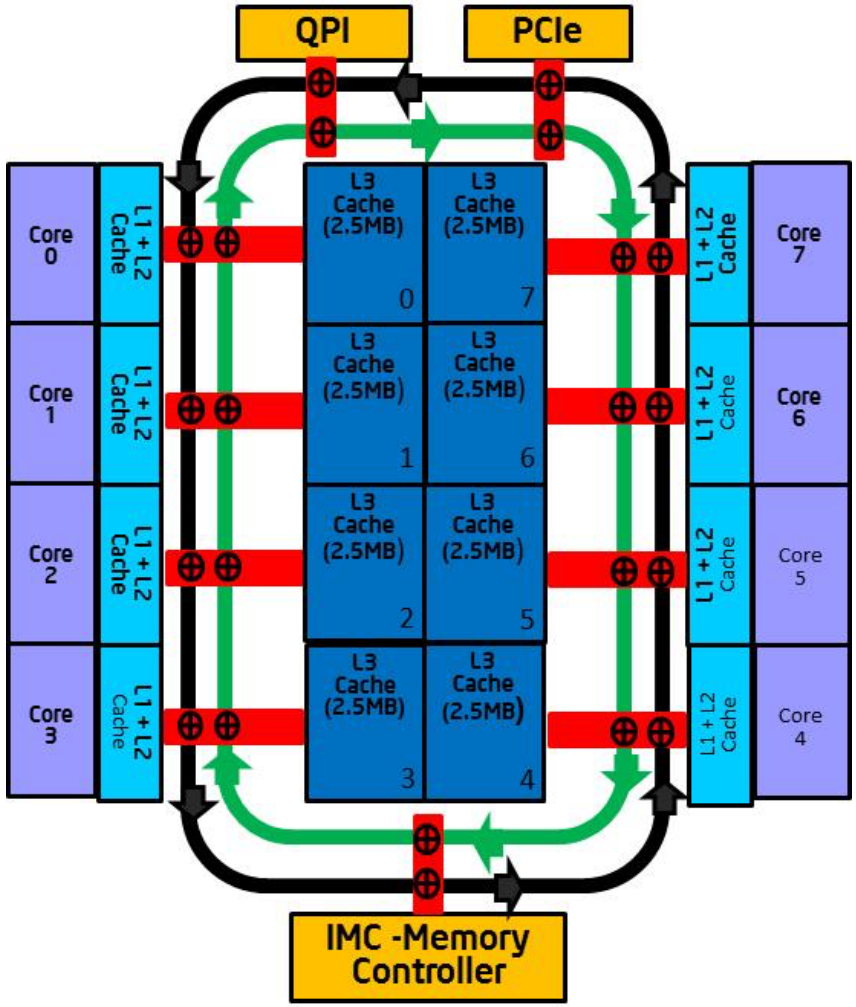
Optimized Scientific Computing:
Coding Efficiently for Real Computing Architectures

Noah Kurinsky

SASS Talk, November 11 2015

Processor Architecture

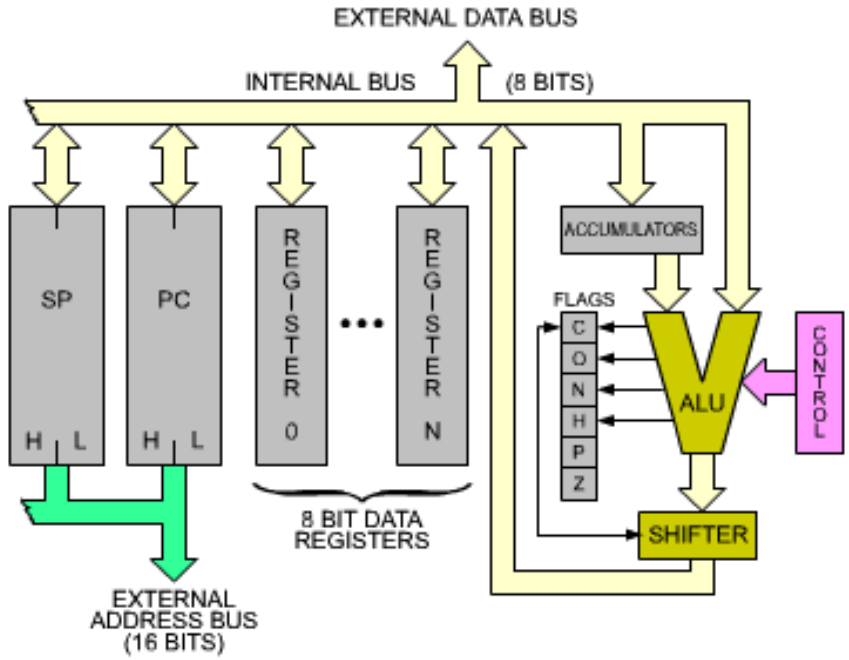
- A processor consists of memory, data-paths, and control units, as well as compute cores; all of these components can vary depending on the application of the processor.
- Note the different levels of memory on the right; these are on-chip caches, which can be accessed faster than RAM and disk, with distance a critical performance factor
- The compute core takes information from the data paths, processes it, and puts it back on the data paths. The design of the core depends on the type of data typically contained by these paths.



Basic Core Architecture

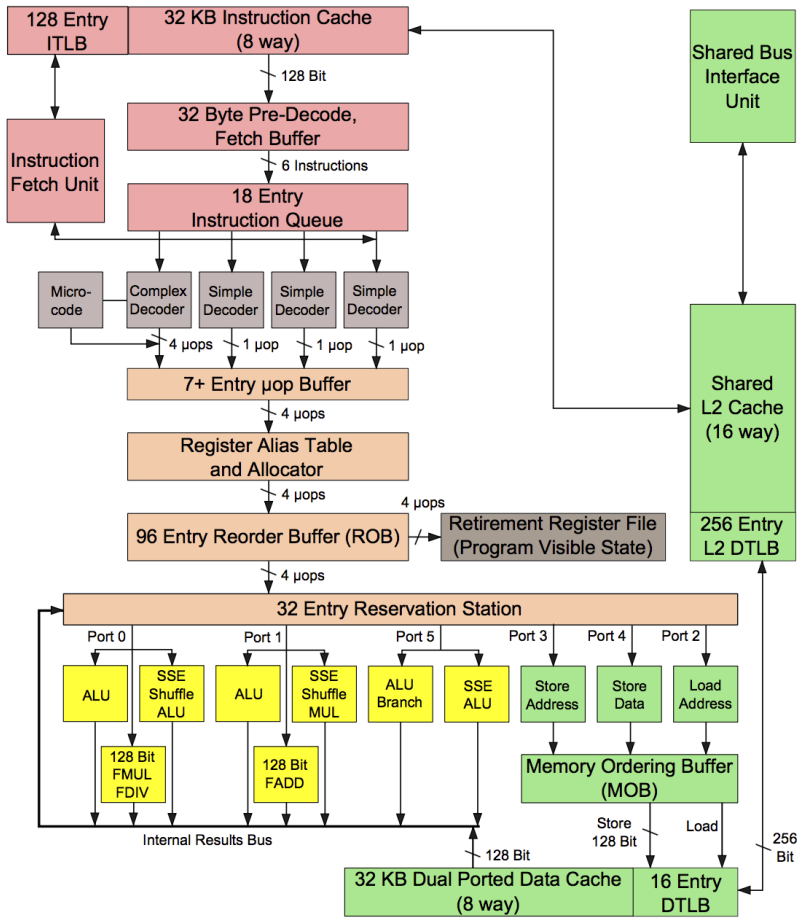
A compute core has the following essential components:

- Control unit - determines which computations to be done, puts data on lines or in registers
- Registers - a handful of memory locations close to the chip to load and store temporary results
- Accumulators - special registers for holding inputs and outputs of computations
- ALU (Arithmetic Logic Unit) - logic which computes functions on accumulators, storing result in an output register
- Processor Flags - used to indicate errors or signal results to the control unit



Intel Core Microarchitecture

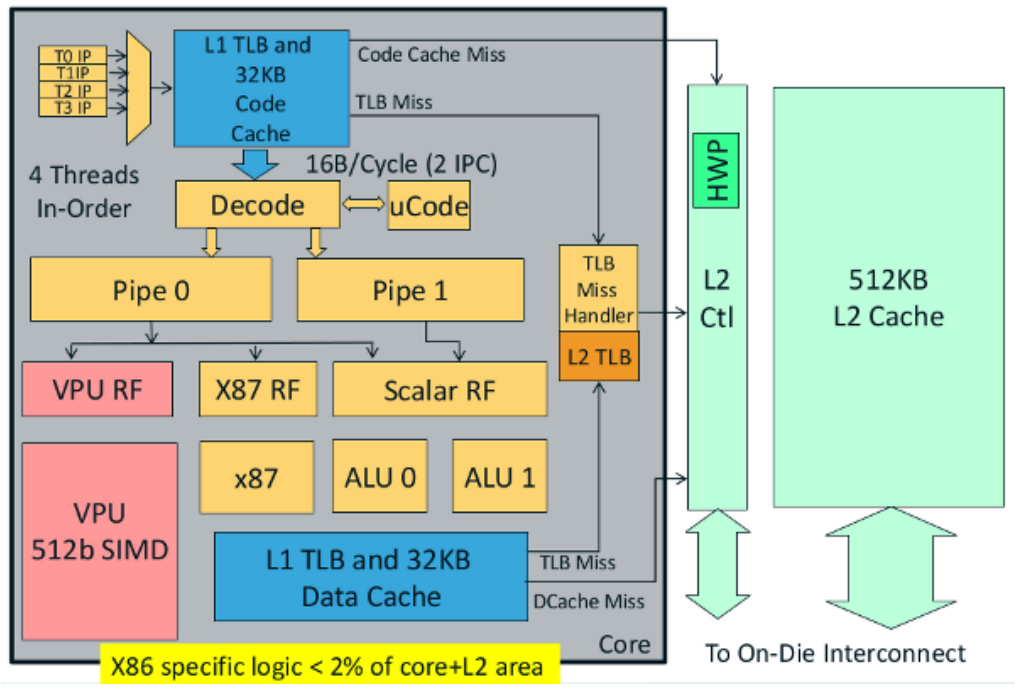
- A CPU core might contain parallel data paths and special ALUs for different operations
- The set of operations performed by a core comprise its instruction set, which describes the operation performed for each control input
- Hyper-threading allows multiple computations on a given chip, where the ALUs each run for different amounts of time
- Operating systems require a different set of operations than scientific computing, and thus these are very general purpose



Intel Core 2 Architecture

Intel MIC Microarchitecture (GPU-Like Processors)

- A GPU core assumes it and all other cores are performing a similar computation, and has one optimized data path
- The input is not a single register, but a group of registers called a vector. The data paths are much wider.
- We don't need to concern ourselves with interrupts or operating system features which reduce power consumption, we only are concerned with mathematical operations



Why is this Relevant to Physics?

- How you utilize your hardware, and how you optimize code, depends on the strengths of your hardware
- How do we know when numerical computations will be sped up by using GPUs/Parallel Processing?
- When is it worthwhile to move to a more machine-optimized language (such as C, Cuda, OpenMP) versus a high-level (Python/Julia/Matlab) implementation?
- More and more physicists utilize Monte Carlo simulation for blind analysis, and writing efficient simulation/analysis code will often determine how good your statistics can be
- Basic knowledge of computer architecture can produce much more efficient code, and allow you to work more quickly

Arithmetic Logic Unit

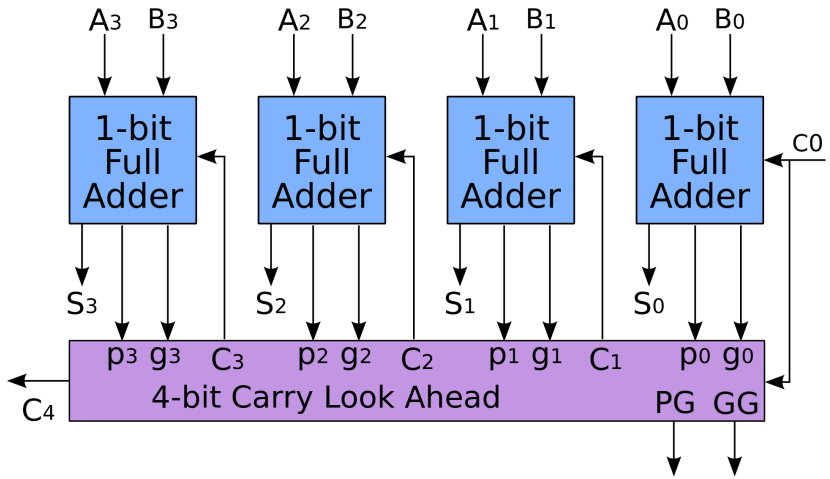
The ALU has classes of functions, and a corresponding instruction set which describes the speed of each function

- Boolean operations generally occur in one clock cycle, they only need to compare one bit from each of the input registers
- Data manipulation operations require the CPU to wait some characteristic time to retrieve data from memory, depending on the memory level
- Mathematical operations depend on how "deep" the logic network needs to be, with addition shallow and division very deep
- The most expensive functions are those which you can't do in your head (exponentiation, logarithms, and trig functions). Trig functions on the Xeon are 150-300 cycles!

The speed of an operation is characterized by "latency" and "throughput"; latency describes the time it takes to complete an operation, throughput describes the speed with which the ALU can reset from performing the operation

Addition and Subtraction

- A full-adder is a very simple and fast component
- Addition is fairly flat; the second path can be highly optimized.
- Addition almost always takes less than a few clock cycles
- Subtraction is easy as well, using a two's complement binary representation it is easy to negate numbers
- Double Precision on Intel Xeon: 3 cycles (5 for vectors), but one addition can be dispatched per cycle (functionally 1.5 cycle per operation)



Multiplication

- One layer per input bit, can be fairly optimized but not as easily as addition
- Has the same building block as the adder, so these diagrams can be directly compared
- Double Precision on Intel Xeon: 5 cycles (7 for vectors), but one multiplication can be dispatched per two cycles (functionally 2.5 cycle per operation)

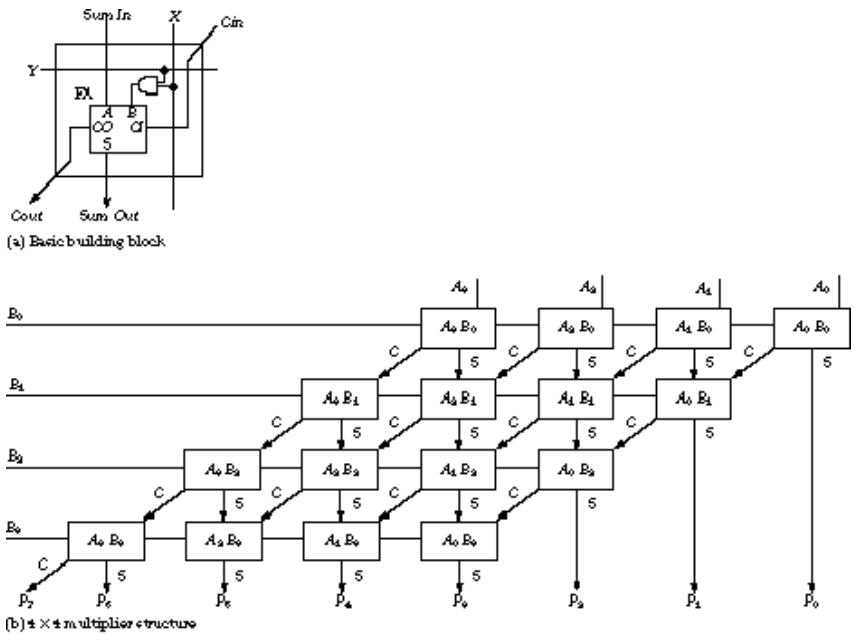
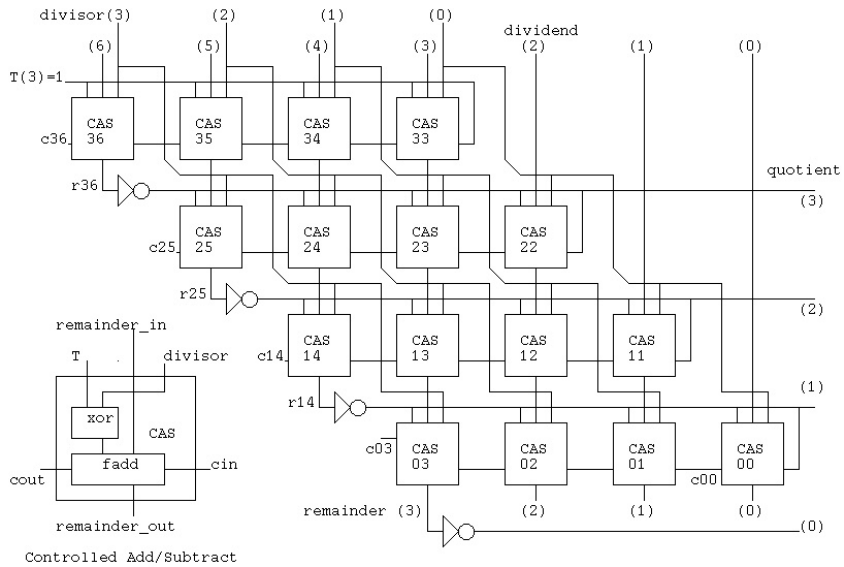


Figure 5.29 4 × 4 combinational multiplier.

Division

- Looks similar to multiplication, but the need for more data lines limits to one dispatch
- The building block is not a full-adder, it is much slower (uses an XOR not an AND)
- Takes up more room on chip, so often an iterative solution is used
- Double Precision on Intel Xeon: 34 cycles (69 for vectors), one dispatch per operation.



Variable Representation

Table 15-4. Unsigned Integer Division Operation Latency

	Dividend	Divisor	Quotient	Remainder	Cycles
DIV r8	AX	r8	AL	AH	25
DIV r16	DX:AX	r16	AX	DX	26-30
DIV r32	EDX:EAX	r32	EAX	EDX	26-38
DIV r64	RDX:RAX	r64	RAX	RDX	38-123

Table 15-5. Signed Integer Division Operation Latency

	Dividend	Divisor	Quotient	Remainder	Cycles
IDIV r8	AX	r8	AL	AH	34
IDIV r16	DX:AX	r16	AX	DX	35-40
IDIV r32	EDX:EAX	r32	EAX	EDX	35-47
IDIV r64	RDX:RAX	r64	RAX	RDX	49-135

The precision with which you store data greatly affects the latency of the operation, especially for division (above are division operations for Xeon processor, depending on bit count and whether signed)

Base 2 Operations

Many operations can (and often are) sped up by working in the natural base of computating, base two.

- Bitshifting is a unit-time operation, so multiplying and dividing by 2 is a single cycle
- If you can re-cast an expression in boolean logic, it will usually become a unit-time operation
- This is why bit-masking is used so often in intensive simulation
- Comparisons are also unit-time operations; if you can easily turn a problem into a "compare to 0", it will often speed up substantially
- Your compiler knows how to do this, more on that later

Accessing Memory Layers

As you go further from the processor, you have access to more layers of memory, but access time increases

- Immediate registers are 1 or 2 cycle access
- Further cache can be up to 100 cycles
- RAM and disk are even slower
- Modularize your programs so the computer can use as much cache as possible
- This is a give and take, using memory is a good alternative to very complex calculations

Working With Your Compiler

Many of the factors mentioned earlier can be taken care of with -O1 flag, for example in gcc, compiler options. In order for your compiler to be able to help optimize your program, you need to be specific

- If a variable is constant, tell the compiler (especially in C) by labeling it "const". This will affect where it is placed in memory, and which instruction is used
- Minimize use of global variables so that your compiler can use cache as efficiently as possible
- Minimize use of dynamic memory, you're making your life harder by putting your data further from the processor (STL classes are much preferable)
- Use well structured loops, which your compiler can help optimize, rather than recursive function calls, which also add overhead (and JMP commands which take a few cycles to complete)

One optimization you can make that your compiler can't is to reduce the size of your variables where possible, or use integers, or unsigned floats, to speed up computations. This will always save at least 1 cycle, but often can speed up computations by 2 or more

Multiply, Don't Divide

This deserves its own slide. If I change

```
for i in 0:1000{ x[i]/=3.4 }
```

to

```
for i in 0:1000{ x[i]*=0.29 }
```

I will see an almost 10 times speed increase (try it in C, it works). Note that your compiler might make the same optimization if you allow it to do so.

Imagine that you've pre-computed this overall factor, but it's not a constant, so your compiler can't simply invert it. You can do it yourself to speed up the computation.

Add, Don't Multiply

We can make a similar change with multiplcation. Suppose I make a histogram, and want to scale it; I can multiply all the entries by a scale factor:

```
for sample in samples{ i=getIndex(sample); x[i]++ }
for i in 0:1000{ x[i]*=scale }
```

to

```
weight=1.0/scale
for sample in samples{i=getIndex(sample);
x[i]+=weight }
```

I will see an almost 2 time speed increase.

Using Lookup Tables

Suppose you have a complicated calculation, such as (for the cosmologists) luminosity distance, or computing a function like the following:

$$f(x|b) = \sum_x \sum_b \frac{x(i)^j}{b(j)}$$

Suppose that you know that b is constant for some number of calculations (maybe this is a pre-fit function), and x takes on discrete values, maybe 1000 values total.
I need kilobytes of memory to store my information (L1 or L2 cache) and I can access a stored value in 10 cycles. It would take at least 60 cycles even for one b and one x to compute this function, so I have at least a 6 times (realistically 10-1000 times) increase.
If you need any discrete set of trig results, you should pre-compute them.

When to go Parallel

How do you know whether your application would work better on specific specialized hardware?

- Do you compute individual matrix elements which are independent of one another?
- Is your code very sequential (i.e. not a large amount of logical jumps)?
- Are your input data regular?
- Critically: Do you need to access memory regularly

Most likely, any job can be run on a GPU or Xeon Phi cluster, and almost any highly parallel computation will run faster, but any cross-talk between computations will slow down the entire calculation. Sometimes it isn't worth the effort, especially when you need to write to/from memory often and when you need to utilize CPU functions.
Remember that a GPU core is a crappy CPU core, and judge accordingly.

In Conclusion

- We've achieved orders of magnitude improvement in performance by simply changing which functions we use and in what order.
- We see that not all processes are created equally, and that CPUs are really machines with design considerations that affect their performance
- We now know why global variables are bad
- We now know why the compiler accepts so many annoying keywords
- This is the tip of the iceberg; check out a developer's handbook (they're 1000 pages long) to get more details on every operation the CPU can perform. These manuals have optimization suggestions specific to the processor, though many are already integrated into your compiler.
- These often apply only to lower-level languages; if you try this in python the overhead will swamp out the CPU time improvement (except in the case of lookup tables)!