Performance: a measure of how cool your machine is

- Performance = 1 / Execution time
- Speedup = $T_{old} / T_{new}$
- Amdahl’s Law: Make the common case fast!
- CPI (clock cycles per instruction)
  \[
  CPI = \frac{t_{ex} \times f_{ck}}{I}
  \]
  where $I$ is the instruction count in the sample program,
  $f_{ck}$ is the clock frequency of the machine,
  $t_{ex}$ is the total execution time of the sample program
  \[
  \overline{CPI} = \frac{\sum \text{CPI}_i \times \text{Weight}_i}{\sum \text{Weight}_i}
  \]

- MIPS (million instructions per second)
  \[
  \text{MIPS} = \frac{1}{t_{ex} \times 10^6}
  \]
  \[
  \text{MIPS} = \frac{f_{ck}}{CPI \times 10^6}
  \]
  - Native MIPS uses mean (average) CPI; Relative MIPS is relative to another machine; Peak MIPS minimizes the CPI (uses best-case CPI)
  - MOPS (million operations per second) is like MIPS, but one operation does not necessarily map to one instruction (an instruction can take multiple operations)
  - FLOPS (floating point operations per second) is like MOPS, but takes into account only the floating point instructions (use CPI of floating-point class instructions only)

Benchmark: a standard way to quantify the performance of a machine

- Run real applications or “synthetic” programs to get nice numbers.
- Arithmetic Mean (average): Sum up all $n$ values, divide by $n$.
- Weighted Arithmetic Mean: Multiply each value $n$ by weight $w$. Add.
- Geometric Mean: Multiply all $n$ values, take the $n$th root.

Architectures are your friends
• Stack (all operands are kept on a stack); Accumulator (one operand is in accumulator); General Purpose Register (load/store)
• Endianness: Little-endian (word address is at LSByte); Big-endian (word address is at MSByte)

MIPS Instruction Set is great; its registers are many! But if you want accumulators, with MIPS you won’t find any.

• MIPS is a RISC machine (CPI ≈ 1) with a load/store architecture
• 32 registers, 32-bit (4-byte) words
• Instruction Types
  R-type (e.g., add, xor, jr)
  [ opcode(6) ][ rs(5) ][ rt(5) ][ rd(5) ][ shamt(5) ][ func(6) ]
  I-type (e.g., addi, lui, lw, sw, beq)
  [ opcode(6) ][ rs(5) ][ rt(5) ][ immediate(16) ]
  J-type (e.g., j, jal)
  [ opcode(6) ][ target(26) ]

Addressing Modes: know all six MIPS addressing modes, and the others too

Register: Specify register number
  add $t0, $t1, $t2
Immediate: Operand embedded in instruction (16 bits only!)
  addi $t0, $t1, 77
Base: (Displacement) Add register and immediate
  lw $s0, 12($s1)
PC-Relative: Add immediate (and the “understood zeros”) to PC
  beq $0, $t3, -44
Indirect: Register has address
  jr, jair
Pseudodirect: Add 26-bit immediate to top 4 bits of PC
  j, jal
Direct (NOT IN MIPS): Address is the immediate
Indexed (NOT IN MIPS): Address is sum of two registers
Update (NOT IN MIPS): Base addressing, plus automatically increment register
Memory Indirect (NOT IN MIPS): Register points to memory location, which points to operand
Inherent (NOT IN MIPS): No operands provided; accumulator assumed

Number Conversion goes without saying
• Given a number in base X, can you convert it to base Y? With a radix point?
• Binary (see 12e lab manual)
  – Unsigned: cannot represent negative numbers; Sign-Magnitude: MSb is sign bit; One’s Complement: if negative, flip all bits; Two’s Complement: if negative, add one to 1’s comp; Bias (or Excess): to get in, add bias to number; to get out, subtract bias from number
• Octal: group binary into threes
• Hex: group binary into fours

Logic Design
• Truth tables, logical operations
• Simple logic: and, or, not, xor, mux
• Memory elements: flip-flops, registers
• Propagation delay (how long it takes for the device to give output)
• Simple CMOS circuits (inverter, nand, nor)
• Stuck-at faults
• Timing diagrams

Making Hardware Add
• Half-adder (works only with inputs a,b — not carry-in)
• Full adder (1 bit)
• Ripple-carry (slow): Adds serially; you don’t know the result until all bits are finished
• Carry lookahead (faster): Adds in parallel; uses generate and propagate equations to compute carry-ins.
  \[
g_i = a_i \times b_i \\
p_i = a_i + b_i \\
\text{Cin}_0 = c_0 \\
\text{Cin}_n = g_{n-1} + p_{n-1}\text{Cin}_{n-1}
\]
• Carry select: Adds both with Cin=0 and =1, then figures out what you really wanted
• Two-level logic (fastest, but impractical)
• Manchester carry chain

The ALU
1. Logical operations
2. Full adder (one that works with carries)
3. Add comparison
4. Ripple-carry ALU

Multiplication Schemes
• A hardware multiplier: Let \(A=\)multiplicand; \(B=\)multiplier; \(P=\)product
  \(P = (A \times \text{LSb } B); \) shift \(B\) right, shift \(A\) left
• Booth’s Algorithm: Remember to sign-extend partial products to \(2n\) bits!!

<table>
<thead>
<tr>
<th>Bit pattern</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>—</td>
</tr>
<tr>
<td>01</td>
<td>+A</td>
</tr>
<tr>
<td>10</td>
<td>-A</td>
</tr>
<tr>
<td>11</td>
<td>—</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit pattern</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>—</td>
</tr>
<tr>
<td>001</td>
<td>+A</td>
</tr>
<tr>
<td>001</td>
<td>+A</td>
</tr>
<tr>
<td>011</td>
<td>+2A</td>
</tr>
<tr>
<td>100</td>
<td>-2A</td>
</tr>
<tr>
<td>101</td>
<td>-A</td>
</tr>
<tr>
<td>110</td>
<td>-A</td>
</tr>
<tr>
<td>111</td>
<td>—</td>
</tr>
</tbody>
</table>
Division Schemes

- Division in Hardware: Restoring, Non-restoring
- Remember to restore to positive remainder, regardless of algorithm
- Both restoring and non-restoring division work only for positive numbers!

Floating Point

- IEEE-754 Floating Point Standard:
  Single-precision (32 bits): 1-bit sign; 8-bit bias-127 exponent; 23-bit unsigned mantissa
  Double-precision (64 bits): 1-bit sign; 11-bit bias-1023 exponent; 52-bit unsigned mantissa
- FP numbers are in the form (S) 1.FFF.. x 2^E
- The “1.” is the hidden bit – it is always 1 for a normalized number. You MUST have your number normalized to convert to FP.

<table>
<thead>
<tr>
<th>Form</th>
<th>Sign</th>
<th>Exponent</th>
<th>Mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td>±0</td>
<td>0</td>
<td>All 0</td>
<td>All 0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>All 0</td>
<td>All 0</td>
</tr>
<tr>
<td>± Inf</td>
<td>0</td>
<td>All 1</td>
<td>All 0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>All 1</td>
<td>All 0</td>
</tr>
<tr>
<td>NaN</td>
<td>0</td>
<td>All 1</td>
<td>nonzero</td>
</tr>
<tr>
<td>Denorm</td>
<td>0</td>
<td>All 0</td>
<td>nonzero</td>
</tr>
</tbody>
</table>

- Arithmetic with special forms: Division by zero gives ±Inf; Division by Inf gives ±0. NaN propagates through any computation.
Single-cycle CPU: Each instruction takes one cycle

- Clock period is propagation delay of the longest instruction (usually load word)
- Control signals for various instructions

Multi-cycle CPU: Each instruction has a different CPI

- Our stages:
  IF: Instruction fetch from instruction memory; PC+4
  ID: Instruction decode; break into bitfields; calculate branch and jump targets; read registers
  EX: Use ALU for computations, including branch condition, memory addresses, etc.; branches and jumps are resolved here
  MEM: Optional cycle for data memory access (loads and stores)
  WB: Optional cycle for register writes (all instructions that need to update the register bank)

- In our system, R-type instructions take 4 cycles (no MEM);
- J-type take 3 cycles; Branch takes 3; SW takes 4 (no WB); LW takes 5.
- Clock period is longest stage (usually MEM)
Pipelined CPU: Each instruction takes five cycles, but overlap with other instructions

- See newsgroup post
- Clock period is longest stage (usually MEM)
Cache is closer than memory, but farther than registers

- Mappings: Direct-mapped (a.k.a. one-way set associative), two-way, four-way, etc. set-associative, and fully associative
- Parts of the cache
  - The index: the address of the cache line (not a part of cache)
  - Valid bit: present in all caches. Answers the question, “Is the data here valid data?” The valid bit is zero until a cache hit occurs.
  - Dirty bit: only in write-back cache. Answers the question, “Is the data in memory older than the data here?”
  - LRU bit: only in caches with a replacement policy (associative caches that use a replacement policy other than “random”). Answers the question, “Has this data been accessed most recently?”
  - Tag: used for address comparison in cache lookup
  - Data: the other part of cache that isn’t housekeeping
- Parts of the lookup (starting at least significant bits)
  - The lookup itself is a memory address consisting of \( N \) bits. The size of the lookup is the size of the main memory (not cache) address. For example, for a 1-GB (= \( 2^{30} \)) main memory, the address (and, hence, lookup) is \( N = 30 \) bits.
  - Byte offset (in a byte-addressed system): \( y \) bits, where \( 2^y \) bytes in a block (usually 4 bytes to a block, hence 2 bits for byte offset)
  - Block offset (in a multi-word cache): \( b \) bits, where \( 2^b \) blocks per line. In a single-word cache \( b = 0 \).
  - Index: \( n \) bits, where
    * (for direct-mapped) \( 2^n \) lines in cache
    * (for set-associative) \( 2^n \) sets in cache
    * (for fully-associative) \( n = 0 \)
  - Tag: the rest of the bits: tag bits = \( N - y - b - n \)
- What happens on a memory access (read) with a cache
  1. Send memory address to cache. Break it up into offset, index.
  2. Activate cache line at index specified.
  5. Cache hit! Fetch cache data; complete instruction.
  6. Cache miss. Send memory address to memory. Fetch data. Wait forever (the CPU is stalled).
  7. Update cache with new data / tag / valid / LRU bits, as appropriate.
  8. Restart (the same) instruction at step 1.

Virtual Memory (1) is a neat system to give each process its own address space in memory and on disk; (2) provides us with the ability to exceed memory space and spill out onto disk.

- Parts of the VM lookup (starting at least significant bits)
  - The lookup itself is a virtual address which is translated to become a physical address. The virtual address is \( V \) bits, where \( 2^V \) is the size of virtual memory. The physical (main) memory is smaller; the physical address is \( N \) bits, where \( 2^N \) is the size of main memory.
- Page offset: \( p \) bits, with \( 2^p \) bytes in a page
- Virtual page number: \( v = V - p \) bits. Undergoes translation to become physical page number
- Physical page number: \( n = N - p \) bits, where \( 2^n \) is the number of physical pages allowed in memory
- Virtual to physical address translation: done by lookup on the page table
- Parts of the page table
  - The page table is a cache for address translation. It lives in main memory; all pages in the virtual memory space have an entry in the page table. A page fault is when a page is not in main memory (but is off on a disk somewhere), and is handled similarly to the cache miss (though much more slowly).
  - Valid bit: Answers the question, “Is the data here valid data?” The valid bit is zero until a page hit occurs.
  - Dirty bit: Answers the question, “Is the data on disk older than the data here?”
  - LRU or referenced bit(s), if least recently used replacement policy is used
  - Data: physical page number
- VM is usually fully-associative with an LRU replacement policy

**TLB (Translation-Lookaside Buffer)** is usually small, fast, and fully-associative; it is a cache for the page table. It lives in cache and has a much lower miss penalty than the page table.
- Parts of the TLB
  - Valid bit (read-only)
  - Dirty bit (copied to the page table upon replacement)
  - Referenced bit, if LRU policy is used (copied to the page table upon replacement)
  - Tag (virtual page number, if fully associative TLB)
  - Physical page address (read-only)

**Accessing data with VM, TLB, Memory, and Cache**

1. Send \( V \)-bit virtual address to TLB (in cache). Only the top \( v = V - p \) bits are used as the tag, if the TLB is fully associative.
2. Check TLB valid bit. If valid, go to step 3. Otherwise, skip to step 5.
3. Check TLB tag against virtual page number. If match, go to step 4. Otherwise, skip to step 5.
4. **TLB hit!** Get \( n \)-bit physical page number from TLB; combine it with the \( p \)-bit page offset to get a \( N \)-bit physical address. Update referenced bit. Send this physical address to cache; on a miss, try memory; on a miss, try disk, and so on.
5. **TLB miss!** Send virtual address to page table (in memory). Activate line in page table at virtual page number; check valid bit. If valid, go to step 6. Otherwise, skip to step 7.
6. **Page table hit!** Get \( n \)-bit physical page number from page table; combine it with the \( p \)-bit page offset to get a \( N \)-bit physical address. Update referenced bit. Send this physical address to cache; on a miss, try memory; on a miss, try disk, and so on.
7. **Page fault!** The page requested is not in memory. Wait forever while the operating system deals with it (by going to find it on disk or farther away). Then, restart instruction at step 1.
<table>
<thead>
<tr>
<th>Cache</th>
<th>TLB</th>
<th>VM</th>
<th>Under what circumstances?</th>
</tr>
</thead>
<tbody>
<tr>
<td>miss</td>
<td>hit</td>
<td>hit</td>
<td>Possible, although the page table is never really checked if TLB hits.</td>
</tr>
<tr>
<td>hit</td>
<td>miss</td>
<td>hit</td>
<td>TLB misses, but entry found in page table; after retry, data is found in cache.</td>
</tr>
<tr>
<td>miss</td>
<td>miss</td>
<td>hit</td>
<td>TLB misses, but entry found in page table; after retry, data misses in cache.</td>
</tr>
<tr>
<td>miss</td>
<td>miss</td>
<td>miss</td>
<td>TLB misses and is followed by a page fault; after retry, data must miss in cache.</td>
</tr>
</tbody>
</table>

All other combinations impossible
(Figure 7.27 in course textbook)

**Other Topics**

- Kestrel is the coolest massively parallel processor known to man.
- See your class notes.