# Basics of modern computers

Dag Sverre Seljebotn

September 7, 2012

# The world of the 50's to 90's:
# Central Processing Unit + Random Access Memory

# Memory, encoding and pointers

Everything (data, program) is encoded in *memory bytes*, which (nowadays) are always 8 bits, allowing $2^8 = 256$ distinct values.

$$
\begin{aligned}
124_{10} &= 1 \cdot 10^2 + 2 \cdot 10^1 + 4 \cdot 10^0 \\
&= 0 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0
\end{aligned}
$$

Demo: `binary.c`

# Memory, encoding and pointers

Everything (data, program) is encoded in *memory bytes*, which (nowadays) are always 8 bits, allowing $2^8 = 256$ distinct values.

$$
\begin{aligned}
124_{10} &= 1 \cdot 10^2 + 2 \cdot 10^1 + 4 \cdot 10^0 \\
&= 0 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 \\
&= 7 \cdot 16^1 + 12 \cdot 16^0 = \texttt{0x7c}
\end{aligned}
$$

Demo: `binary.c`

# Pointers (as seen in C)

| | |
|---|---|
| `int32_t x;` | Declare a 4-byte integer "x" |

# Pointers (as seen in C)

| | |
|---|---|
| `int32_t x;` | Declare a 4-byte integer "x" |
| `x = 42;` | Memory of "x" now contains 42 |

# Pointers (as seen in C)

| | |
|---|---|
| `int32_t x;` | Declare a 4-byte integer "x" |
| `x = 42;` | Memory of "x" now contains 42 |
| `int32_t *p;` | Declare an 8-byte **pointer to int** "p" |

# Pointers (as seen in C)

| | |
|---|---|
| `int32_t x;` | Declare a 4-byte integer "x" |
| `x = 42;` | Memory of "x" now contains 42 |
| `int32_t *p;` | Declare an 8-byte **pointer to int** "p" |
| `p = &x;` | Memory of "p" now contains 0x00007fffa63cc4b0 |

# Pointers (as seen in C)

| | |
|---|---|
| `int32_t x;` | Declare a 4-byte integer "x" |
| `x = 42;` | Memory of "x" now contains 42 |
| `int32_t *p;` | Declare an 8-byte **pointer to int** "p" |
| `p = &x;` | Memory of "p" now contains 0x00007fffa63cc4b0 |
| `*p = 10;` | Memory of "x" now contains 10 |

# Pointers and arrays...

| | |
|---|---|
| `int32_t *p` | "x" is a pointer to integers |

# Pointers and arrays...

| | |
|---|---|
| `int32_t *p` | "x" is a pointer to integers |
| `p = malloc(40);` | Allocate space for 10 integers ("array") |
| | Memory of "x" contains 0x00007fffa63cc4b0 |

# Pointers and arrays...

| | |
|---|---|
| `int32_t *p` | "x" is a pointer to integers |
| `p = malloc(40);` | Allocate space for 10 integers ("array") |
| | Memory of "x" contains 0x00007fffa63cc4b0 |
| `int32_t *u;` | |

# Pointers and arrays...

| | |
|---|---|
| `int32_t *p` | "x" is a pointer to integers |
| `p = malloc(40);` | Allocate space for 10 integers ("array") |
| | Memory of "x" contains 0x00007fffa63cc4b0 |
| `int32_t *u;` | |
| `u = p + 1;` | Memory of "u" contains 0x00007fffa63cc4b4 |

# Pointers and arrays...

| | |
|---|---|
| `int32_t *p` | "x" is a pointer to integers |
| `p = malloc(40);` | Allocate space for 10 integers ("array") |
| | Memory of "x" contains 0x00007fffa63cc4b0 |
| `int32_t *u;` | |
| `u = p + 1;` | Memory of "u" contains 0x00007fffa63cc4b4 |
| `*p = 10;` | Assign first element of array |

# Pointers and arrays...

| | |
|---|---|
| `int32_t *p` | "x" is a pointer to integers |
| `p = malloc(40);` | Allocate space for 10 integers ("array") |
| | Memory of "x" contains 0x00007fffa63cc4b0 |
| `int32_t *u;` | |
| `u = p + 1;` | Memory of "u" contains 0x00007fffa63cc4b4 |
| `*p = 10;` | Assign first element of array |
| `*u = 20;` | Assign second element of array |

# Pointers and arrays...

| | |
|---|---|
| `int32_t *p` | "x" is a pointer to integers |
| `p = malloc(40);` | Allocate space for 10 integers ("array") |
| | Memory of "x" contains 0x00007fffa63cc4b0 |
| `int32_t *u;` | |
| `u = p + 1;` | Memory of "u" contains 0x00007fffa63cc4b4 |
| `*p = 10;` | Assign first element of array |
| `*u = 20;` | Assign second element of array |
| | |
| `p[0] = 10;` | Same as `*p = 10` |

# Pointers and arrays...

| | |
|---|---|
| `int32_t *p` | "x" is a pointer to integers |
| `p = malloc(40);` | Allocate space for 10 integers ("array") |
| | Memory of "x" contains 0x00007fffa63cc4b0 |
| `int32_t *u;` | |
| `u = p + 1;` | Memory of "u" contains 0x00007fffa63cc4b4 |
| `*p = 10;` | Assign first element of array |
| `*u = 20;` | Assign second element of array |
| | |
| `p[0] = 10;` | Same as `*p = 10` |
| `p[1] = 20;` | Same as `*(p + 1) = 20` |

# Pointers and arrays...

| | |
|---|---|
| `int32_t *p` | "x" is a pointer to integers |
| `p = malloc(40);` | Allocate space for 10 integers ("array") |
| | Memory of "x" contains 0x00007fffa63cc4b0 |
| `int32_t *u;` | |
| `u = p + 1;` | Memory of "u" contains 0x00007fffa63cc4b4 |
| `*p = 10;` | Assign first element of array |
| `*u = 20;` | Assign second element of array |
| | |
| `p[0] = 10;` | Same as `*p = 10` |
| `p[1] = 20;` | Same as `*(p + 1) = 20` |
| `p[10] = 30;` | Trampling on unallocated memory! |

# Multi-dimensional arrays

```
int32_t array = malloc(400);
int32_t nrows = 5, ncols = 20;

/* Treat memory in row-major ordering (C-order) */
for (int32_t i = 0; i < nrows; i++) {
  for (int32_t j = 0; j < ncols; j++) {
    array[i * ncols + j] = some_function_of(i, j);
  }
}

/* Treat memory in column-major ordering (Fortran-order) */
for (int32_t i = 0; i < nrows; i++) {
  for (int32_t j = 0; j < ncols; j++) {
    array[i + j * nrows] = some_function_of(i, j);
  }
}
```

# Other uses for pointers

Consider the Fortran routine:

```
subroutine square_me(x)
    integer(4) :: x
    x = x * x
end subroutine
...
call square_me(a) ! a is squared
```

C is closer to hardware:

```
void square_me(int32_t x) {
    x = x * x;
}
...
square_me(a); /* nothing happens */
```

# Other uses for pointers

Consider the Fortran routine:

```
subroutine square_me(x)
    integer(4) :: x
    x = x * x
end subroutine
...
call square_me(a) ! a is squared
```

C is closer to hardware:

```
void square_me(int32_t *p) {
    int32_t x = *p;
    *p = x * x;
}
...
square_me(&a); /* a is squared */
```

# Simple model of CPUs

- **Registers:** "Hardware variables". E.g., RAX/EAX, RBX/EAX, XMM0, . . . ; about 32 registers on 64-bit AMD/Intel CPUs

# Simple model of CPUs

- **Registers:** "Hardware variables". E.g., RAX/EAX, RBX/EAX, XMM0, . . . ; about 32 registers on 64-bit AMD/Intel CPUs
- **Instructions:** Stream of numbers telling the CPU what to do.

# Simple model of CPUs

- **Registers:** "Hardware variables". E.g., RAX/EAX, RBX/EAX, XMM0, . . . ; about 32 registers on 64-bit AMD/Intel CPUs
- **Instructions:** Stream of numbers telling the CPU what to do.
  `mov eax, [rsp]` Load from memory

# Simple model of CPUs

- **Registers:** "Hardware variables". E.g., RAX/EAX, RBX/EAX, XMM0, . . . ; about 32 registers on 64-bit AMD/Intel CPUs
- **Instructions:** Stream of numbers telling the CPU what to do.

| | |
|---|---|
| `mov eax, [rsp]` | Load from memory |
| `imul eax, eax` | Do something with register |

# Simple model of CPUs

- **Registers:** "Hardware variables". E.g., RAX/EAX, RBX/EAX, XMM0, . . . ; about 32 registers on 64-bit AMD/Intel CPUs
- **Instructions:** Stream of numbers telling the CPU what to do.

| | |
|---|---|
| `mov eax, [rsp]` | Load from memory |
| `imul eax, eax` | Do something with register |
| `mov [rsp], eax` | Store to memory |

# Simple model of CPUs

- **Registers:** "Hardware variables". E.g., RAX/EAX, RBX/EAX, XMM0, . . . ; about 32 registers on 64-bit AMD/Intel CPUs
- **Instructions:** Stream of numbers telling the CPU what to do.

| | |
|---|---|
| `mov eax, [rsp]` | Load from memory |
| `imul eax, eax` | Do something with register |
| `mov [rsp], eax` | Store to memory |

Example: Disassemble `add_and_square.o`

# Ways of using memory

Conventionally we divide memory into:

# Ways of using memory

Conventionally we divide memory into:

- **Static**: Your program, global variables, constants

# Ways of using memory

Conventionally we divide memory into:

- **Static**: Your program, global variables, constants
- **Heap**: Dynamically allocated/deallocated blocks
  - C: `malloc()`, `free()` functions
  - C++: `new`, `delete` keywords
  - Fortran: `allocate`, `deallocate` keywords
  - Modern languages: `new` + Garbage Collection

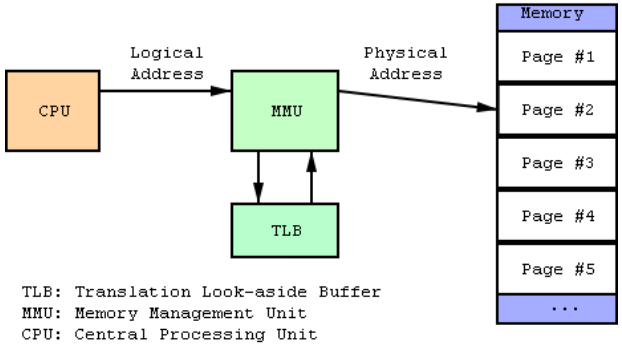
# Ways of using memory

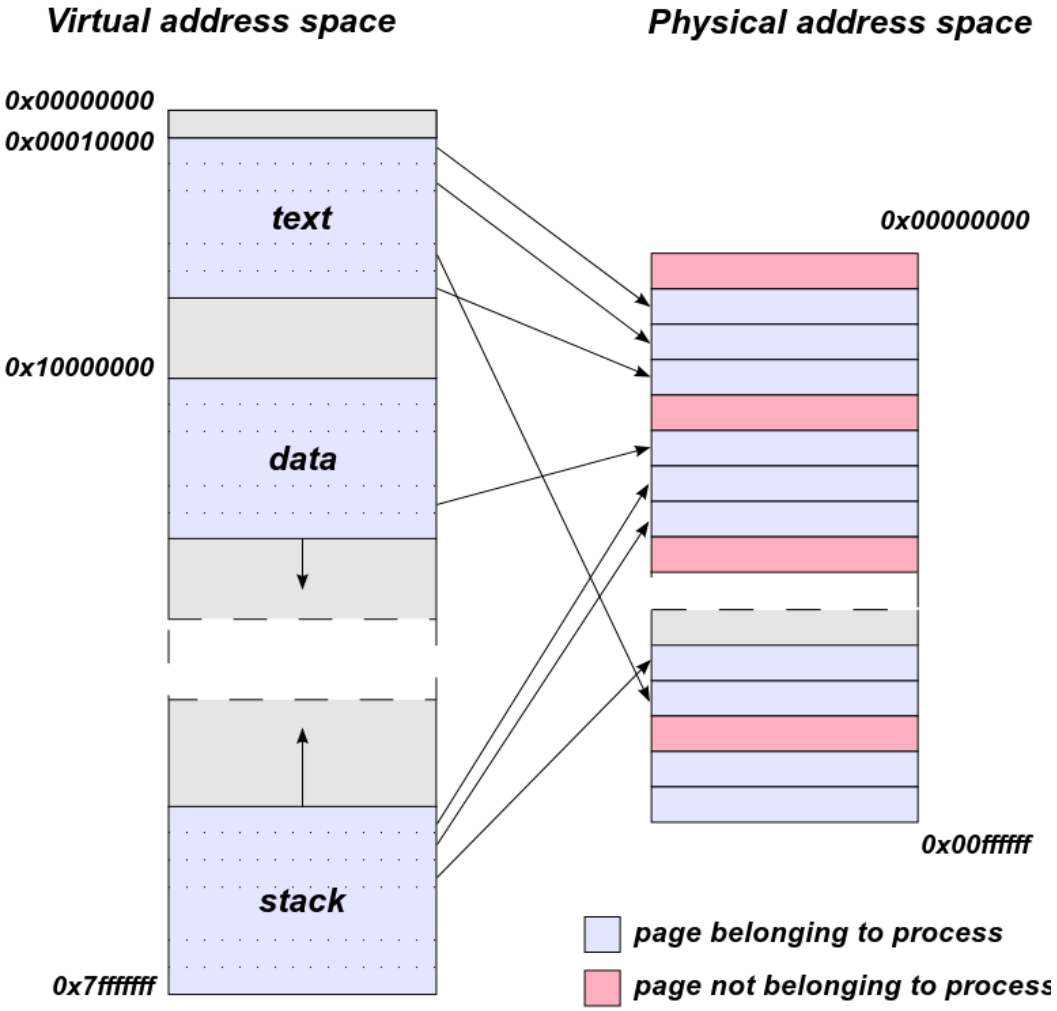
Conventionally we divide memory into:

- **Static**: Your program, global variables, constants
- **Heap**: Dynamically allocated/deallocated blocks
  - C: `malloc()`, `free()` functions
  - C++: `new`, `delete` keywords
  - Fortran: `allocate`, `deallocate` keywords
  - Modern languages: `new` + Garbage Collection
- **Stack**: Allocated at program startup
  - Implicitly used through program flow

Demo program: `knapsack.c`

# Virtual memory




Illustrations from Wikipedia

# The memory bandwidth wall

# Benchmarking

- **Problem:** What speed is your CPU *really* running at?
  1. Power saving
  2. "Turbo mode"

# Benchmarking

- **Problem:** What speed is your CPU *really* running at?
  1. Power saving
  2. "Turbo mode"

  **Solution:** Disable these features

# Benchmarking

- **Problem:** What speed is your CPU *really* running at?
  1. Power saving
  2. "Turbo mode"

  **Solution:** Disable these features

- **Problem:** Interruptions from the OS

# Benchmarking

- **Problem:** What speed is your CPU *really* running at?
  1. Power saving
  2. "Turbo mode"

  **Solution:** Disable these features

- **Problem:** Interruptions from the OS

  **Solution:** Take many samples of *wall time*, then use *minimum*. For each sample, may have to repeat and average.

# Benchmarking

- **Problem:** What speed is your CPU *really* running at?
  1. Power saving
  2. "Turbo mode"

  **Solution:** Disable these features

- **Problem:** Interruptions from the OS

  **Solution:** Take many samples of *wall time*, then use *minimum*. For each sample, may have to repeat and average.

- **Problem:** Does benchmark include memory bus etc. or just CPU?

# Benchmarking

- **Problem:** What speed is your CPU *really* running at?
  1. Power saving
  2. "Turbo mode"

  **Solution:** Disable these features

- **Problem:** Interruptions from the OS

  **Solution:** Take many samples of *wall time*, then use *minimum*. For each sample, may have to repeat and average.

- **Problem:** Does benchmark include memory bus etc. or just CPU?
  **Solution:** Benchmark using all CPU cores and with realistically sized data sets

# Benchmarking

- **Problem:** What speed is your CPU *really* running at?
  1. Power saving
  2. "Turbo mode"

  **Solution:** Disable these features

- **Problem:** Interruptions from the OS

  **Solution:** Take many samples of *wall time*, then use *minimum*. For each sample, may have to repeat and average.

- **Problem:** Does benchmark include memory bus etc. or just CPU?
  **Solution:** Benchmark using all CPU cores and with realistically sized data sets

- **Problem:** The compiler optimizing away the test code

Experiments exploring memory bus, cache, cache lines:

1. Memory bandwidth

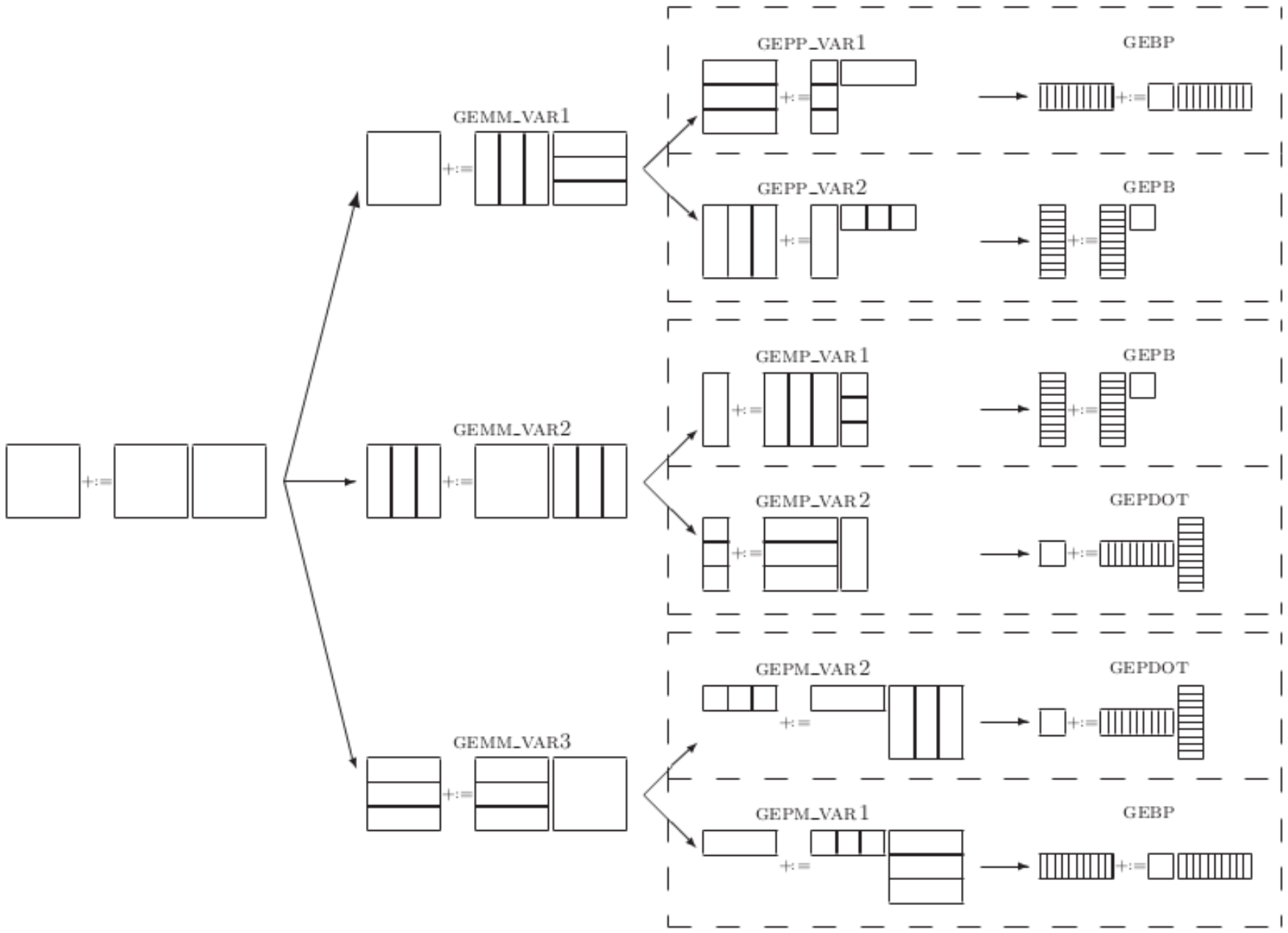
Experiments exploring memory bus, cache, cache lines:

1. Memory bandwidth
2. Memory latency

Experiments exploring memory bus, cache, cache lines:

1. Memory bandwidth
2. Memory latency
3. Matrix transpose

# Matrix multiplication (Goto & van de Geijn, 2008)



# Parallel computing

# Multi-core false dependencies (cache line ping pong)

Experiment: `pingpong.c`

# Competing programming models

# Competing programming models

- MPI: Explicit sharing through message passing (harder, safer)

# Competing programming models

- MPI: Explicit sharing through message passing (harder, safer)
- OpenMP: Implicit sharing through shared memory (easier, less safe)

# Competing programming models

- MPI: Explicit sharing through message passing (harder, safer)
- OpenMP: Implicit sharing through shared memory (easier, less safe)

Example: NERSC Hopper cluster

- 6384 nodes
- 24 cores per node
- 32 GB memory per node $\rightarrow$ 1.33 GB per core

# Competing programming models

- MPI: Explicit sharing through message passing (harder, safer)
- OpenMP: Implicit sharing through shared memory (easier, less safe)

Example: NERSC Hopper cluster

- 6384 nodes
- 24 cores per node
- 32 GB memory per node $\rightarrow$ 1.33 GB per core

May be better to write 24-threaded program with 32 GB memory than 1-threaded program with 1.33 GB memory...BUT!

# Non-Uniform Memory Architecture

```
dagss@owl3 $ numactl --hardware
available: 8 nodes (0-7)
node 0 size: 24216 MB
node 0 free: 22685 MB
node 1 size: 24240 MB
node 1 free: 24111 MB
node 2 size: 24240 MB
node 2 free: 24158 MB
<...>
node distances:
node   0   1   2   3   4   5   6   7
  0:  10  16  16  22  16  22  16  22
  1:  16  10  16  22  22  16  22  16
  2:  16  16  10  16  16  16  16  22
  3:  22  22  16  10  16  16  22  16
  4:  16  22  16  16  10  16  16  16
  5:  22  16  16  16  16  10  22  22
  6:  16  22  16  22  16  22  10  16
  7:  22  16  22  16  16  22  16  10
```

# Non-Uniform Memory Architecture: AMD

```
dagss@owl3 $ bin/numabench 6
== Write (6 threads per node, GiB/s) ==
   5.9     4.9     2.2     1.8     2.1     1.8     4.0     3.6
   4.9     5.9     2.1     1.8     1.8     2.0     1.9     2.2
   2.1     2.2     5.9     4.9     2.2     2.2     2.0     1.8
   2.1     2.1     4.9     5.9     4.0     4.0     1.8     2.1
   3.7     3.6     2.2     3.9     5.9     4.9     2.2     3.9
   3.5     3.7     2.2     2.2     4.8     6.0     2.1     3.8
   2.2     1.9     3.7     3.5     2.2     1.8     5.9     4.9
   1.8     2.2     1.8     2.1     2.1     1.8     4.9     5.9
== Read (6 threads per node, GiB/s) ==
  11.1     4.7     2.1     2.0     2.4     2.5     2.1     1.8
   4.7    11.2     2.1     2.1     2.5     2.4     1.8     2.1
   2.1     2.1    11.2     4.8     2.1     2.2     2.4     1.7
   1.8     1.8     4.8    11.1     4.1     2.1     2.5     2.0
   1.9     1.8     2.1     4.1    11.2     4.4     2.1     2.1
   1.8     1.9     2.2     2.7     3.2    11.2     1.8     1.8
   2.6     1.8     1.9     1.7     2.1     2.1    11.1     4.4
   2.5     2.1     1.7     2.0     2.6     2.3     4.7    11.1
```

# Non-Uniform Memory Architecture: Intel

```
dagss@mimosa $ bin/numabench 8
== Write (8 threads per node, GiB/s) ==
  10.1     7.2     5.4     5.1     4.6     5.4     5.3     7.3
   7.2    10.2     7.7     5.4     5.6     4.6     5.2     5.4
   5.4     7.6    10.0     7.2     5.4     5.2     4.5     5.5
   5.1     5.4     7.2    10.0     7.3     5.4     5.4     4.5
   4.5     5.4     5.3     7.2    10.0     7.2     5.4     5.1
   5.5     4.5     5.2     5.4     7.2    10.0     7.6     5.4
   5.3     5.2     4.5     5.5     5.4     7.6    10.0     7.2
   7.3     5.3     5.4     4.5     5.1     5.4     7.2    10.0
== Read (8 threads per node, GiB/s) ==
  22.0     7.6     8.2     7.4     7.1     7.8     8.0     8.5
   7.7    21.9     8.6     8.2     7.3     7.0     7.5     8.0
   8.1     9.0    21.8     7.6     8.0     7.4     7.0     7.4
   7.3     8.1     7.6    22.0     9.4     8.0     7.7     6.9
   7.0     7.7     7.9     9.2    21.9     7.6     8.2     7.3
   7.4     6.9     7.4     7.9     7.6    21.8     8.9     8.2
   8.0     7.5     7.0     7.3     8.1     8.6    21.8     7.6
   9.0     8.0     7.7     6.9     7.3     8.1     7.6    22.0
```

# Hybrid approach!

NERSC:

> *When using hybrid OpenMP-MPI in applications NERSC strongly suggests running with 4 MPI tasks and 6 OpenMP threads each per node. Using 1 MPI task and 24 OpenMP threads per node is NOT an optimal way to use the Hopper nodes, as it is very hard to address the performance implications of the Non-Uniform Memory Architecture (NUMA) of the Hopper nodes using more than 6 OpenMP threads.*

`http://1.usa.gov/UuFzYe`

`numactl` is your friend!

# Summary and opinions on parallelism

- Sharing memory is convenient but dangerous for performance

# Summary and opinions on parallelism

- Sharing memory is convenient but dangerous for performance
- Does OpenMP make it too easy to share memory?

# Summary and opinions on parallelism

- Sharing memory is convenient but dangerous for performance
- Does OpenMP make it too easy to share memory?
- Non-trivial code: Dead-locks, races, ...

# Summary and opinions on parallelism

- Sharing memory is convenient but dangerous for performance
- Does OpenMP make it too easy to share memory?
- Non-trivial code: Dead-locks, races, ...
- Alternative 1: Asynchronous message passing with ZeroMQ!
  - Beautiful programming interface!
  - Extremely fast
  - In-process, cross-process, cluster, or internet

ØMQ

# Summary and opinions on parallelism

- Sharing memory is convenient but dangerous for performance
- Does OpenMP make it too easy to share memory?
- Non-trivial code: Dead-locks, races, ...
- Alternative 1: Asynchronous message passing with ZeroMQ!
  - Beautiful programming interface!
  - Extremely fast
  - In-process, cross-process, cluster, or internet

ØMQ

- Alternative 2 (experts): Use MPI and explicitly share memory on same node
  - `man shmem`
  - `man mmap`

# Summary and opinions on parallelism

- Sharing memory is convenient but dangerous for performance
- Does OpenMP make it too easy to share memory?
- Non-trivial code: Dead-locks, races, ...
- Alternative 1: Asynchronous message passing with ZeroMQ!
  - Beautiful programming interface!
  - Extremely fast
  - In-process, cross-process, cluster, or internet

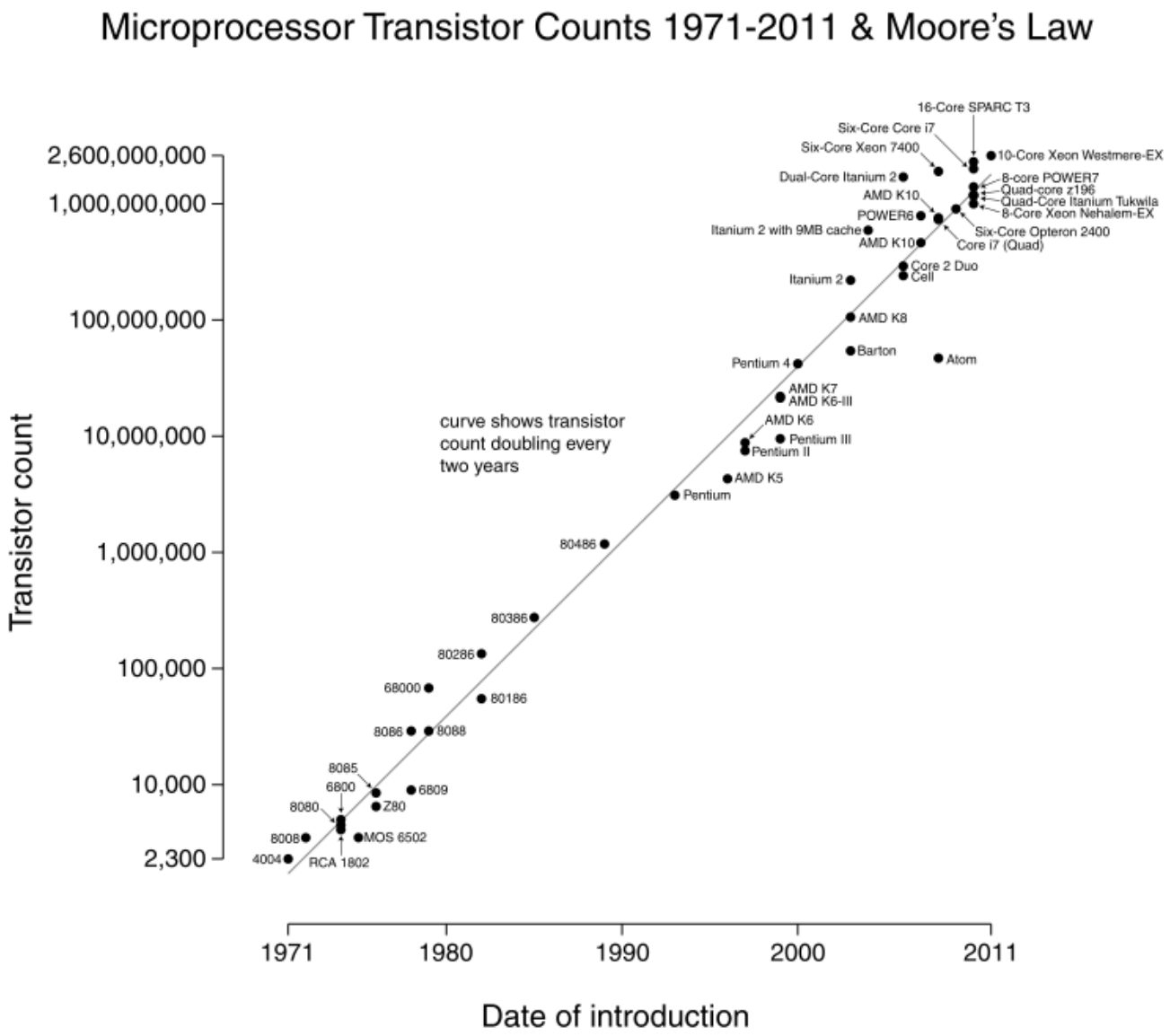
ØMQ

- Alternative 2 (experts): Use MPI and explicitly share memory on same node
  - `man shmem`
  - `man mmap`
- Alternative $n$: Too many frameworks to even begin...

# A more complicated picture of CPUs

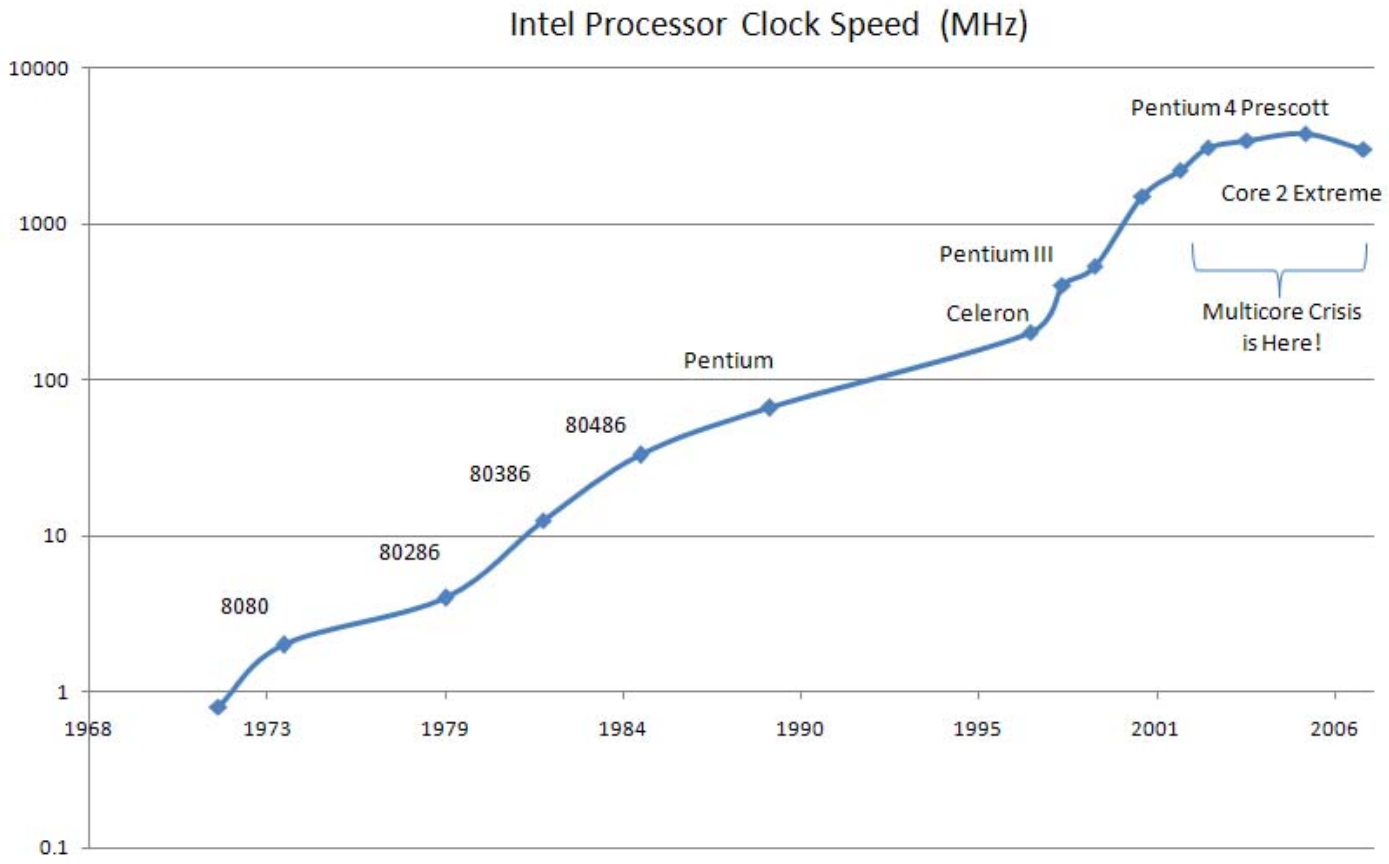
(+ some GPU)

# CPUs getting more complicated



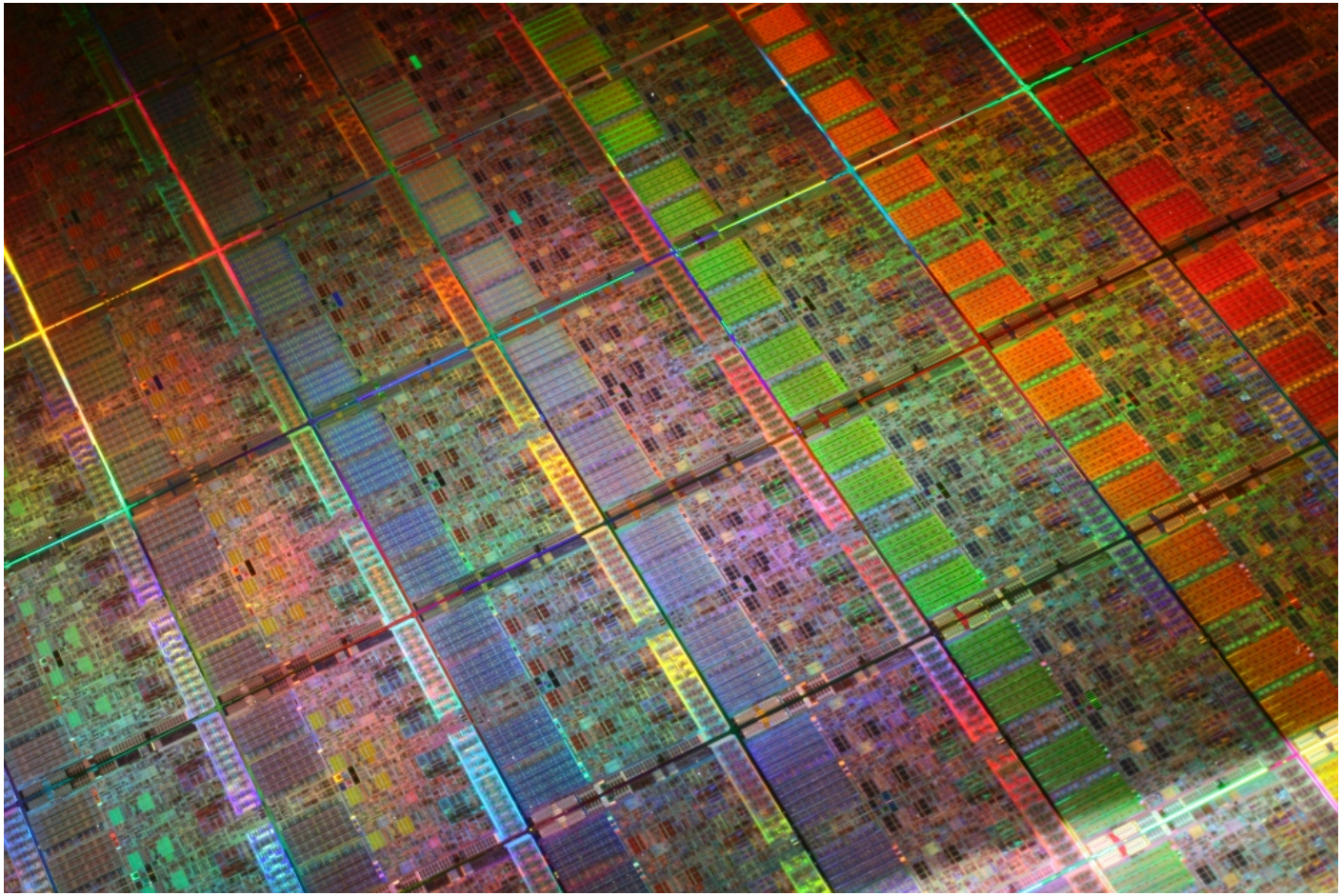
# CPUs getting more complicated

But:

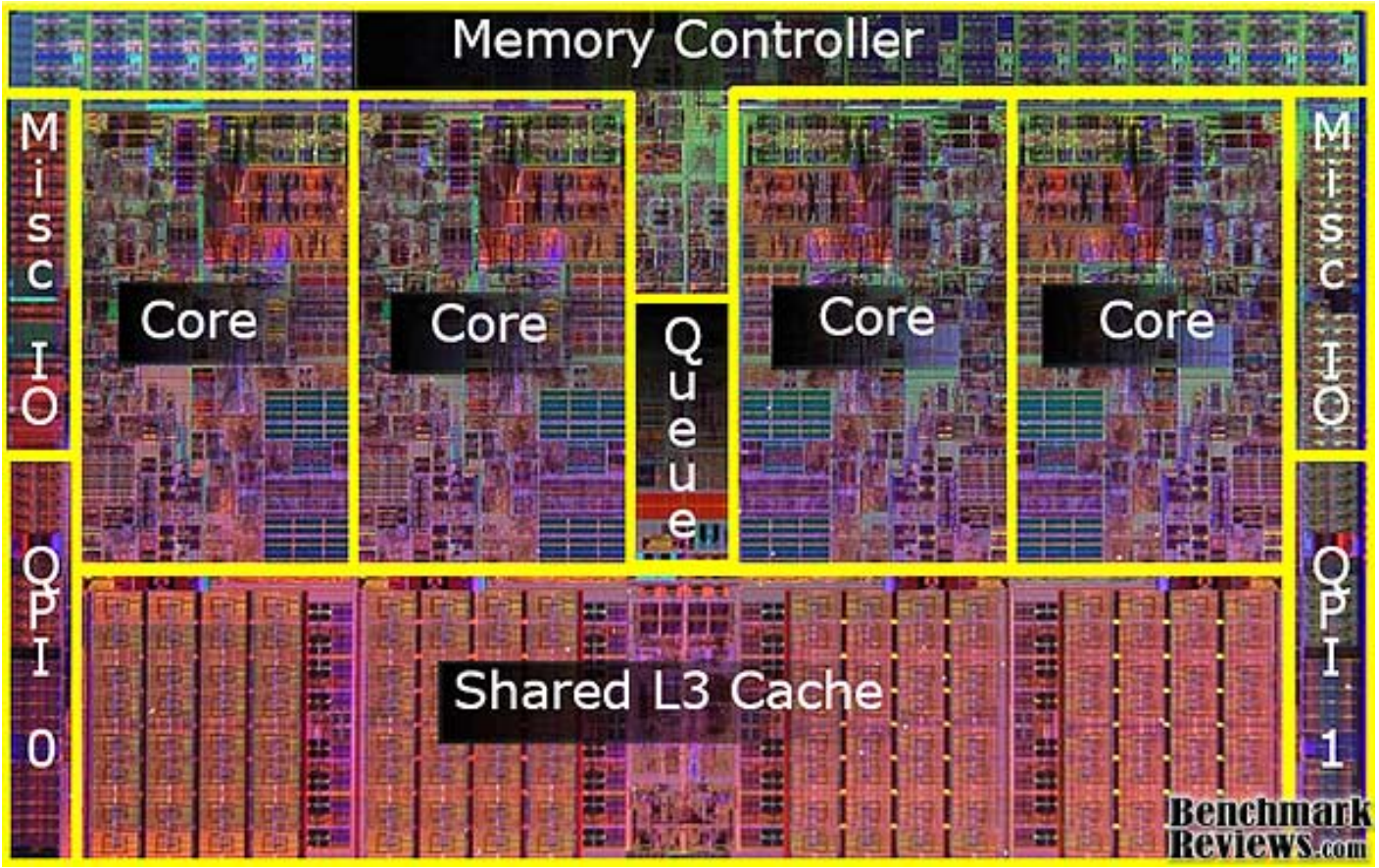


# CPUs today

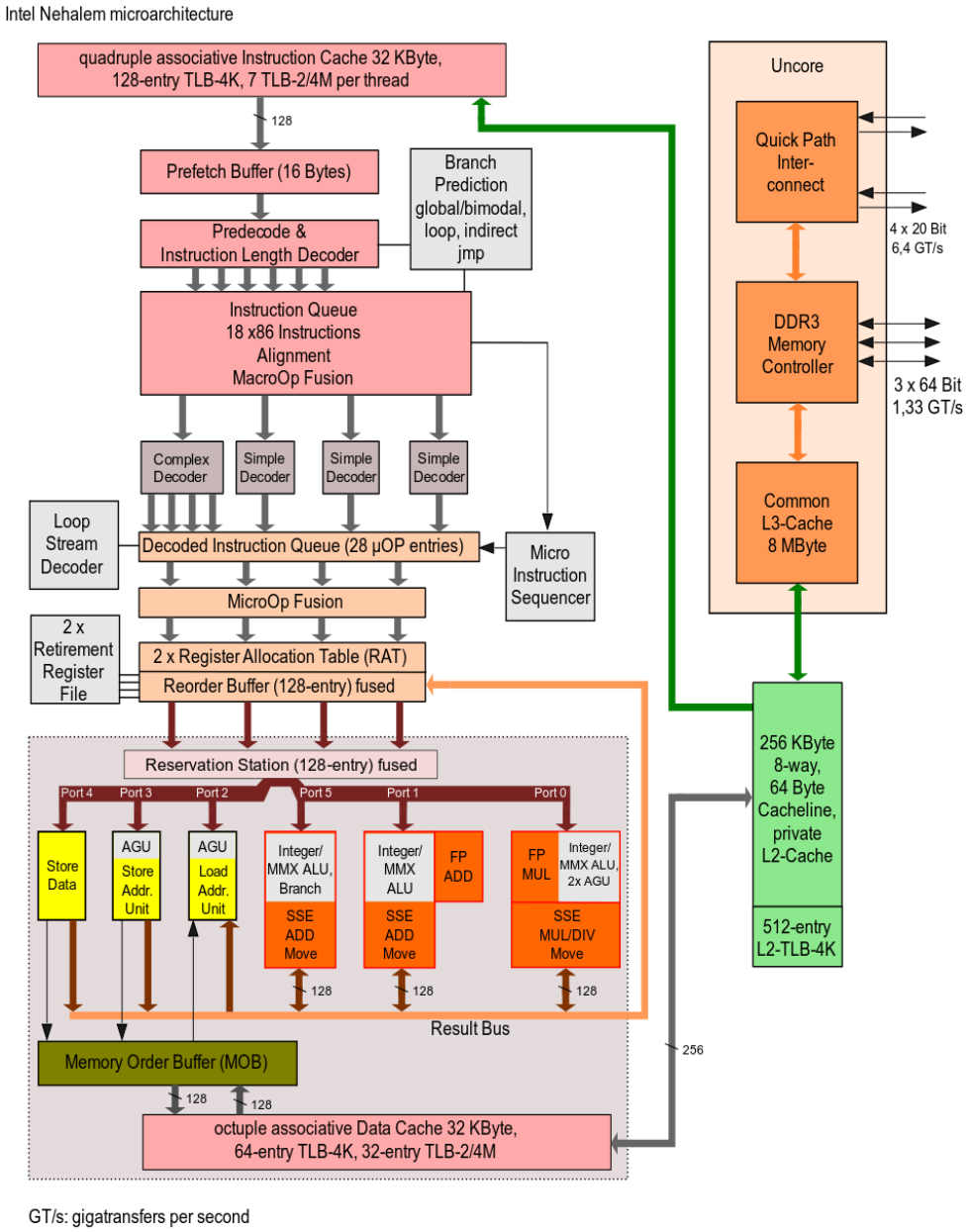
# CPUs today



# CPUs today

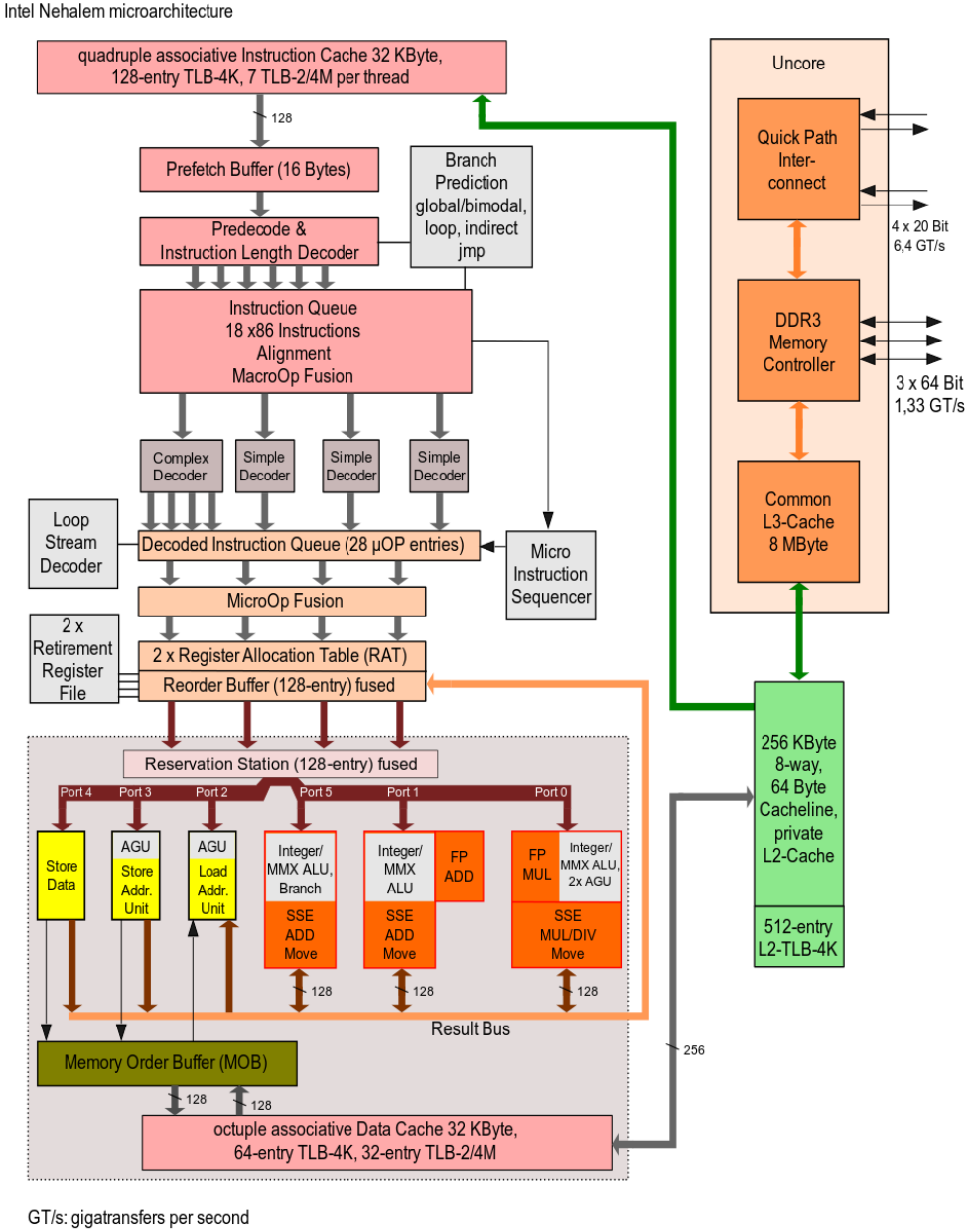

Memory Controller
Misc IO
Core
Core
Queue
Core
Core
Misc IO
QPI 0
Shared L3 Cache
QPI 1
Benchmark Reviews.com

# CPUs today


Intel Nehalem microarchitecture
quadruple associative Instruction Cache 32 KByte, 128-entry TLB-4K, 7 TLB-2/4M per thread
128
Prefetch Buffer (16 Bytes)
Branch Prediction global/bimodal, loop, indirect jmp
Predecode & Instruction Length Decoder
Instruction Queue 18 x86 Instructions Alignment MacroOp Fusion
Complex Decoder
Simple Decoder
Simple Decoder
Simple Decoder
Loop Stream Decoder
Decoded Instruction Queue (28 µOP entries)
Micro Instruction Sequencer
MicroOp Fusion
2 x Retirement Register File
2 x Register Allocation Table (RAT)
Reorder Buffer (128-entry) fused
Reservation Station (128-entry) fused
Port 4
Port 3
Port 2
Port 5
Port 1
Port 0
Store Data
AGU Store Addr. Unit
AGU Load Addr. Unit
Integer/ MMX ALU, Branch
SSE ADD Move
Integer/ MMX ALU
FP ADD
SSE ADD Move
FP MUL
Integer/ MMX ALU, 2x AGU
SSE MUL/DIV Move
128
128
128
Result Bus
Memory Order Buffer (MOB)
128
128
octuple associative Data Cache 32 KByte, 64-entry TLB-4K, 32-entry TLB-2/4M
256
Uncore
Quick Path Inter-connect
4 x 20 Bit 6,4 GT/s
DDR3 Memory Controller
3 x 64 Bit 1,33 GT/s
Common L3-Cache 8 MByte
256 KByte 8-way, 64 Byte Cacheline, private L2-Cache
512-entry L2-TLB-4K
GT/s: gigatransfers per second

# Out-of-order execution & register renaming

```
mov   eax, [rsp]
imul  eax, eax
mov   [rsp], eax
mov   eax, [rsp+0x4]
imul  eax, eax
mov   [rsp+0x4], eax
```



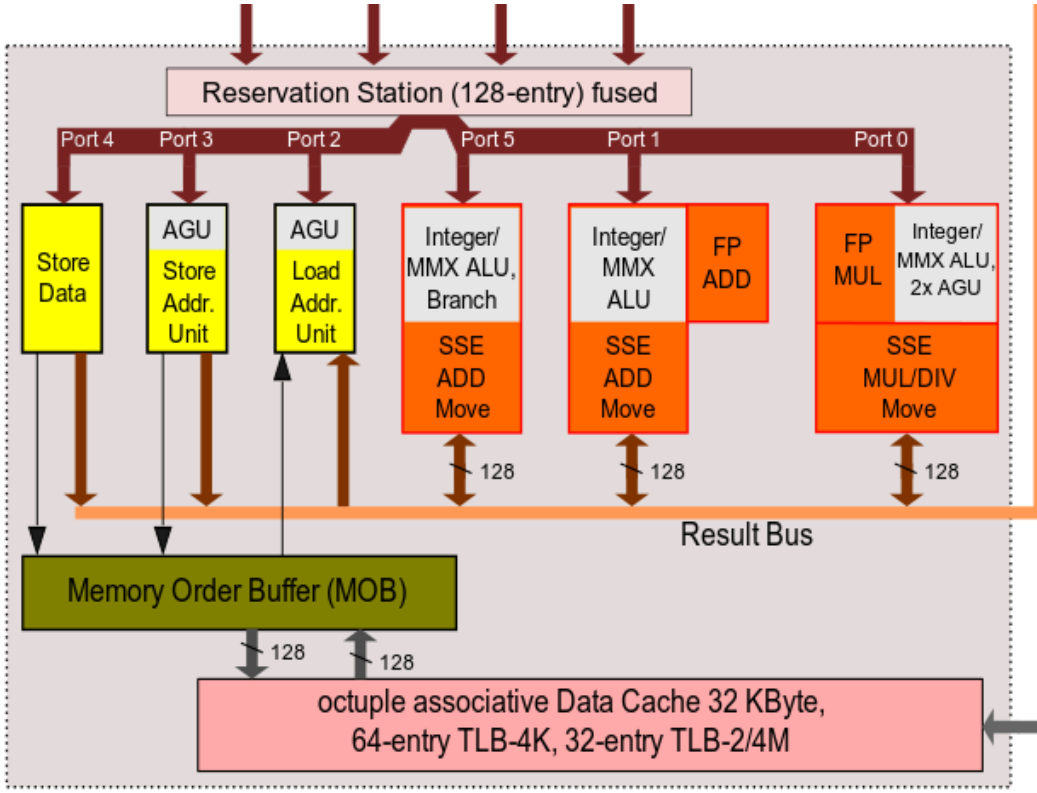
# Pipelining

Not so good:

```
for (i = 2; i < n; ++i) {
   out[i] = a[i] * out[i - 2] + b[i] * out[i - 1];
}
```

Good:

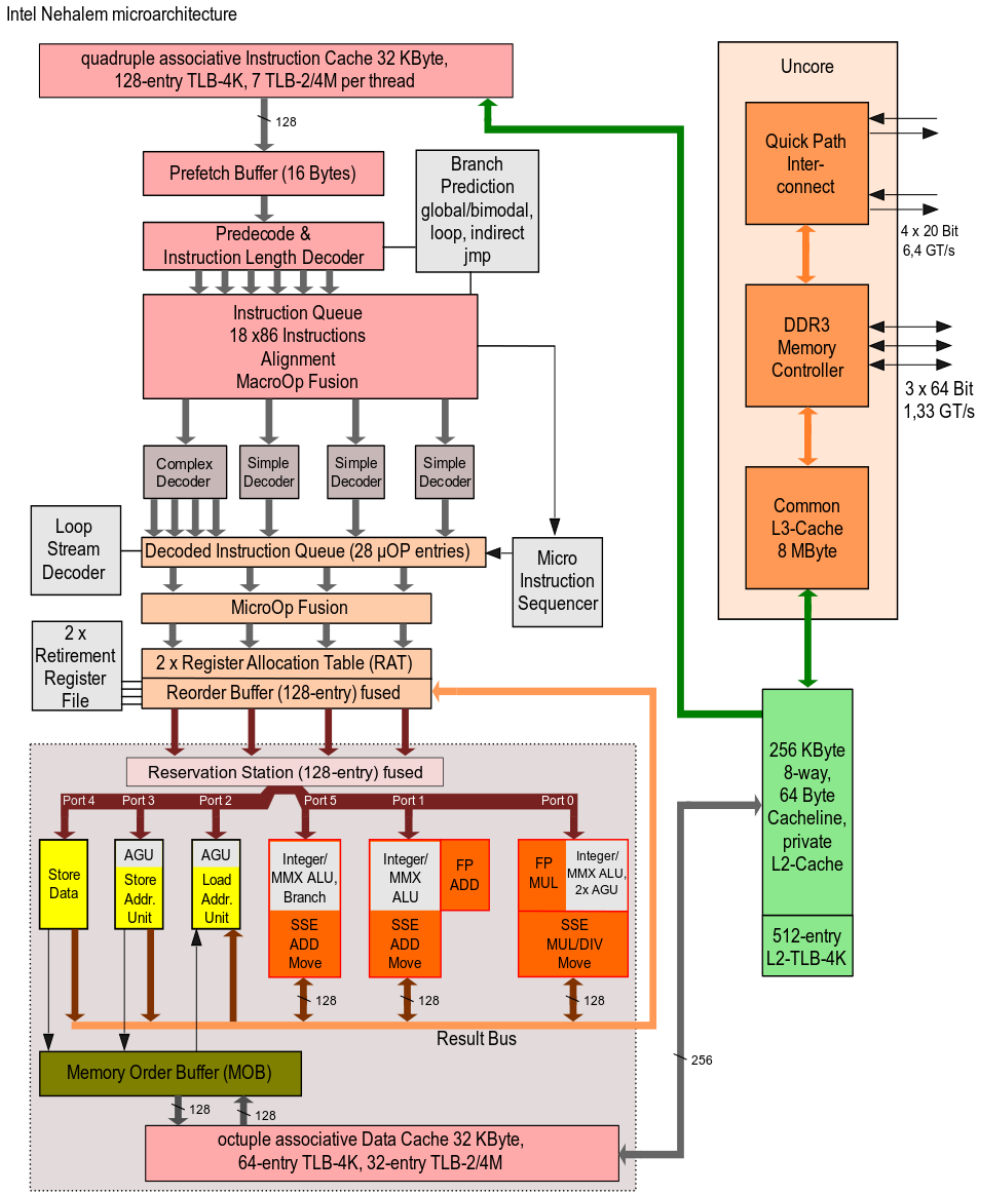
```
for (i = 0; i < n; ++i) {
   out[i] = a[i] + b[i];
}
```




GT/s: gigatransfers per second

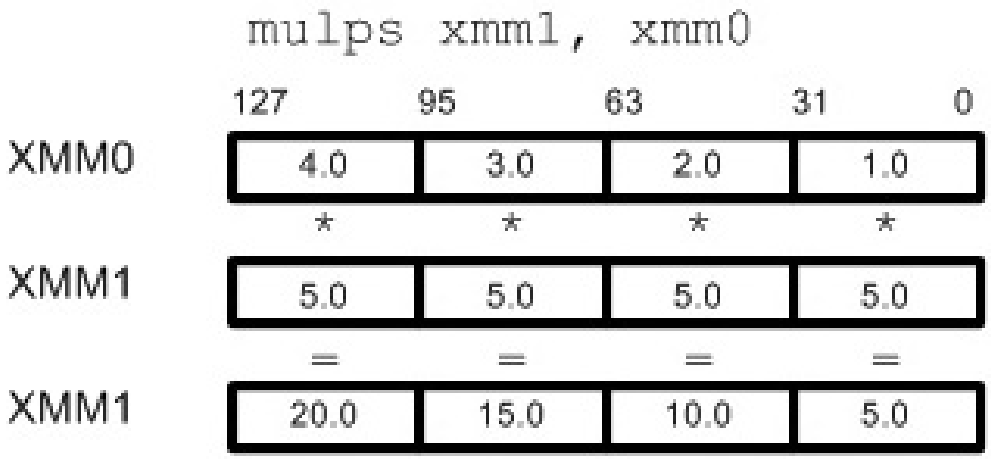
# Branch prediction & speculative execution

```
x = f(y);
if (x < 100) {
   x *= 2;
} else {
  x -= 100;
}
```



# Vector instructions

- SSE: 128-bit registers (4 $\times$ SP or 2 $\times$ DP)
- AVX: 256-bit registers (8 $\times$ SP or 4 $\times$ DP)



# CPUs have become complicated...what about GPUs?

# CPUs have become complicated...what about GPUs?

- Out-of-order execution: **No**
- Register renaming: **No**
- Branch prediction: **No**
- Speculative execution: **No**
- Cache: **Yes, but manual** (optional CPU-style cache available recently)
- Pipelining: **Yes**
- Vector registers: **Sort of...**

SM
Instruction Cache
Warp Scheduler
Warp Scheduler
Dispatch Unit
Dispatch Unit
Register File (32,768 x 32-bit)
CUDA Core
Dispatch Port
Operand Collector
FP Unit
INT Unit
Result Queue
Core
LD/ST
SFU
Interconnect Network
64 KB Shared Memory / L1 Cache
Uniform Cache
Tex
Texture Cache
PolyMorph Engine
Vertex Fetch
Tessellator
Viewport Transform
Attribute Setup
Stream Output