CMPE110 Lecture 03
Performance II

Heiner Litz

https://canvas.ucsc.edu/courses/19290

Announcements

- Find your group members
- Quizzes: Every Friday starting next week
- DRC, please provide academic access letters

Review

Performance: Latency vs Throughput

- Latency or response/execution time
  - How long it takes to do a task
- Throughput
  - Total work done per unit time (e.g., queries/sec)
- Questions
  - Example of cases where we care about one or the other?
  - Which one improves by speeding up a core?
  - Which one improves by adding more cores?
  - Does improving latency help improve throughput?
  - Does improving throughput help improve latency?

Performance Summary

$$\textit{Execution Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

- Must take all 3 factors into account
  - Caveat: we will add OS time and I/O time later on
- Performance depends on
  - Algorithm: affects IC and (possibly) CPI
  - Programming language: affects IC and CPI
  - Compiler: affects IC and CPI
  - Instruction set architecture: affects IC and CPI
  - HW design: affects CPI and Frequency

CPI Example

- Two alternative code using instructions types A, B, C

Class	A	B	C
CPI for class	1	2	3
IC in sequence 1	2	1	2
IC in sequence 2	4	1	1

- CPI_1 =

- CPI_2 =

CPI Example

- Two alternative code using instructions types A, B, C

Class	A	B	C
CPI for class	1	2	3
IC in sequence 1	2	1	2
IC in sequence 2	4	1	1

- $CPI_1 = (1*2 + 2*1 + 3*2)/5 = 10/5 = 2$

- $CPI_2 = (1*4 + 2*1 + 3*1)/6 = 9/6 = 1.5$

Relative Performance

- Define Performance = 1/Execution Time
- "X is n time faster than Y" means

$$\text{Performance}_X / \text{Performance}_Y$$
$$= \text{Execution time}_Y / \text{Execution time}_X = n$$

- Example:
  - Program runs for 10s on machine A, for 15s on B
  - Execution TimeB / Execution TimeA = 15s / 10s = 1.5
  - So A is 1.5 times faster than B
  - Or A is 50% faster than B

Performance Example

- Processors A and B for the same ISA
  - A: cycle time= 250ps, CPI = 2.0
  - B: Cycle time= 500ps, CPI = 1.2
- Which is one faster, and by how much?

Performance Example

- Processors A and B for the same ISA
  - A: cycle time= 250ps, CPI = 2.0
  - B: Cycle time= 500ps, CPI = 1.2
- Which is one faster, and by how much?

$$\text{CPU Time}_A = \text{Instruction Count} \times \text{CPI}_A \times \text{Cycle Time}_A$$
$$= I \times 2.0 \times 250\text{ps} = I \times 500\text{ps}$$

A is faster…

$$\text{CPU Time}_B = \text{Instruction Count} \times \text{CPI}_B \times \text{Cycle Time}_B$$
$$= I \times 1.2 \times 500\text{ps} = I \times 600\text{ps}$$

$$\frac{\text{CPU Time}_B}{\text{CPU Time}_A} = \frac{I \times 600\text{ps}}{I \times 500\text{ps}} = 1.2$$

…by this much

Power and Energy

Why Are Power & Energy Important?

- Power density (cooling)
  - Limits compaction & integration
  - E.g., a cellphone chip cannot exceed 1-2W
- Battery life for mobile devices
- Reliability at high temperatures
- Cost
  - Energy cost
  - Cost of power delivery, cooling system, packaging
- Environmental issues
  - IT responsible for 0.53 billion tons of CO_2 in 2002

Power Consumption in Chips

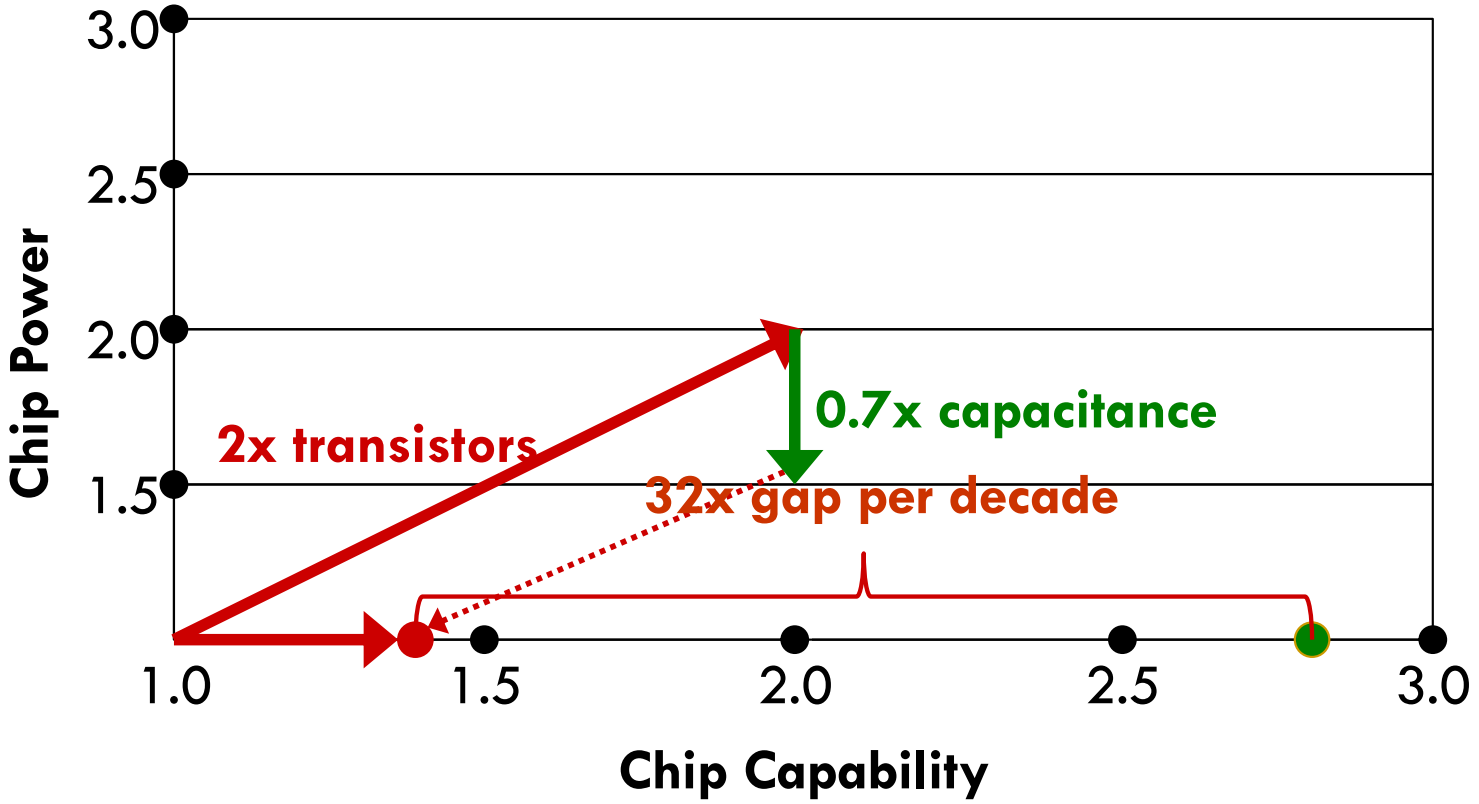
$$\text{Power} = C * Vdd^2 * F_{0\rightarrow1} + Vdd * I_{leakage}$$

- Dynamic or active power consumption
  - Charging and discharging capacitors
  - Depends on switching transistors and switching activity
- Leakage current or static power consumption
  - Leaking diodes and transistors
  - Gets worse with smaller devices and lower Vdd
  - Gets worse with higher temperatures

Energy (≠ Power)

- Energy = Average power * Execution time
  - Joules = Watts * sec
  - Power is limited by infrastructure (e.g., power supply)
  - Energy: what the utilities charge for or battery can store
- You can improve energy by
  - Reducing power consumption
  - Or by improving execution time
- Race to halt!

Semiconductor Scaling: The Present



- Moore's Law without Dennard scaling
  - 1.4x in chip capability per generation at constant power
  - 32x capability gap compared to past scaling

Question

- So, what do we do now?
  - Remember: $\mathbf{Power = C * Vdd^2 * F_{0\rightarrow1} + Vdd * I_{leakage}}$
- What should we optimize processors for?

Question

- So, what do we do now?
  - Remember: **Power = C *Vdd^2*$F_{0\rightarrow1}$ + Vdd*$I_{leakage}$**
- What should we optimize processors for?
  - Answer: energy per instruction (EPI)

$$Power = \frac{energy}{second} = \frac{energy}{instruction} \times \frac{instructions}{second}$$

  - After minimizing EPI, tune performance & power as needed
    - Higher for server, lower for cellphone

Useful Techniques for Evaluating Efficiency

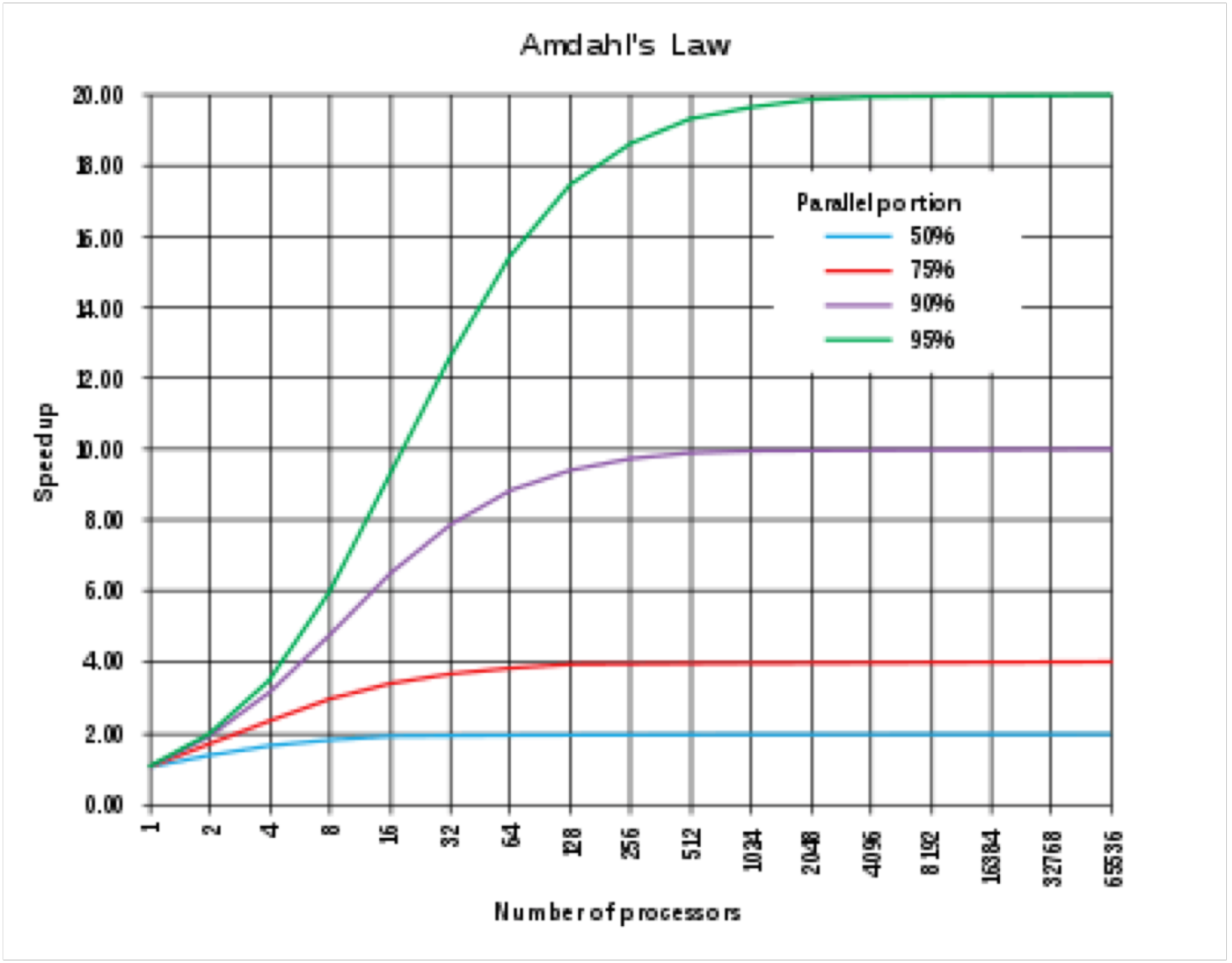
Amdahl's Law: Make Common Case Efficient

- Given an optimization x that accelerates fraction f_x of program by a factor of S_x, how much is the overall speedup?

$$Speedup = \frac{CPUTime_{old}}{CPUTime_{new}} = \frac{CPUTime_{old}}{CPUTime_{old}[(1-f_x)+\frac{f_x}{S_x}]} = \frac{1}{(1-f_x)+\frac{f_x}{S_x}}$$

- Lesson's from Amdahl's law
  - Make **common cases** fast: as $f_x \rightarrow 1$, speedup$\rightarrow S_x$
  - But don't overoptimize common case: as $S_x \rightarrow \infty$, speedup$\rightarrow 1 / (1-f_x)$
    - Speedup is limited by the fraction of the code accelerated
    - Uncommon case will eventually become the common one
- Amdahl's law applies to cost, power consumption, energy …

Amdahl's Law


Amdahl's Law
Speedup
20.00
18.00
16.00
14.00
12.00
10.00
8.00
6.00
4.00
2.00
0.00
Parallel portion
50%
75%
90%
95%
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
Number of processors

Benchmarks

- Programs used to measure performance
  - Supposedly typical of actual workload
- Benchmark suite: collection of benchmarks
  - Plus datasets, metrics, and rules for evaluation
  - Plus a way to summarize performance in one number
- Examples
  - SPEC CPU2006 (integer and FP benchmarks)
  - TPC-H and TCP-W (database benchmarks)
  - EEMBC (embedded benchmarks)
- Warning
  - Different benchmarks focus on different workloads
  - All benchmarks have shortcomings
  - Your design will be as good as the benchmarks you use

Example: CINT2006 for Intel Core i7 920

Description	Name	Instruction Count x 10^9	CPI	Clock cycle time (seconds x 10^{-9})	Execution Time (seconds)	Reference Time (seconds)	SPECratio
Interpreted string processing	perl	2252	0.60	0.376	508	9770	19.2
Block-sorting compression	bzip2	2390	0.70	0.376	629	9650	15.4
GNU C compiler	gcc	794	1.20	0.376	358	8050	22.5
Combinatorial optimization	mcf	221	2.66	0.376	221	9120	41.2
Go game (AI)	go	1274	1.10	0.376	527	10490	19.9
Search gene sequence	hmmer	2616	0.60	0.376	590	9330	15.8
Chess game (AI)	sjeng	1948	0.80	0.376	586	12100	20.7
Quantum computer simulation	libquantum	659	0.44	0.376	109	20720	190.0
Video compression	h264avc	3793	0.50	0.376	713	22130	31.0
Discrete event simulation library	omnetpp	367	2.10	0.376	290	6250	21.5
Games/path finding	astar	1250	1.00	0.376	470	7020	14.9
XML parsing	xalancbmk	1045	0.70	0.376	275	6900	25.1
Geometric mean	–	–	–	–	–	–	25.7

Other Benchmark Suites

- PARSEC
  - Focus on Multicore/Multithreaded
- STAMP
  - Transactional Memory
- Cloudsuite
  - Datacenter applications
- Microbenchmarks
  - FIO (block I/O)
  - Stream (memory bandwidth)
  - LINPACK (HPC, embarrassingly parallel)

Summarizing Benchmarks

Arithmetic mean $\frac{1}{n}\sum_{i=1}^{n} T_i$ — Use with times, not with rates

Represents total execution time

Harmonic mean $\frac{n}{\sum_{i=1}^{n} \frac{1}{R_i}}$ — Use with rates not with times

Geometric mean $\left(\prod_{i=1}^{n} \frac{T_i}{T_{ri}}\right)^{\frac{1}{n}} = \exp\left(\frac{1}{n}\sum_{i=1}^{n} \log\left(\frac{T_i}{T_{ri}}\right)\right)$ — Good with normalized performance

Does not represent total execution time

- Can also use weighted version
- Be careful which one you use!

Example: SPECpower_ssj2008 for Xeon X5650

- Power/performance benchmark

$$\text{Overall ssj\_ops per Watt} = \left(\sum_{i=0}^{10} \text{ssj\_ops}_i\right) \Big/ \left(\sum_{i=0}^{10} \text{power}_i\right)$$

Target Load %	Performance (ssj_ops)	Average Power (Watts)
100%	865,618	258
90%	786,688	242
80%	698,051	224
70%	607,826	204
60%	521,391	185
50%	436,757	170
40%	345,919	157
30%	262,071	146
20%	176,061	135
10%	86,784	121
0%	0	80
Overall Sum	4,787,166	1,922
Σssj_ops/Σpower =		2,490

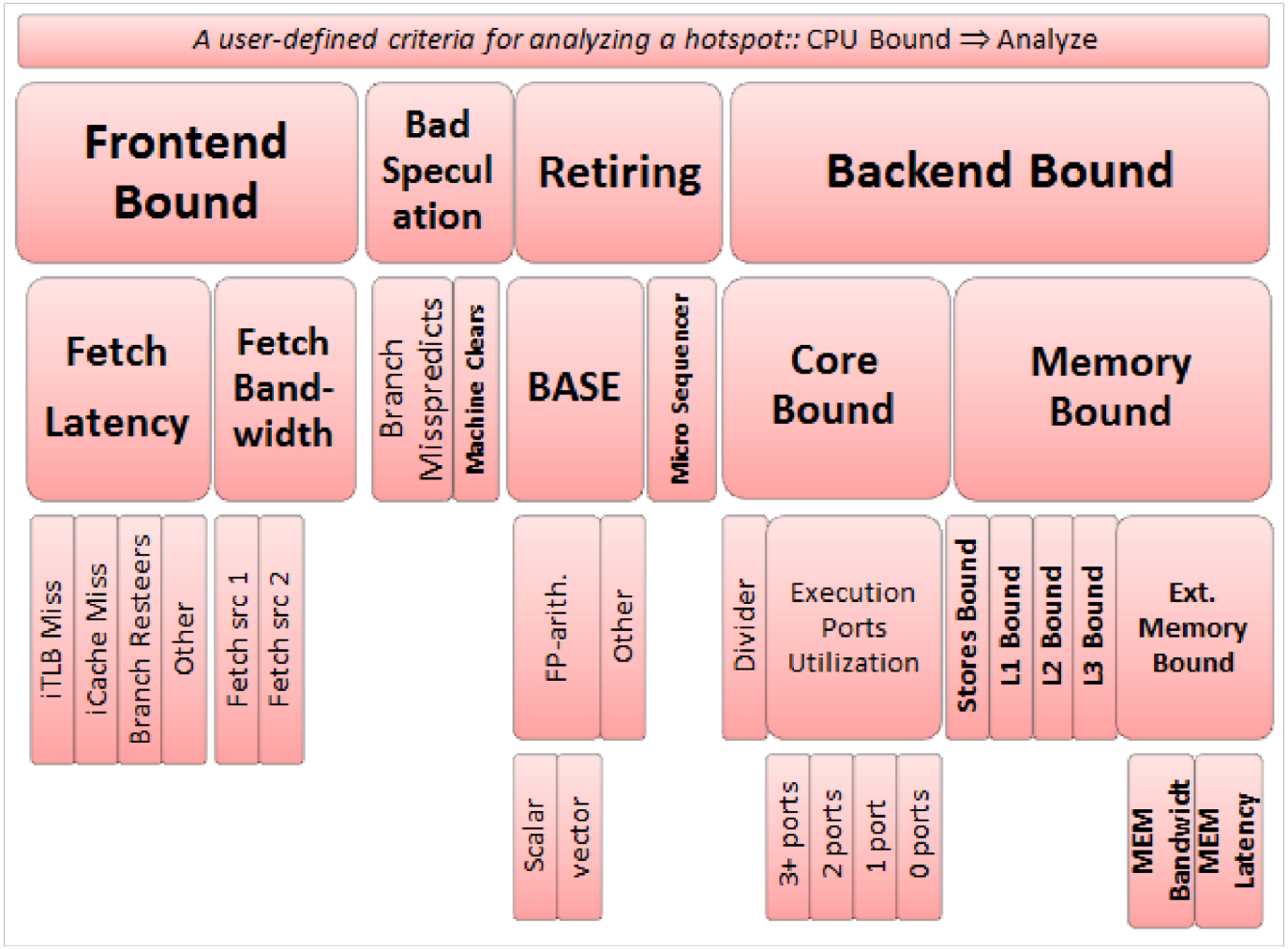
Measuring Performance (Linux)

- ***time** application*
  - Measures execution time of the application
  - Distinguishes between user and kernel
- ***perf record***
  - Low overhead sampling
  - Lists time spent per function
- ***perf topdown***
  - Uses hardware performance counters (PMU)
  - Enables microarchitectural studies

Perf record/report


```
94.72%  perf-test  libm-2.19.so       [.] __sin_sse2
        |
        --- __sin_sse2
            |
            |--44.20%-- foo
            |           |
            |            --100.00%-- main
            |                        __libc_start_main
            |                        _start
            |                        0x0
            |
            |--27.95%-- baz
            |           |
            |           |--51.78%-- bar
            |           |           foo
            |           |           main
            |           |           __libc_start_main
            |           |           _start
            |           |           0x0
            |           |
            |            --48.22%-- foo
            |                       main
            |                       __libc_start_main
            |                       _start
            |                       0x0
            |
```

Perf Topdown



Yasin, Ahmad. "A top-down method for performance analysis and counters architecture."

Microarchitectural Simulation

- Simulators: Gem5, Zsim, Mars, HDL
- Emulation + Timing Model
- Models processors hardware
- Can probe everything -> detailed analysis
- Enables modeling of new techniques

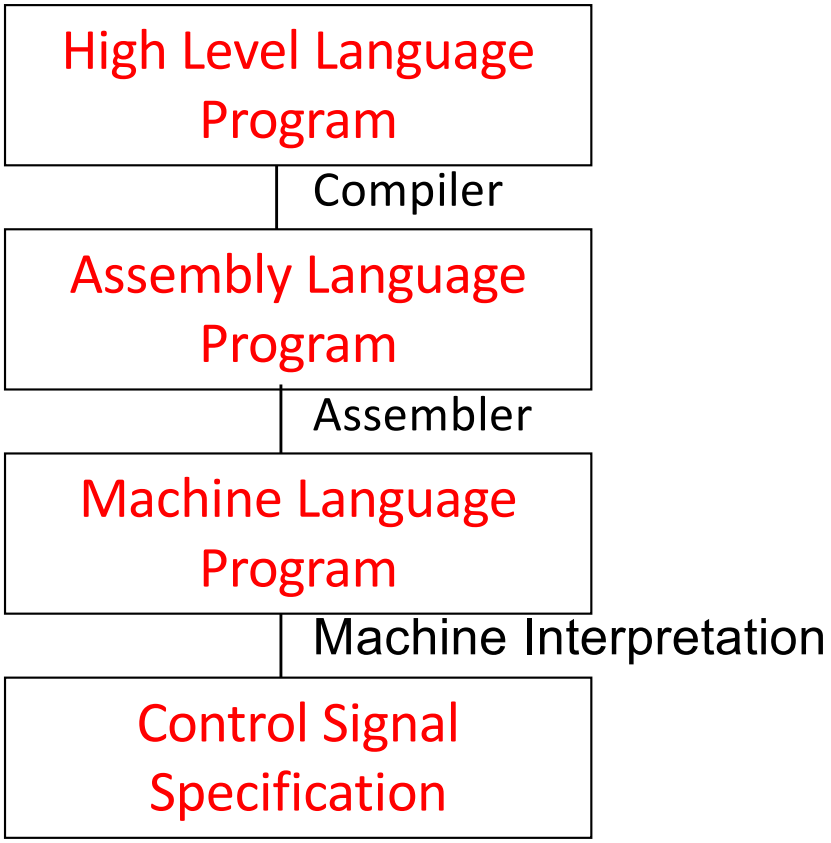
Key Tools for System Architects

1. **Pipelining**: overlap steps in execution; watch out for dependencies
2. **Parallelism**: execute independent tasks in parallel
3. **Out-of-order execution**: execute task in order of true dependencies
4. **Prediction**: better to ask for forgiveness than permission…
5. **Caching**: keep close a copy of frequently used information
6. **Indirection**: go through a translation step to allow intervention
7. **Amortization**: coarse-grain actions to amortize start/end overheads
8. **Redundancy**: extra information or resources to recover from errors
9. **Specialization**: trim overheads of general-purpose systems
10. **Focus on the common case**: optimize only the critical aspects of the system

Outlook: ISA

- Instruction Set Architecture will be the focus of the next lectures
- Example: RISC-V ISA
- Readings: Chapter 2, Instructions Language of the Computer
  - Sections 2.1 – 2.6
  - RISC-V specific

Big Picture: Running a Program




```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
ld $15, 0($2)
ld $16, 4($2)
sd $16, 0($2)
sd $15, 4($2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

High/Low on control lines

Instruction-Set Architecture
Where does it fit?