Putting it all Together: Modern Computer Architecture

Daniel Sanchez
Computer Science & Artificial Intelligence Lab
M.I.T.
Administrivia

• Quiz 3 tonight on room 50-340 (Walker Gym)
• Quiz 3 grades + final grade cutoffs tomorrow night
• Design project:
  - If you want to work a project of your choice for the optional part, contact us (preferably by Fri or Sat).
  - If you would like us to grade your project early to see if you got the necessary number of points for your desired grade, please send an e-mail to the course staff saying so.
• We will hold optional lectures Tue & Thu next week
• You must have completed all labs and part 1 of the design project by Thursday 3/17 to pass the course
• Please evaluate 6.S084. Your feedback matters!
A Trip Down Memory Lane...

• The 70s: Birth of the Microprocessor
  – Single-chip processors, multi-cycle execution
  – Personal computers

• The 80s: Quantitative Computer Architecture
  – Pipelining, caches, compiler considerations, RISC vs CISC
  – Workstations

• The 90s: Instruction-Level Parallelism
  – Superscalar out-of-order processors, cache hierarchies
  – Low-cost desktops, supercomputers

• The 2000s: Multicore Era
  – Multicore architectures, GPUs, power-constrained systems
  – Laptops, smartphones, datacenters
Exploiting Instruction-Level Parallelism
Complex Pipelines
(1985-2005)
Microprocessor Performance

\[
\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \cdot \frac{\text{Cycles}}{\text{Instruction}} \cdot \frac{\text{Time}}{\text{Cycle}} \cdot \text{CPI} \cdot \text{t}_{\text{CK}}
\]

- Pipelining lowers \( t_{\text{CK}} \). What about CPI?

- \( \text{CPI} = \text{CPI}_{\text{ideal}} + \text{CPI}_{\text{hazard}} \)
  - \( \text{CPI}_{\text{ideal}} \): cycles per instruction if no stall

- \( \text{CPI}_{\text{hazard}} \) contributors
  - Data hazards: long operations, cache misses
  - Control hazards: branches, exceptions
5-Stage Pipelined Processors

- **Advantages**
  - $CPI_{\text{ideal}} = 1$ (pipelining)
  - Simple and elegant
    - Still used in ARM & MIPS processors

- **Shortcomings**
  - Upper performance bound is $CPI=1$
  - Unnecessary stalls due to rigid pipeline
    - If one instruction stalls, anything behind it stalls
Improving 5-Stage Pipeline Performance

- **Increase clock frequency**: deeper pipelines
  - Overlap more instructions

- **Reduce CPI_{ideal}**: wider pipelines
  - Each pipeline stage processes multiple instructions

- **Reduce impact of data hazards**: out-of-order execution
  - Execute each instruction as soon as its source operands are available

- **Reduce impact of control hazards**: branch prediction
  - Predict both direction and target of branches and jumps
A Modern Out-of-Order Superscalar Processor

- I-Cache
- Fetch Unit
- Instruction Buffer
- Decode/Rename
- Dispatch
- Reservation Stations
- Int
- Int
- FP
- FP
- L/S
- L/S
- Reorder Buffer
- Retire
- Write Buffer
- D-Cache

Execute each instruction as soon as its source operands are available

Write back results in program order

Why is this needed?
Resolving Hazards

- **Strategy 1:** Stall. Wait for the result to be available by freezing earlier pipeline stages.

- **Strategy 2:** Bypass. Route data to the earlier pipeline stage as soon as it is calculated.

- **Strategy 3:** Speculate
  - Guess a value and continue executing anyway
  - When actual value is available, two cases
    - Guessed correctly → do nothing
    - Guessed incorrectly → kill & restart with correct value

- **Strategy 4:** Find something else to do
Out-of-Order Execution

• Consider the expression \( D = 3(a - b) + 7ac \)

Sequential code

```
ld a
ld b
sub a-b
mul 3(a-b)
ld c
mul ac
mul 7ac
add 3(a-b)+7ac
std d
```

Dataflow graph

Out-of-order execution runs instructions as soon as their inputs become available
Out-of-Order Execution Example

- If `ld b` takes a few cycles (e.g., cache miss), can execute instructions that do not depend on `b`

**Sequential code**

- `ld a`
- `ld b`
- `sub a-b`
- `mul 3(a-b)`
- `ld c`
- `mul ac`
- `mul 7ac`
- `add 3(a-b)+7ac`
- `st d`

**Dataflow graph**

- `ld b` → `ld a` → `ld c`
- `sub a-b` → `-`
- `mul ac` → `*`
- `mul 7ac` → `*`
- `add 3(a-b)+7ac` → `+`
- `st d`
Control Flow Penalty

- Modern processors have >10 pipeline stages between next PC calculation and branch resolution!

- How much work is lost every time pipeline does not follow correct instruction flow?

  Loop length x Pipeline width

- How much performance impact if 4-wide pipeline, 15 cycles from fetch to branch resolution, one branch every 5 instrs, and 50% of branches are mispredicted?

  60 instructions killed once every 10 instructions \( \Rightarrow \sim 7x \) slower than perfect prediction
Modern Processors Have Multiple Specialized Predictors

Need next PC immediately

Inst. type & branch target known

Branch direction & jump target known

Best predictors reflect program behavior

May 10, 2018
Putting It All Together: Intel Core i7 (Nehalem)

- Each core has 16 pipeline stages, ~3GHz
- 4-wide superscalar
- Out of order execution
- 2-level branch predictors
- Caches:
  - L1: 32KB I + 32KB D
  - L2: 256KB
  - L3: 8MB, shared

- Huge overheads vs simple, energy-optimized cores!
Exploiting Explicit Parallelism
Multicore, Multithreaded, and Vector Processors
(2005+)
Reminder: Thread-Level Parallelism

• Divide computation among multiple threads of execution
  – Each thread executes a different instruction stream

• Communication models:
  – Shared memory:
    • Single address space
    • Implicit communication by memory loads & stores
  – Message passing:
    • Separate address spaces
    • Explicit communication by sending and receiving messages
Multicore Chips

- Multicores integrate multiple processing cores on a single chip
  - Each core runs a different thread
- Multicores have **multiple private caches** for performance
- Need to provide the illusion of a single shared memory
- Problem:

  ![Diagram of caches and main memory]

  1. LD 0xA → 2
  2. ST 3 → 0xA
  3. LD 0xA → 2 (!!!)

- Solution: A **cache coherence protocol** controls cache contents to avoid stale lines
  - e.g., invalidate core 0’s copy of A before letting core 2 write to it
  - More on this next lecture!
Multithreaded Cores

- Data and control hazards cause frequent stalls
- Multithreaded cores support multiple threads and interleave them over a single pipeline to cope with stalls

Example: Interleave 4 threads, T0-T3, on non-bypassed 5-stage pipeline

T0: lw x2, 0(x1)
T1: add x7, x1, x4
T2: xor x5, x7, x3
T3: sw x1, 0(x3)
T0: lw x3, 0(x2)

- Note difference with operating system scheduling
  - In single-threaded cores, the kernel can schedule multiple thread over time, but scheduling actions are taken on the order of milliseconds
  - In multithreaded cores, hardware interleaves threads at instruction granularity
  - Each multithreaded core appears to software as multiple, albeit slower, cores
Simple Multithreaded Pipeline

- Each thread needs its own architectural state
  - PC and general-purpose registers (+ CSRs)

- Have to carry thread select signal down pipeline to ensure correct state bits read/written at each pipe stage
Multithreading Granularity

A) Conventional Processor

B) Coarse-grained Multithreaded (CMT)

C) Fine-grained Multithreaded (FMT)

D) Simultaneous Multithreaded (SMT)

Execution Units

Time

Interrupt, exception, or OS call

return from exception

Cache miss

Cache miss

Cache miss
Data-Level Parallelism

- Same operation applied to multiple data elements
  
  ```c
  for (int i = 0; i < 16; i++) x[i] = a*b[i] + c[i];
  ```

- Exploit with vector processors or vector ISA extensions
  
  - Each datapath has its own local storage (register file)
  - All datapaths execute the same instruction
  - Memory access with vector loads and stores + wide memory port

May 10, 2018
Vector Code Example

for (i = 0; i < 16; i++)  x[i] = a[i] + b[i];

**RISC-V assembly**

li t0, 16
loop:  lw t4, 0(t1)
       lw t5, 0(t2)
       addi t1, t1, 4
       addi t2, t2, 4
       addi t6, t4, t5
       sw t6, 0(t3)
       addi t3, t3, 4
       addi t1, t1, -1
       bnez t0, loop

**Equivalent vector assembly**

ld.v v1, 0(t1)
ld.v v2, 0(t2)
add.v v3, v1, v2
st.v v3, 0(t3)

# of cycles = 1 + 9*16 = 145

# of cycles = ~ 5
Vector Processing Implementations

• Advantages of vector ISAs:
  – **Compact**: 1 instruction defines N operations
  – **Parallel**: N operations are (data) parallel and independent
  – **Expressive**: Memory operations describe regular patterns

• Modern CPUs: Vector (SIMD) extensions & wider registers
  – SSE (1999): 128-bit operands (4x32-bit or 2x64-bit)
  – AVX (2011): 256-bit operands (8x32-bit or 4x64-bit)
  – AVX-512 (2017): 512-bit operands
  – Explicit parallelism, extracted at compile time (vectorization)

• GPUs: Designed for data parallelism from the ground up
  – Each core processes ~32 32-bit floating-point elements per cycle
  – Each core is highly multithreaded (~64 threads/core)
    • Hide long delays effectively → design for throughput, not latency
  – Implicit parallelism, scalar binary with multiple instances executed in lockstep (and regrouped dynamically)
Putting It All Together: Knights Landing

- 72 energy-efficient cores
  - 2-wide superscalar
  - Out-of-order execution
  - 1.5 GHz
- 4 threads per core
- Vector extensions (512-bit registers)
- 4-level memory hierarchy
- Peak performance ~6 TFLOPS (32-bit floats)
  - Performance for a sequential program?

Intel, 2016, 14 nm, 7.1B transistors, 683mm²
Putting It All Together: NVIDIA Pascal GP100

- 60 cores at 1.5 GHz:
  - 64 threads per core
  - 32x32-bit elements/thread
  - 2-wide superscalar, in-order

- Memory hierarchy:
  - 256 KB register file/core (!)
  - 64 KB L1/core
  - 4 MB shared L2
  - 8GB memory, 720GB/s

- Lots of specialized logic for graphics (texture units, raster ops, ...)

- Peak performance ~12 TFLOPS (32-bit floats)
Recap: Key Computer Architecture Principles

• Amdahl’s Law → Make the common case fast
• Exploit parallelism
  – e.g., pipelining, multicore
• Exploit locality: Recent past is a very good indication of near future
  – Locality == Patterns == Predictability
  – e.g., caches
• Speculate: When dependences are predictable, guess the outcome
  – e.g., branch prediction
• Amortize costs: Divide overheads among many units
  – e.g., large pages, vector instructions

These principles are crucial to design any type of digital system
Unfortunately, Technology Scaling Likely to End Soon…

- Voltage scaling ended around 2005
- Moore’s Law likely to end by 2020-2025
  - Many promising technologies being developed, but none ready to replace CMOS soon
Parallelism and Specialization Tradeoffs

- Good news: Plenty of performance left on the table
  - Simple cores have ~10x lower energy/instruction than complex cores
    → can scale to about ~1000 simple cores within power constraints
  - Specialized compute units have ~10-1000x perf/energy savings over general-purpose cores

- Bad news: Harder to build and use, less general

- Trillion-dollar question: What is the right balance between efficiency, generality, and ease of use?
The Trend Towards Specialization

- Apple A11 (iPhone 8/10): >50% area devoted to specialized logic
- Google TPU (2016) is a specialized system for deep learning
- Microsoft and Amazon deploying FPGA accelerators in their datacenters
- None of these companies built chips ~10 years ago!
- It is increasingly likely that you will build or closely interact with specialized chips in your career

Apple, 2017, TSMC 10nm, 4B transistors, 89mm$^2$

Google, 2017

Microsoft, 2014
[Catapult ISCA 2014]
Thank you!

Next lecture: Cache coherence