your friend the computer (aka computer architecture)

unix weapons school

The purpose of a systems programmer

Given system S, and problem P,
we will implement P,
using the resources of S,
subject to some constraint.

That constraint is typically to minimize either:

- time to completion,
- power to completion, or
- time to completion $\cdot$ utilization.[1]

Proving that these constraints have been met is clearly dependent upon the details of both P and S.

[1] Constrained by some minimum utilization.

Chasing peak

How can we know when we're "done"? Absolute measurements can be taken in terms of the system's theoretical **peak** perf.

Peak is a slippery concept. What follows is my opinion:

- "Peak" seems most *precisely* defined in terms of the ISA of S (which might be irrelevant to our problem).
- "Peak" seems most *correctly* defined in terms of the best possible solution to P (which might be unknown).
- "Peak" seems most *productively* defined in terms of the instructions our actual implementation would use in the absence of structural hazards.

In my experience, it is generally misleading to compare how closely different P approach their own peaks. Most often, we use "peak" within the context of a small section of code.

Synchronous processors

Start in some initial state, and evolve in discrete timesteps according to some primitive recursive function closed on that state.

Each time step is a *cycle*(**c**). 1GHz $\Longrightarrow$ 1ns **c**-time (ϕ).

"Running a program" is the act of denoting words of this state as "current instructions", causing them to drive control flow.

- Static instructions: Instructions in a sequence
- Dynamic instructions: Instructions *executed* in a sequence
- IPC: Instructions per **c**
- CPI: **c** per instruction

Instructions lie at the boundary between those abstractions we control, and those we merely exploit.

Minimizing time to completion

Means of reducing P_{time} are finite:

$$P_{time} = Inst_{dynamic} \cdot CPI_{avg} \cdot \nu \tag{1}$$

1. Reduce **c**-time (better µarchitecture, better materials, $\uparrow V_{DD}$)
2. Require fewer instructions (SIMD, better code, better algorithms, more powerful instructions, AQC)
3. Reduce CPI (better µarch, better code, VLIW/EPIC)
4. Add time (MIMD)

#1 is an exhausted technique. Understanding why will require a detour into electrical engineering. Hold on tight!

c-time

Taken to the limit, we can run asynchronously (measured in FO4 inverter delays):

$$\phi = \phi_{logic} \tag{2}$$

Asynchronous circuits are hard, so we clock. Our clock period must be at least the maximum *logic time* (the sum propagation delays along those gates and traces composing the longest circuit in the system):

$$\mathbf{c} = \phi_{logic_{max}} \tag{3}$$

We'll want to preserve our logic's outputs, and use them as inputs. Latches have a setup time and a hold time[2], the sum of which constitutes latch delay:

$$\mathbf{c} = \phi_{logic_{max}} + \phi_{latch} \tag{4}$$

Our clock distribution network is less than ideal. We must allow for jitter[3] (divergence from the specified waveform[4]) and skew (differences in time of arrived signals):

$$\mathbf{c} = \phi_{logic_{max}} + \phi_{latch} + \phi_{jitter} + \phi_{skew} \tag{5}$$

---

[2] Also a contamination delay, which is critical to reading and writing a register in the same cycle.

[3] *Phase jitter* (absolute deviation), *period jitter* ($\Delta_n \frac{\phi_{phase}}{t}$), and ***c*-to-*c* jitter** ($\Delta_n \frac{\phi_{period}}{t}$).

[4] Typically a 50% duty-cycle square wave.

Moore's Law

The number of transistors on an IC doubles every ϕ_{Moore} (i.e., transistor packing grows exponentially). Chip area is decisively *not* growing exponentially, so transistor areas must be shrinking.

w00t! More transistors means wider SIMD, more integrated functionality, bigger on-die caches, more cores per physical package, or more physical packages per wafer (with implied price savings). Less area per transistor means less capacitance means less current required to switch means less time to switch (faster devices, w00t!) means less work done means less power drawn means less heat dissipated[5].

argh! Interconnect lines show greater resistance with reduced dimension. More transistors, everything else being equal, mean more power draw[6]. When channel length L $\sim$ depletion layer widths, *short-channel effects*[7] come into play. Less junction material amplifies the effect of dopant fluctuations[8]. Electrons tunnel more easily across the oxide from gate to channel. Voltages must be reduced to combat HCI, but carrier velocity $\nu = \mu_{SC} E = \frac{\mu_{SC} V_{DS}}{L} \implies t = \frac{L}{\nu} = \frac{L^2}{\mu_{SC} V_{DS}}$ (slower devices, argh!).

But mainly w00t. Thanks, CMOS technologists!

---

[5] Which itself means lower-power, more reliable, faster (w00t!) devices.

[6] This can be more than offset by needing fewer cycles for a given problem.

[7] Effects like DIBL, velocity saturation, impact ionization, surface scattering, and dread hot carrier injection.

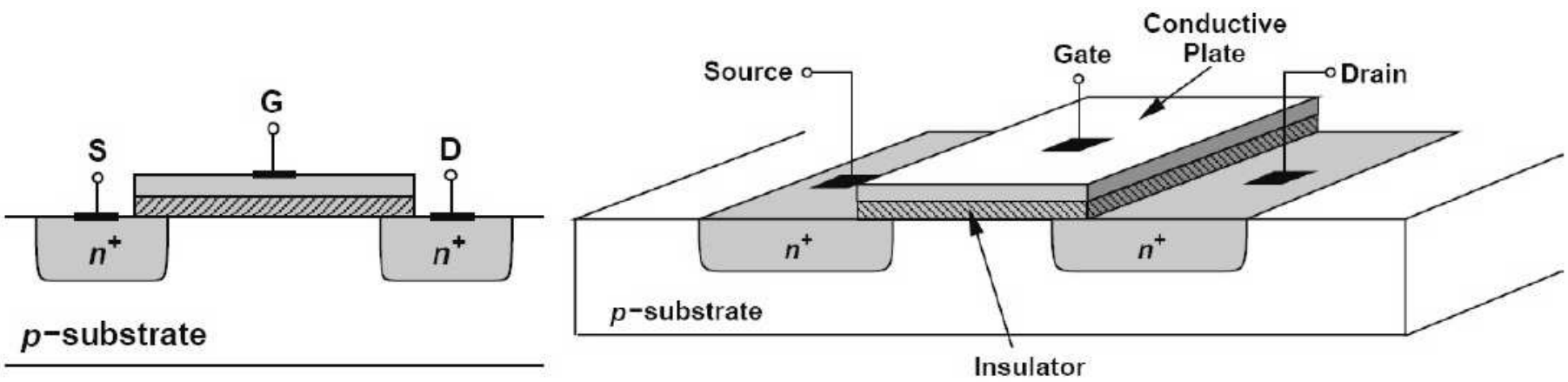
[8] Though dopant density goes up to combat CLM and thus DIBL.

Metal oxide semiconductors I

Let's calculate negative MOS. With no mobile carriers, $I_{D_C} = 0$. V_{GS}^{+} is applied to the gate, attracting electrons as V_{GS} exceeds the threshold voltage V_{TH}, establishing a salient. $I_{D_{TL}} = \frac{\mu_n C_{oxn}^* W}{2L}\left[2(V_{GS} - V_{TH})V_{DS} - V_{DS}^2\right]$, growing linearly with V_{GS}. Once $V_{D_{Sat}} \triangleq V_{GS} - V_{D_{TH}} \geq V_{DS}$, a conducting channel exists from source to drain, and $I_{D_{Sat}} = \frac{\mu_n C_{oxn}^* W}{2L}[V_{GS} - V_{TH}]^2$. We call these three modes *cutoff*, *linear*, and *saturation*. This switch requires discharging the charge in our nMOSFET:

$$\tau_{pHL} \approx \frac{\text{charge on } C_L}{2 * \text{NMOS discharge current}} = \frac{L_n C_L V_{DD}}{W_n \mu_n C_{ox}^* (V_{DD} - V_{TH})^2} \tag{6}$$

Where appropriate, pMOS reverses the signs[9].



[9] Using holes, not e^+ (positrons).

Metal oxide semiconductors II

Drain and source are doped opposite the substrate. L is gap between source and drain. W is width of source/gate/drain[10]. $T_{ox} < 2$nm. Circuit speed increases with increasing I_{on} which increases with decreasing V_{TH}.

nMOS

- V_{GS}, V_{DS}, I_D are positive
- Gate voltage V_{GS} increases
- Negative doping (n+)
- e^- is charge carrier
- Current flows when output is low
- Carries a strong 0 and a weak 1
- Bulk connected to ground
- I_D is drain-to-source

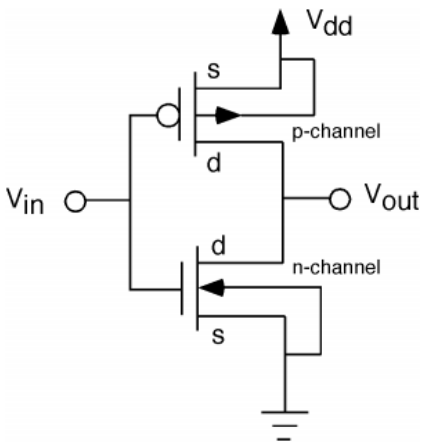
pMOS

- V_{GS}, V_{DS}, I_D are negative
- Gate voltage V_{GS} decreases
- Positive doping (p+)
- Holes are charge carrier
- Current flows when output is high
- Carries a strong 1 and a weak 0
- Bulk connected to supply (V_{DD})
- I_D is source-to-drain

[10] W/L is known as the *drain current capability*.

CMOS Inverter

We'd like timing and other properties to be independent of whether we're going high or low, and not to draw power in a steady state. *Symmetric CMOS inverters* ($V_{T_n} = |V_{T_p}|$ and $K_n = K_p$ ($K_{foo,ox} \triangleq \frac{\mu_{foo}\kappa_{ox}}{T_{ox}}$ ($\mu =$ electron mobility in foo, $\kappa =$ relative permittivity in ox, and $T =$ thickness of insulator oxide ox))) draw power only while switching, maximize use of clocks, and require only a pMOS and nMOS transistor in series.



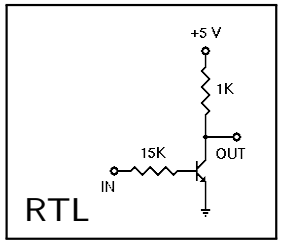
$V_{IN} = 0 \implies V_{OUT} = V_{DD}$

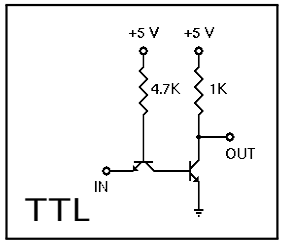
- $V_{GS_n} = 0 < V_{TH_n}$ $\implies$ nMOS off
- $V_{SG_p} = V_{DD} > -V_{TH_p}$ $\implies$ pMOS on

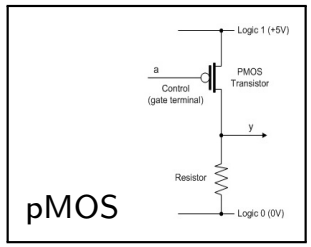
$V_{IN} = V_{DD} \implies V_{OUT} = 0$

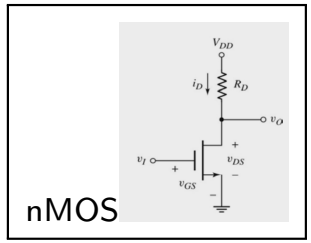
- $V_{GS_n} = V_{DD} > V_{TH_n}$ $\implies$ nMOS on
- $V_{SG_p} = 0 < -V_{TH_p}$ $\implies$ pMOS off

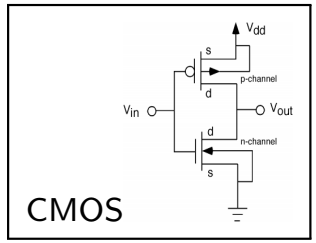
Digital logic



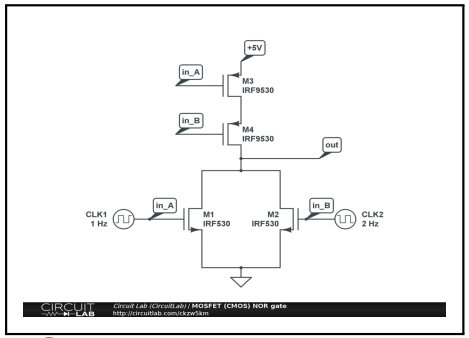


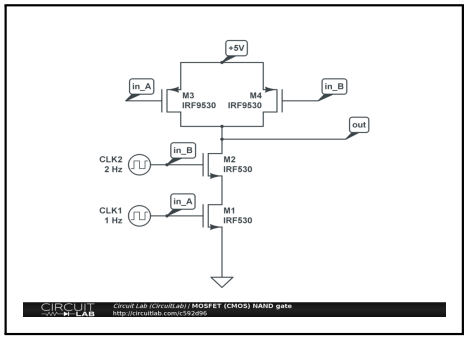




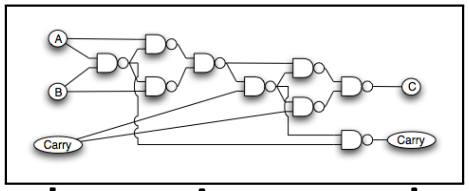


Given inverters, we can build (*functionally complete*) NORs and NANDs[11]...





...given NORs and NANDs, we can build universal computers...



...and now we're getting somewhere. Delicious!

[11] CMOS NOR/NAND in 4 transistors each; TTL multiemitters requires 5 for NOR vs. 4 for NAND.

CMOS dynamic power

Draw *greatest* power only when changing state...[12]

$$P_{dynamic} = \alpha C_L V_{DD}^2 \nu \tag{7}$$

P Dynamic power ($W = J/s = \frac{m^2 \cdot kg}{s^3}$)

C_L Load capacitances ($F = \frac{A^2 \cdot s^4}{m^2 \cdot kg}$)

α Activity factor

V_{DD} Supply voltage ($V = \frac{m^2 \cdot kg}{A \cdot s^3}$)

ν Frequency ($1/s$)

NB: $P_{dynamic}$ increases with the *square* of V_{DD}.

[12] Though, as $\nu \to \infty$, $\frac{T_{switching}}{T_{total}} \to 1$...

CMOS static power

. . . but always consume *some* power:

$$P_{static} = I_{static} \cdot V_{DD} \tag{8}$$

P Static power ($W = V \cdot A$)

I_{static} Static current (I_{subth}[13] $+ I_{tunnel}$[14] $+ I_{leakage}$[15], A)

V_{DD} Supply voltage (V)

This has become more important as transistor count has increased and sizes have shrunk, both for reasons mentioned earlier and because dynamic power has fallen.

Implication: power draw is independent of actual state.
Implication: for fixed work, lower ν usually cannot save power.

[13] Leakage between MOS source and drain
[14] Leakage across semiconductor junctions
[15] Leakage through the insulator

10GHz: don't hold your breath

Why do we care?

General purpose processors won't see large ν increases[16].

Recall that **c**-time $> \phi_{logic_{max}}$. Reducing the maximum logic delay (without sacrificing computational power per instruction) generally requires pipelining. As we add pipeline stages, we lose more and more **c** to |stage|-bounded delays, especially branch mispredictions.

Power requirements for clock distribution increase linearly with frequency. As does dynamic power draw. As does CPI. Jitter tolerances are approached more closely. These lost **c** are not free.

Frequency cannot hide non-computational delay. A 800ns off-die access is 800ns no matter one's frequency.

[16] In the absence of serious overclocking, anyway.

Instruction set architecture

Fully generally, instructions map input states to output states.

- A machine could contain in ROM a lookup table containing every possible function from $2^{n*\text{word}} \implies 2^{m*\text{word}}$. Each instruction specifies inputs, outputs, and a map selection[17].
- A machine could allow each instruction to specify a single lookup table implementing a single such map[18].
- "Arithmetic MISC" provides universal computation via a single arithmetic instruction operation, memory addresses, and a conditional branch target in every instruction.
- "Transport-triggered MISC" (sometimes called ZISC) involves memory-mapped functional units and a branch. All MISC schemes require the ability to modify one's own code[19].

[17] Calculate this ROM's size, and the minimum instruction size.
[18] Calculate the minimum necessary instruction size.
[19] Prove this.

Special registers

Often read-only/privileged/require special instructions.

- EIP: Read-only. Instruction pointer
- $EFLAGS$: Flags register. Contains IOPL/CPL (`POPF`/`IRET`)
- CR_0: Machine Status Control Word (`LMSW`, 286+)
- CR_2: Read-only. Page fault addresses (386+)
- $CR_3 (PDBR)$: Page Directory Base Reg. (386+)
- $MXCSR$ SSE Control Status Reg. (`LD/STMXCSR`, PIII+)
- DR_0, DR_1, DR_2, DR_3, DR_7: Debug Reg.
- CS, DS, ES, FS, GS, SS: Segment Reg.
- TR: Task Register (`LTR`/`STR`)
- $GDTR$: Global Descriptor Table Reg. (`LGDT`/`SGDT`, 286+)
- $LDTR$: Local Descriptor Table Reg. (`LLDT`, 286+)
- $IDTR$: Interrupt Descriptor Table Reg. (`LIDT`/`SIDT`, 286+)
- TPR: Task Priority Reg.
- PPR: Process Priority Reg.
- TSC: Timestamp Counter (`RDTSC`, Pentium+)
- PMC_n: Performance Monitoring Counters (`RDPMC`, MMX+)
- MSR_n: Model-Specific Reg. (`RD/WRMSR`, Pentium+)

General-purpose registers

- Read ports pace superscalability
- Write ports pace instruction retire
- Register width paces bit parallelism
- Architectural registers pace memory accesses
- Physical registers pace OOO hiding of false dependency

Register file (indexed access)

- Architectural registers are exposed as PRAM or a stack
- SRAM cells + read/write lines + decoder tree + sense amps
- One internal bit line per bit of read port
- Two internal bit lines per bit of write port
- One word line per entry
- Transistor area grows linearly with number of ports
- Wire pitch area grows with square of number of ports
- Hence: register *banking*

Fundamental theorem of data starvation

We can only hit peak during sequences of code which fully utilize arithmetic resources. Registers must provide throughput at least equivalent to the processor's arithmetic throughput.[20]

THUS, a memory lacking throughput greater than or equal to the ratio of arithmetic/register throughput to arithmetic intensity (**c** per word) will not be able to sustain peak.

[20] In a classic load/store RISC architecture, arithmetic instructions operate only on registers. In a machine admitting memory addresses as inputs to arithmetic operands, such instructions will exhibit greater latency and lesser throughput than their register-direct counterparts (though not always; see Intel NetBurst/Core "μ-fusion").

FRONTEND

Sandy Bridge frontend

The modern x86 frontend is tremendously complicated[21] due to its inherited ISA, yet it manages to deliver µops to the superscalar dataflow-ordered execution core at full throughput. How?

- 16B/**c** fetches from 32KB 8-way[22] I$
- Speculative decoding at each of fetch's 16B
- 1 complex decoder, 3 simple decoders
- 32 sets of 8 ways of 6 µop µI$ ("decoded I$")
- 28µop µL$ ("Loop stream detector") per thread[23]
- 4µop/**c** renamer-scheduler
- Macro- and µ-fusion

[21] Indeed, it is this dense frontend which made things like 31-stage pipelines even feasible (Northwood's "execution" cycle was 17 of 20). The Loop Stream Decoder (present since Nehalem) effectively shortens the pipeline by bypassing the decoding frontend.

[22] 4-way on Nehalem

[23] On HyperThreaded Ivy Bridge, 56 µops if one logical core is inactive.

Frontend delays

- Instruction fetch miss (7**c** for ITLB miss + 2TLB hit)
- 3**c** for each non-REX length changing prefix[24]
- Instruction decode delay / MSROM access
- Reservation stations full / ROB full
- Self-modifying code flushes all pipelines/cache

We cannot have more instructions in flight than we have ROB entries. We cannot have more instructions for a given set of ports in flight than we do reservation stations at those ports.

[24] 6**c** for any fetch containing an LCP on Nehalem

Superscaler

An ALU contains numerous functional units—address generation, shifting, basic arithmetic, etc. The transistor budget for functional logic is typically dwarfed by that of cache[25].

It would be reasonable, then, to execute multiple instructions at a time (not staggered, as in a pipeline, but truly synchronously). This ought be possible so long as

- There are no true dependencies between the instructions, and
- The instructions use different functional units

Analyzing the instruction stream for such dependencies would add to our $\phi_{logic_{max}}$, increasing **c**-time and reducing frequency[26].

Attempts—EPIC/VLIW—were made to have the compiler perform this analysis itself. Compiler technology was (is?) not up to the task, due largely to the difficulty of modeling varied caches.

[25] This is not true on manycore architectures such as NVIDIA's CUDA.

[26] There would also be a cost in transistors, of course.

BACKEND

Pipelines

In and of itself, pipelining does not improve performance. A scaler pipelined processor continues to retire, at best, 1 instruction per **c**.

The advantage of pipelining is that, by reducing the number of gates and traces a signal travels *in a* ***c***, we reduce $\phi_{logic_{max}}$, and can thus increase ν, increasing our available **c** per unit time. Also, it can hide absolute delays[27], such as the dependency analysis necessary for compiler-oblivious superscaler operation, or the complicated decoding necessary for x86 instructions.

We pay[28] for these extra **c** whether we can exploit them or not. The effects of delays are amplified; a 400ns off-die DRAM access blocks for 200**c** at 500MHz, but 1200**c** at 3GHz.

The techniques by which the execution plane is kept full are collectively known as out-of-order execution.

[27] Upon reaching steady state.

[28] When the processor is running at full frequency, anyway.

The dataflow limit

Can we describe an ideal von Neumann/Harvard machine?

- Infinitely many registers
- Infallible branch prediction
- Infallible dealiasing
- Single-cycle, non-blocking caches
- Infinitely many issues/retires per **c**

This processor will proceed governed only by true dependencies; we call this the *dataflow rate*[29].

[29] *Value prediction* has been studied to exceed the dataflow rate. It's not very awesome.

Out of order execution

- *Register renaming* eliminates WAW/WAR hazards
- *Branch prediction* generates branch statuses earlier
- *Target prediction* generates branch targets earlier
- *Speculative execution* performs work based on predictions
- *Predication* is branchless conditional execution
- *Disambiguation* allows loads and stores to be reordered

Backend delays

- Store forwarding stalls / memory intermediates
- Data migration across execution domains
- False dependencies due to flag regs / partial regs
- Retirement station bandwidth exceeded

Why we fall short of peak

Cause	Penalty	Ameliorations
Reduced-throughput instructions	$\leq$ ALU pipeline length	Str. reduction / pipeline ALU
L1 cache hit	Very few **c**	Multiport/pipelined L1
False dependency	Handful of **c**	Register renaming
True / unrenamable dependency	Handful of **c**	Value prediction, OOO
Decoding/execution/retirement stalls	Handful of **c**	More/better hardware
Startup / pipeline flush	Pipeline length	Shorten pipeline
Branch misprediction	~Pipeline length	Branch prediction, µcaches, predication, shorten pipeline
L1 D$ miss (L2 D$ cache hit)	$\leq$ L2 time ($<$10**c**)	Faster L2, bigger L1, non-temporal L/S, OOO
L1 I$ miss (L2 I$ cache hit)	L2 time ($<$10**c**)	Faster L2, bigger L1
Procedure call	Dozens of **c**	Register windows, inlining
System call	Hundreds of **c**	Call gates
L2 cache miss	Memory access time	Bigger/better L2
Thread switch		
TLB miss		
Process switch		
Access resident page via MMU	Hundreds of **c**	Larger caches, better bus
Bus lock (atomics, UC accesses[30])	Bus + exclusion	Coherence protocols
Access faulted page via DMA+MMU		
Load via FSB PIO		
Mutual exclusion	Arbitrary	Transactional memory, RCU

[30] Including pagewalks of uncacheable tables.

Optimization methodology

Are we hitting peak? If not, let's account for each cycle:

Are we fully utilizing the system?

- No. Why not?
  - Contention among threads?
  - Blocking on I/O?
  - Incomplete exploitation of multiple processors / SIMD?

Yes. Are we issuing µops?

- No. What's causing our frontend stall?
  - Store-forwarding?
  - LCP delay?
  - Cache miss?

Yes. Are we retiring µops?

- No. What's clogging our backend?
  - Branch mispredictions?
  - Dependency chains?

Yes. Look for strength reduction or new algorithms.

Recommended reading

- Andi Kleen. "Linux multi-core scalability" (2009).
- Doug Carmean and Eric Sprangle. "Increasing Processor Performance by Implementing Deeper Pipelines" (2002).
- Ulrich Drepper. "What Every Programmer Needs to Know About Memory" (2007). Linux Weekly News, in eight parts.
- Paul McKenney. "Transactional Memory Everywhere" (2012).
- Ward and Halstead. "Computation Structures" (1990).
- Fisher et al. *Embedded Computing: A VLIW Approach* (2004).
- John Shen and Mikko Lipasti. *Modern Processor Design* (2004) and "Exceeding Dataflow Limit via Value Prediction" (1996).
- Agner Fog. *The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers* (regularly updated). http://www.agner.org.
- Bruce Shriver and Bennett Smith. *The Anatomy of a High-Performance Microprocessor: A Systems Perspective* (1998).
- *Intel 64 and IA-32 Architectures Optimization Manuals*.

Hack on!

"The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise."

—Edsger Dijkstra

"The most amazing achievement of the computer software industry is its continuing cancellation of the steady and staggering gains made by the computer hardware industry." —Henry Petroski

"First you learn the value of abstraction. Then you learn the cost of abstraction. Then you're ready to engineer."

—Kent Beck