Last Time …

- Addressing modes
- Other ISA-level tradeoffs
- Programmer vs. microarchitect
  - Virtual memory
  - Unaligned access
  - Transactional memory
- Control flow vs. data flow
- The Von Neumann Model
- The Performance Equation
Review: Other ISA-level Tradeoffs

- Load/store vs. Memory/Memory
- Condition codes vs. condition registers vs. compare&test
- Hardware interlocks vs. software-guaranteed interlocking
- VLIW vs. single instruction
- 0, 1, 2, 3 address machines
- Precise vs. imprecise exceptions
- Virtual memory vs. not
- Aligned vs. unaligned access
- Supported data types
- Software vs. hardware managed page fault handling
- Granularity of atomicity
- Cache coherence (hardware vs. software)
- …
Review: The Von-Neumann Model

INPUT

MEMORY
- Mem Addr Reg
- Mem Data Reg

PROCESSING UNIT
- ALU
- TEMP

CONTROL UNIT
- IP
- Inst Register

OUTPUT
Review: The Von-Neumann Model

- Stored program computer (instructions in memory)
- One instruction at a time
- Sequential execution
- Unified memory
  - The interpretation of a stored value depends on the control signals

- All major ISAs today use this model
- Underneath (at uarch level), the execution model is very different
  - Multiple instructions at a time
  - Out-of-order execution
  - Separate instruction and data caches
Review: Fundamentals of Uarch Performance Tradeoffs

- Zero-cycle latency (no cache miss)
- No branch mispredicts
- No fetch breaks

Data Path (Functional Units)

- Perfect data flow (reg/memory dependencies)
- Zero-cycle interconnect (operand communication)
- Enough functional units
- Zero latency compute?

Data Supply

- Zero-cycle latency
- Infinite capacity
- Zero cost
- Zero latency compute?

We will examine all these throughout the course (especially data supply)
Review: How to Evaluate Performance Tradeoffs

\[
\text{Execution time} = \frac{\text{time}}{\text{program}} = \frac{\text{# instructions}}{\text{program}} \times \frac{\text{# cycles}}{\text{instruction}} \times \frac{\text{time}}{\text{cycle}}
\]

- Algorithm
- Program
- ISA
- Compiler
- Microarchitecture
- Logic design
- Circuit implementation
- Technology
Improving Performance (Reducing Exec Time)

- Reducing instructions/program
  - More efficient algorithms and programs
  - Better ISA?

- Reducing cycles/instruction (CPI)
  - Better microarchitecture design
    - Execute multiple instructions at the same time
    - Reduce latency of instructions (1-cycle vs. 100-cycle memory access)

- Reducing time/cycle (clock period)
  - Technology scaling
  - Pipelining
Other Performance Metrics: IPS

- Machine A: 10 billion instructions per second
- Machine B: 1 billion instructions per second
- Which machine has higher performance?

Instructions Per Second (IPS, MIPS, BIPS)

\[
\text{IPS} = \frac{\text{# of instructions}}{\text{cycle}} \times \frac{\text{cycle}}{\text{time}}
\]

- How does this relate to execution time?
- When is this a good metric for comparing two machines?
  - Same instruction set, same binary (i.e., same compiler), same operating system
  - Meaningless if “Instruction count” does not correspond to “work”
    - E.g., some optimizations add instructions, but do not change “work”
Other Performance Metrics: FLOPS

- Machine A: 10 billion FP instructions per second
- Machine B: 1 billion FP instructions per second
- Which machine has higher performance?

- **Floating Point Operations per Second** (FLOPS, MFLOPS, GFLOPS)
  - Popular in scientific computing
  - FP operations used to be very slow (think Amdahl’s law)
- Why not a good metric?
  - Ignores all other instructions
    - what if your program has 0 FP instructions?
  - Not all FP ops are the same
Other Performance Metrics: Perf/Frequency

- **SPEC/MHz**

- Remember: Execution time \(\frac{\text{time}}{\text{program}}\) = \(\frac{1}{\text{Performance}}\)

- **Performance/Frequency**

\[
\frac{\text{time}}{\text{cycle}} = \frac{\# \text{ instructions}}{\text{program}} \times \frac{\# \text{ cycles}}{\text{instruction}} \times \frac{\text{time}}{\text{cycle}}
\]

\[
= \frac{1}{\{\frac{\# \text{ cycles}}{\text{program}}\}}
\]

- What is wrong with comparing only “cycle count”?
  - Unfairly penalizes machines with high frequency
  - For machines of equal frequency, fairly reflects performance assuming equal amount of “work” is done
    - Fair if used to compare two different same-ISA processors on the same binaries
An Example

### Amdahl’s Law: Bottleneck Analysis

- **Speedup** = \( \frac{\text{time}_{\text{without enhancement}}}{\text{time}_{\text{with enhancement}}} \)
- Suppose an enhancement speeds up a fraction \( f \) of a task by a factor of \( S \)

\[
\text{time}_{\text{enhanced}} = \text{time}_{\text{original}} \cdot (1-f) + \text{time}_{\text{original}} \cdot \frac{f}{S}
\]

\[
\text{Speedup}_{\text{overall}} = \frac{1}{(1-f) + \frac{f}{S}}
\]

*Focus on bottlenecks with large \( f \) (and large \( S \))*
Microarchitecture Design Principles

- **Bread and butter design**
  - Spend time and resources on where it matters (i.e. improving what the machine is designed to do)
  - Common case vs. uncommon case

- **Balanced design**
  - Balance instruction/data flow through uarch components
  - Design to eliminate bottlenecks

- **Critical path design**
  - Find the maximum speed path and decrease it
    - Break a path into multiple cycles?
Cycle Time (Frequency) vs. CPI (IPC)

- Usually at odds with each other

- Why?
  - Memory access latency: Increased frequency increases the number of cycles it takes to access main memory
  - Pipelining: A deeper pipeline increases frequency, but also increases the “stall” cycles:
    - Data dependency stalls
    - Control dependency stalls
    - Resource contention stalls
Intro to Pipelining (I)

- Single-cycle machines
  - Each instruction executed in one cycle
  - The slowest instruction determines cycle time

- Multi-cycle machines
  - Instruction execution divided into multiple cycles
    - Fetch, decode, eval addr, fetch operands, execute, store result
    - Advantage: the slowest “stage” determines cycle time
  - Microcoded machines
    - Microinstruction: Control signals for the current cycle
    - Microcode: Set of all microinstructions needed to implement instructions → Translates each instruction into a set of microinstructions
Microcoded Execution of an ADD

- **ADD DR ← SR1, SR2**
- **Fetch:**
  - MAR ← IP
  - MDR ← MEM[MAR]
  - IR ← MDR
- **Decode:**
  - Control Signals ← DecodeLogic(IR)
- **Execute:**
  - TEMP ← SR1 + SR2
- **Store result (Writeback):**
  - DR ← TEMP
  - IP ← IP + 4

What if this is SLOW?
Intro to Pipelining (II)

- In the microcoded machine, some resources are idle in different stages of instruction processing
  - Fetch logic is idle when ADD is being decoded or executed

- Pipelined machines
  - Use idle resources to process other instructions
  - Each stage processes a different instruction
  - When decoding the ADD, fetch the next instruction
  - Think “assembly line”

- Pipelined vs. multi-cycle machines
  - Advantage: Improves instruction throughput (reduces CPI)
  - Disadvantage: Requires more logic, higher power consumption
A Simple Pipeline
Execution of Four Independent ADDs

- Multi-cycle: 4 cycles per instruction

- Pipelined: 4 cycles per 4 instructions (steady state)
Issues in Pipelining: Increased CPI

- **Data dependency stall**: what if the next ADD is dependent

  \[
  \begin{array}{l}
  \text{ADD R3} \leftarrow \text{R1, R2} \\
  \text{ADD R4} \leftarrow \text{R3, R7}
  \end{array}
  \]

  \[
  \begin{array}{cccccc}
  F & D & E & W \\
  F & D & D & E & W
  \end{array}
  \]

- **Solution**: data forwarding. Can this always work?
  - How about memory operations? Cache misses?
  - If data is not available by the time it is needed: STALL

- **What if the pipeline was like this?**

  \[
  \begin{array}{l}
  \text{LD R3} \leftarrow \text{R2(0)} \\
  \text{ADD R4} \leftarrow \text{R3, R7}
  \end{array}
  \]

  \[
  \begin{array}{cccccc}
  F & D & E & M & W \\
  F & D & E & E & M & W
  \end{array}
  \]

  - R3 cannot be forwarded until read from memory
  - Is there a way to make ADD not stall?
Implementing Stalling

- Hardware based interlocking
  - Common way: scoreboard
  - i.e. valid bit associated with each register in the register file
  - Valid bits also associated with each forwarding/bypass path
Data Dependency Types

- Types of data-related dependencies
  - Flow dependency (true data dependency – read after write)
  - Output dependency (write after write)
  - Anti dependency (write after read)

- Which ones cause stalls in a pipelined machine?
  - Answer: It depends on the pipeline design
  - In our simple strictly-4-stage pipeline, only flow dependencies cause stalls
  - *What if instructions completed out of program order?*
Issues in Pipelining: Increased CPI

- **Control dependency stall**: what to fetch next

  ![Pipeline States](image)

  - Solution: predict which instruction comes next
    - What if prediction is wrong?

- Another solution: hardware-based *fine-grained multithreading*
  - Can **tolerate** both data and control dependencies