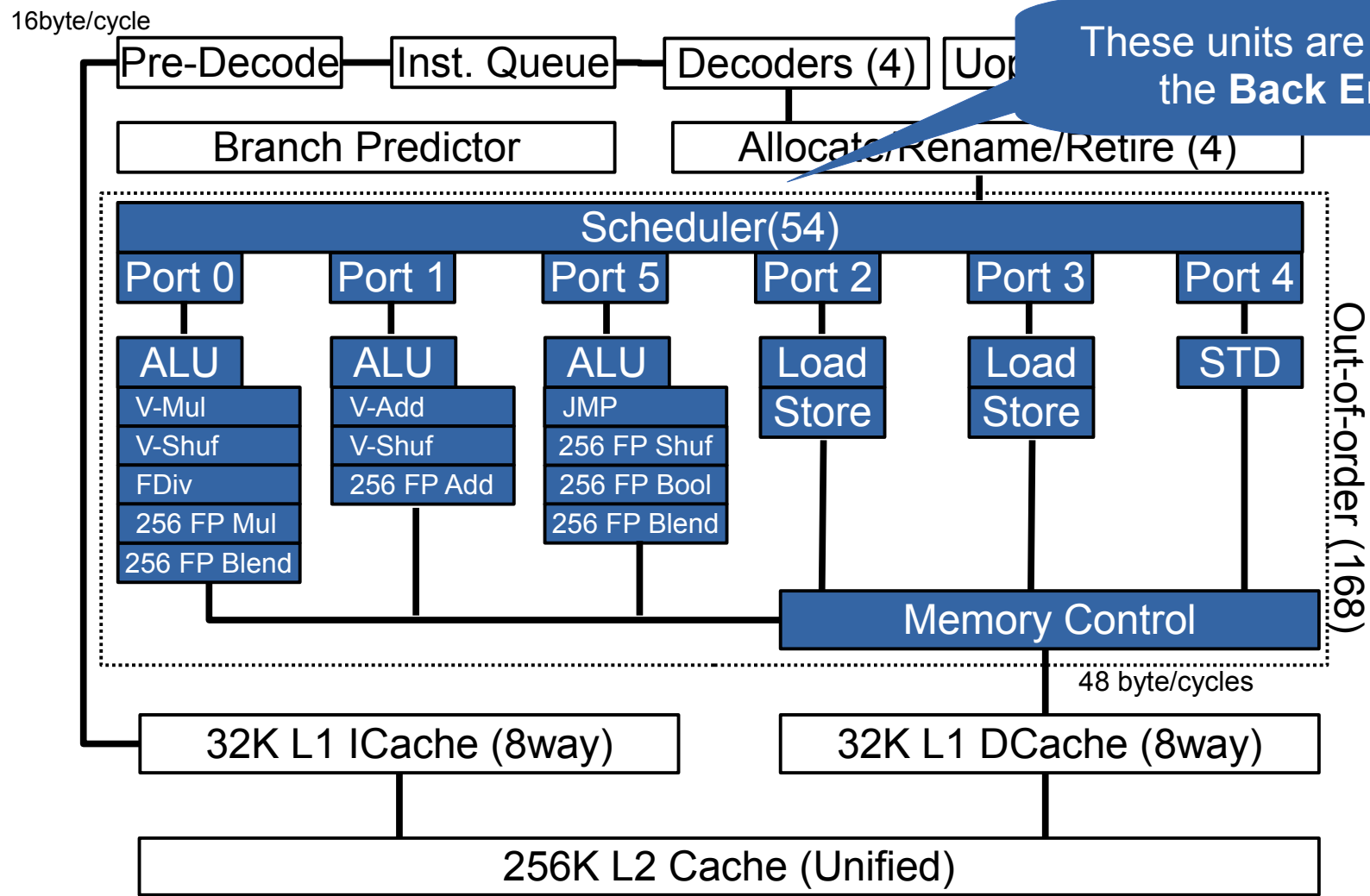
Sandy Bridge Pipeline



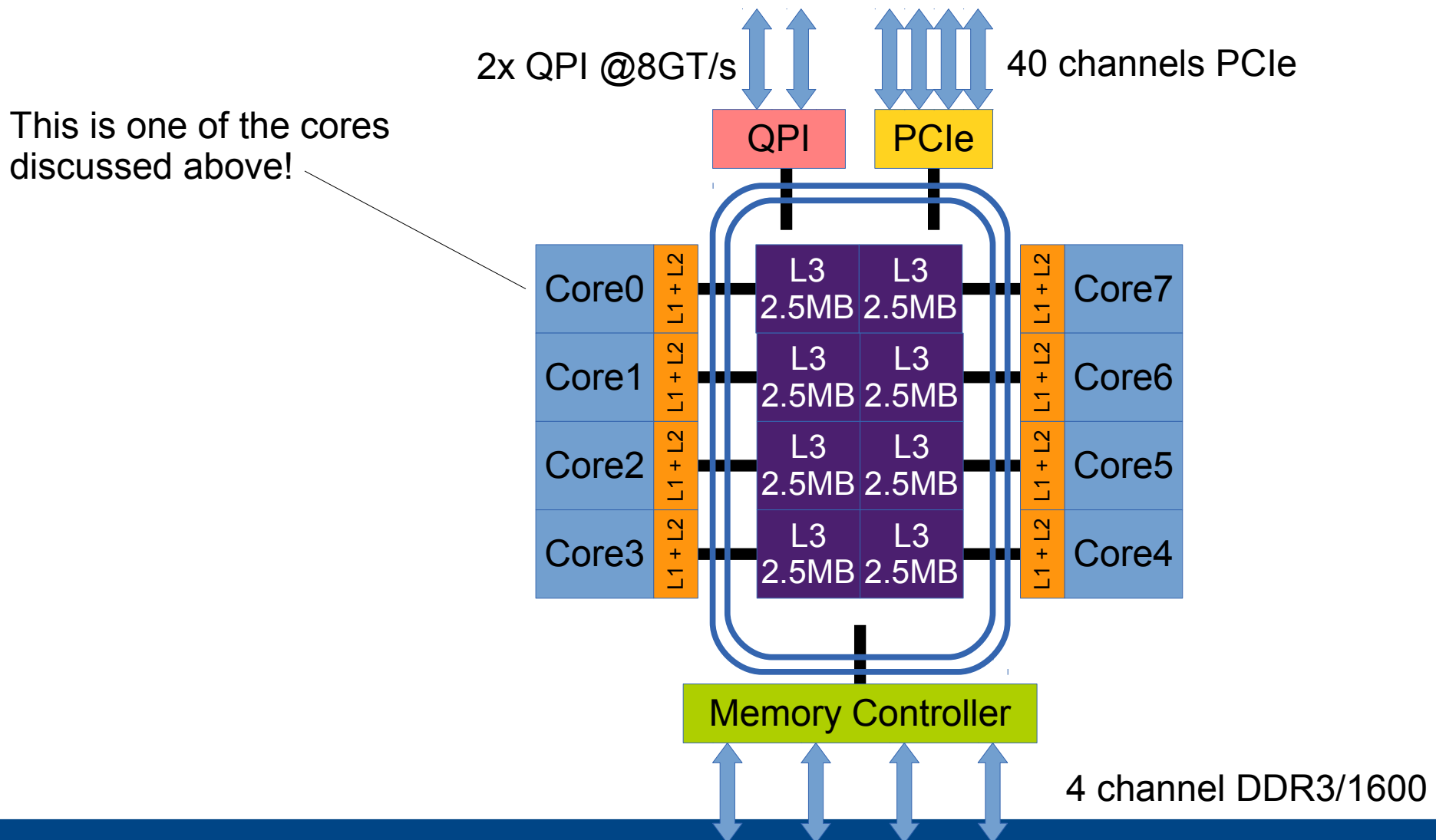
Sandy Bridge Pipeline



Sandy Bridge Pipeline



Sandy Bridge Processor



Summary

- Modern CPU architecture, such as Sandy Bridge, base on
 - Pipelining
 - Out of order execution
 - Superscalarity
 - SIMD
 - Branch prediction
 - Caches
- Thorough optimization needs to reflect these features