Review of Last Lecture

- Intro to on-chip networks
- More on DRAM controllers and bank operation
- Begin ISA tradeoffs
  - Semantic gap
Today

- More ISA tradeoffs
Review: Levels of Transformation

- **ISA**
  - Agreed upon interface between software and hardware
    - SW/compiler assumes, HW promises
  - What the software writer needs to know to write system/user programs

- **Microarchitecture**
  - Specific implementation of an ISA
  - Not visible to the software

- **Microprocessor**
  - **ISA, uarch, circuits**
  - “Architecture” = ISA + microarchitecture
Review: ISA vs. Microarchitecture

- What is part of ISA vs. Uarch?
  - Gas pedal: interface for “acceleration”
  - Internals of the engine: implements “acceleration”
  - Add instruction vs. Adder implementation

- Implementation (uarch) can be various as long as it satisfies the specification (ISA)
  - Bit serial, ripple carry, carry lookahead adders
  - x86 ISA has many implementations: 286, 386, 486, Pentium, Pentium Pro, ...

- Uarch usually changes faster than ISA
  - Few ISAs (x86, SPARC, MIPS, Alpha) but many uarchs
  - Why?
Review: Microarchitecture

- Implementation of the ISA under specific design constraints and goals
- Anything done in hardware without exposure to software
  - Pipelining
  - In-order versus out-of-order instruction execution
  - Memory access scheduling policy
  - Speculative execution
  - Superscalar processing (multiple instruction issue?)
  - Clock gating
  - Caching? Levels, size, associativity, replacement policy
  - Prefetching?
  - Voltage/frequency scaling?
  - Error correction?
Review: Tradeoffs: Soul of Computer Architecture

- ISA-level tradeoffs
- Uarch-level tradeoffs
- System and Task-level tradeoffs
  - How to divide the labor between hardware and software
Review: ISA-level Tradeoffs: Semantic Gap

- **Where to place the ISA?** Semantic gap
  - Closer to high-level language (HLL) or closer to hardware control signals? → Complex vs. simple instructions
  - RISC vs. CISC vs. HLL machines
    - FFT, QUICKSORT, POLY, FP instructions?
    - VAX INDEX instruction (array access with bounds checking)
  - **Tradeoffs:**
    - Simple compiler, complex hardware vs. complex compiler, simple hardware
    - Caveat: Translation (indirection) can change the tradeoff!
    - Burden of backward compatibility
    - Performance?
      - Optimization opportunity: Example of VAX INDEX instruction: who (compiler vs. hardware) puts more effort into optimization?
        - Instruction size, code size
REP MOVSM DEST SRC

IF AddressSize = 16
THEN
  Use CX for CountReg;
ELSE IF AddressSize = 64 and REX.W used
  THEN Use RCX for CountReg; Fl;
ELSE
  Use ECX for CountReg;
Fl;
WHILE CountReg ≠ 0
  DO
    Service pending interrupts (if any);
    Execute associated string instruction;
    CountReg ← (CountReg - 1);
    IF CountReg = 0
      THEN exit WHILE loop; Fl;
    IF (Repeat prefix is REPZ or REPE) and (ZF = 0)
      or (Repeat prefix is REPNZ or REPNE) and (ZF = 1)
      THEN exit WHILE loop; Fl;
  OD;

How many instructions does this take in Alpha?
Small Semantic Gap Examples in VAX

- **FIND FIRST**
  - Find the first set bit in a bit field
  - Helps OS resource allocation operations
- **SAVE CONTEXT, LOAD CONTEXT**
  - Special context switching instructions
- **INSQUEUE, REMQUEUE**
  - Operations on doubly linked list
- **INDEX**
  - Array access with bounds checking
- **STRING Operations**
  - Compare strings, find substrings, ...
- **Cyclic Redundancy Check Instruction**
- **EDITPC**
  - Implements editing functions to display fixed format output

Small versus Large Semantic Gap

- **CISC vs. RISC**
  - Complex instruction set computer → complex instructions
    - Initially motivated by “not good enough” code generation
  - Reduced instruction set computer → simple instructions
    - John Cocke, mid 1970s, IBM 801 (see ASPLOS 1982 paper)
      - Goal: enable better compiler control and optimization

- **RISC motivated by**
  - Memory stalls (no work done in a complex instruction when there is a memory stall?)
    - When is this correct?
  - Simplifying the hardware → lower cost, higher frequency
  - Enabling the compiler to optimize the code better
    - Find fine-grained parallelism to reduce stalls
Small versus Large Semantic Gap

- John Cocke’s RISC (large semantic gap) concept:
  - Compiler generates control signals: open microcode

- Advantages of Small Semantic Gap (Complex instructions)
  + Denser encoding $\rightarrow$ smaller code size $\rightarrow$ saves off-chip bandwidth, better cache hit rate (better packing of instructions)
  + Simpler compiler

- Disadvantages
  - Larger chunks of work $\rightarrow$ compiler has less opportunity to optimize
  - More complex hardware $\rightarrow$ translation to control signals and optimization needs to be done by hardware

ISA-level Tradeoffs: Instruction Length

- **Fixed length**: Length of all instructions the same
  - + Easier to decode single instruction in hardware
  - + Easier to decode multiple instructions concurrently
  - -- Wasted bits in instructions *(Why is this bad?)*
  - -- Harder-to-extend ISA (how to add new instructions?)

- **Variable length**: Length of instructions different (determined by opcode and sub-opcode)
  - + Compact encoding *(Why is this good?)*
    - Intel 432: Huffman encoding (sort of). 6 to 321 bit instructions. *How?*
  - -- More logic to decode a single instruction
  - -- Harder to decode multiple instructions concurrently

- **Tradeoffs**
  - Code size (memory space, bandwidth, latency) vs. hardware complexity
  - ISA extensibility and expressiveness
  - Performance? Smaller code vs. imperfect decode
ISA-level Tradeoffs: Uniform Decode

**Uniform decode:** Same bits in each instruction correspond to the same meaning
- Opcode is always in the same location
- Ditto operand specifiers, immediate values, ...
- Many “RISC” ISAs: Alpha, ARM, MIPS, PowerPC, SPARC
  + Easier decode, simpler hardware
  + Enables parallelism: generate target address before knowing the instruction is a branch
  -- Restricts instruction format (fewer instructions?) or wastes space

**Non-uniform decode**
- E.g., opcode can be the 1st-7th byte in x86
  + More compact and powerful instruction format
  -- More complex decode logic (e.g., more logic to speculatively generate branch target)
x86 vs. Alpha Instruction Formats

- **x86:**

  ![x86 Instruction Formats Diagram]

- **Alpha:**

  ![Alpha Instruction Formats Diagram]
ISA-level Tradeoffs: Number of Registers

- Affects:
  - Number of bits used for encoding register address
  - Number of values kept in fast storage (register file)
  - (uarch) Size, access time, power consumption of register file

- Large number of registers:
  + Enables better register allocation (and optimizations) by compiler → fewer saves/restores
  - Larger instruction size
  - Larger register file size
  - (Superscalar processors) More complex dependency check logic
ISA-level Tradeoffs: Addressing Modes

- Addressing mode specifies how to obtain an operand of an instruction
  - Register
  - Immediate
  - Memory (displacement, register indirect, indexed, absolute, memory indirect, autoincrement, autodecrement, ...)

- More modes:
  + help better support programming constructs (arrays, pointer-based accesses)
    -- make it harder for the architect to design
    -- too many choices for the compiler?
      - Many ways to do the same thing complicates compiler design
      - Read Wulf, “Compilers and Computer Architecture”
x86 vs. Alpha Instruction Formats

- **x86:**
  - Instruction Prefixes: Up to four prefixes of 1 byte each (optional)
  - Opcode: 1-, 2-, or 3-byte opcode (if required)
  - ModR/M: 1 byte (if required)
  - SIB: 1 byte (if required)
  - Displacement: Address displacement of 1, 2, or 4 bytes or none
  - Immediate: Immediate data of 1, 2, or 4 bytes or none

- **Alpha:**
  - PALcode Format
  - Branch Format
  - Memory Format
  - Operate Format
### Table 2-2: 32-Bit Addressing Forms with the ModR/M Byte

<table>
<thead>
<tr>
<th>Effective Address</th>
<th>Mod</th>
<th>R/M</th>
<th>Value of ModR/M Byte (in Hexadecimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>00</td>
<td>00</td>
<td>08 10 18 20 28 30 38</td>
</tr>
<tr>
<td>ECX</td>
<td>01</td>
<td>01</td>
<td>09 11 19 21 29 31</td>
</tr>
<tr>
<td>EDX</td>
<td>01</td>
<td>02</td>
<td>0A 12 1A 22 2A 32</td>
</tr>
<tr>
<td>EBX</td>
<td>01</td>
<td>03</td>
<td>0B 13 1B 23 2B 33</td>
</tr>
<tr>
<td>[EAX] + disp8</td>
<td>01</td>
<td>04</td>
<td>0C 14 1C 24 2C 34</td>
</tr>
<tr>
<td>[ECX] + disp8</td>
<td>01</td>
<td>05</td>
<td>0D 15 1D 25 2D 35</td>
</tr>
<tr>
<td>[EDX] + disp8</td>
<td>01</td>
<td>06</td>
<td>0E 16 1E 26 2E 36</td>
</tr>
<tr>
<td>[EBX] + disp8</td>
<td>01</td>
<td>07</td>
<td>0F 17 1F 27 2F 37</td>
</tr>
<tr>
<td>[EAX] + disp32</td>
<td>10</td>
<td>00</td>
<td>88 90 98 A0 A8 B0 B8</td>
</tr>
<tr>
<td>[ECX] + disp32</td>
<td>10</td>
<td>01</td>
<td>89 91 99 A1 A9 B1 B9</td>
</tr>
<tr>
<td>[EDX] + disp32</td>
<td>10</td>
<td>02</td>
<td>8A 92 9A A2 AA B2 BA</td>
</tr>
<tr>
<td>[EBX] + disp32</td>
<td>10</td>
<td>03</td>
<td>8B 93 9B A3 AB B3 BB</td>
</tr>
<tr>
<td>[EAX] - disp32</td>
<td>10</td>
<td>04</td>
<td>8C 94 9C A4 AC B4 BC</td>
</tr>
<tr>
<td>[ECX] - disp32</td>
<td>10</td>
<td>05</td>
<td>8D 95 9D A5 AD B5 BD</td>
</tr>
<tr>
<td>[EDX] - disp32</td>
<td>10</td>
<td>06</td>
<td>8E 96 9E A6 AE B6 BE</td>
</tr>
<tr>
<td>[EBX] - disp32</td>
<td>10</td>
<td>07</td>
<td>8F 97 9F A7 AF B7 BF</td>
</tr>
<tr>
<td>EAX/X/AL/MMT/XMMO</td>
<td>11</td>
<td>00</td>
<td>C0 C8 D0 D8 E0 E8 F0 F8</td>
</tr>
<tr>
<td>ECX/CL/MM/MMM/XMM1</td>
<td>11</td>
<td>01</td>
<td>C1 C9 D1 D9 E1 E9 F1 F9</td>
</tr>
<tr>
<td>EDX/DL/MM/MMM/XMM2</td>
<td>11</td>
<td>02</td>
<td>C2 CA D2 DA E2 EA F2 FA</td>
</tr>
<tr>
<td>EBX/BL/MM/MMM/XMM3</td>
<td>11</td>
<td>03</td>
<td>C3 CB D3 DB E3 EB F3 FB</td>
</tr>
<tr>
<td>ESP/SP/HH/MM/MMM4</td>
<td>11</td>
<td>04</td>
<td>C4 CC D4 DC E4 EC F4 FC</td>
</tr>
<tr>
<td>EBP/SP/C/MM/MMM5</td>
<td>11</td>
<td>05</td>
<td>C5 CD D5 DD E5 ED F5 FD</td>
</tr>
<tr>
<td>ESI/SI/DH/MM/MMM6</td>
<td>11</td>
<td>06</td>
<td>C6 CE D6 DE E6 EE F6 FE</td>
</tr>
<tr>
<td>EDI/DI/BH/MM/MMM7</td>
<td>11</td>
<td>07</td>
<td>C7 CF D7 DF E7 EF F7 FF</td>
</tr>
</tbody>
</table>

**NOTES:**
1. The [ ]-[ ] nomenclature means a SIB follows the ModR/M byte.
2. The disp32 nomenclature denotes a 32-bit displacement that follows the ModR/M byte (or the SIB byte if one is present) and that is added to the index.
3. The disp8 nomenclature denotes an 8-bit displacement that follows the ModR/M byte (or the SIB byte if one is present) and that is sign-extended and added to the index.

Table 2-3 is organized to give 256 possible values of the SIB byte (in hexadecimal). General purpose registers used as a base are indicated across the top of the table.
### Table 2-3. 32-Bit Addressing Forms with the SIB Byte

<table>
<thead>
<tr>
<th>Scaled Index</th>
<th>SS</th>
<th>Index</th>
<th>Value of SIB Byte (in Hexadecimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[EAX]</td>
<td>00</td>
<td>000</td>
<td>00 01 02 03 04 05 06 07</td>
</tr>
<tr>
<td>[ECX]</td>
<td>00</td>
<td>001</td>
<td>08 09 10 11 12 13 14 15</td>
</tr>
<tr>
<td>[EDX]</td>
<td>00</td>
<td>010</td>
<td>16 17 18 19 20 21 22 23</td>
</tr>
<tr>
<td>[EBX]</td>
<td>00</td>
<td>011</td>
<td>24 25 26 27 28 29 30 31</td>
</tr>
<tr>
<td>none</td>
<td>00</td>
<td>100</td>
<td>32 33 34 35 36 37 38 39</td>
</tr>
<tr>
<td>[EBP]</td>
<td>00</td>
<td>101</td>
<td>40 41 42 43 44 45 46 47</td>
</tr>
<tr>
<td>[ESI]</td>
<td>00</td>
<td>110</td>
<td>50 51 52 53 54 55 56 57</td>
</tr>
<tr>
<td>[EDI]</td>
<td>00</td>
<td>111</td>
<td>60 61 62 63 64 65 66 67</td>
</tr>
<tr>
<td>[EAX*2]</td>
<td>00</td>
<td>001</td>
<td>70 71 72 73 74 75 76 77</td>
</tr>
<tr>
<td>[ECX*2]</td>
<td>00</td>
<td>010</td>
<td>80 81 82 83 84 85 86 87</td>
</tr>
<tr>
<td>[EDX*2]</td>
<td>00</td>
<td>011</td>
<td>90 91 92 93 94 95 96 97</td>
</tr>
<tr>
<td>[EBX*2]</td>
<td>00</td>
<td>100</td>
<td>A0 A1 A2 A3 A4 A5 A6 A7</td>
</tr>
<tr>
<td>none</td>
<td>00</td>
<td>101</td>
<td>B0 B1 B2 B3 B4 B5 B6 B7</td>
</tr>
<tr>
<td>[EBP*2]</td>
<td>00</td>
<td>110</td>
<td>C0 C1 C2 C3 C4 C5 C6 C7</td>
</tr>
<tr>
<td>[ESI*2]</td>
<td>00</td>
<td>111</td>
<td>D0 D1 D2 D3 D4 D5 D6 D7</td>
</tr>
<tr>
<td>[EDI*2]</td>
<td>00</td>
<td></td>
<td>E0 E1 E2 E3 E4 E5 E6 E7</td>
</tr>
<tr>
<td>[EAX*4]</td>
<td>00</td>
<td>001</td>
<td>F0 F1 F2 F3 F4 F5 F6 F7</td>
</tr>
<tr>
<td>[ECX*4]</td>
<td>00</td>
<td>010</td>
<td></td>
</tr>
<tr>
<td>[EDX*4]</td>
<td>00</td>
<td>011</td>
<td></td>
</tr>
<tr>
<td>[EBX*4]</td>
<td>00</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>none</td>
<td>00</td>
<td>101</td>
<td></td>
</tr>
<tr>
<td>[EBP*4]</td>
<td>00</td>
<td>110</td>
<td></td>
</tr>
<tr>
<td>[ESI*4]</td>
<td>00</td>
<td>111</td>
<td></td>
</tr>
<tr>
<td>[EDI*4]</td>
<td>00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[EAX*8]</td>
<td>00</td>
<td>000</td>
<td></td>
</tr>
<tr>
<td>[ECX*8]</td>
<td>00</td>
<td>001</td>
<td></td>
</tr>
<tr>
<td>[EDX*8]</td>
<td>00</td>
<td>010</td>
<td></td>
</tr>
<tr>
<td>[EBX*8]</td>
<td>00</td>
<td>011</td>
<td></td>
</tr>
<tr>
<td>none</td>
<td>00</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>[EBP*8]</td>
<td>00</td>
<td>101</td>
<td></td>
</tr>
<tr>
<td>[ESI*8]</td>
<td>00</td>
<td>110</td>
<td></td>
</tr>
<tr>
<td>[EDI*8]</td>
<td>00</td>
<td>111</td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**

1. The [*] nomenclature means a disp32 with no base if the MOD is 00B. Otherwise, [*] means disp8 or disp32 + [EBP]. This provides the following address modes:

<table>
<thead>
<tr>
<th>MOD bits</th>
<th>Effective Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>[scaled index] + disp32</td>
</tr>
<tr>
<td>01</td>
<td>[scaled index] + disp8 + [EBP]</td>
</tr>
<tr>
<td>10</td>
<td>[scaled index] + disp32 + [EBP]</td>
</tr>
</tbody>
</table>
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 vs. SIMD
- 0, 1, 2, 3 address machines (stack, accumulator, 2 or 3-operands)
- 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)
- ...

...
Programmer vs. (Micro)architect

- Many ISA features designed to aid programmers
- But, complicate the hardware designer’s job

- Virtual memory
  - vs. overlay programming
  - Should the programmer be concerned about the size of code blocks?

- Unaligned memory access
  - Compile/programmer needs to align data

- Transactional memory?

- VLIW vs. SIMD? Superscalar execution vs. SIMD?