Reminder: Homeworks for Next Two Weeks

- Homework 1
  - Due tonight, 11:59pm, on Blackboard
  - MIPS warmup, ISA concepts, basic performance evaluation

- Homework 2
  - Will be out today or early tomorrow
  - Due February 13
Reminder: Lab Assignment 1

- Due this Friday (Feb 3), at the end of Friday lab
A Note on the Primary Textbook

- 4th Edition
- Revised 4th Edition

- They are very similar → you are fine if you have either
- Problem statements are different
  - We will specify the full problems in the future
Review of Last Lecture

- ISA Principles and Tradeoffs

- Elements of the ISA
  - Sequencing model, instruction processing style
  - Instructions, data types, memory organization, registers, addressing modes, orthogonality, I/O device interfacing ...

- What is the benefit of autoincrement addressing mode?
- What is the downside of having an autoincrement addressing mode?
- Is the LC-3b ISA orthogonal?
  - Can all addressing modes be used with all instructions?
Is the LC-3b ISA Orthogonal?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Code</th>
<th>Field 1</th>
<th>Field 2</th>
<th>Field 3</th>
<th>Field 4</th>
<th>Field 5</th>
<th>Field 6</th>
<th>Field 7</th>
<th>Field 8</th>
<th>Field 9</th>
<th>Field 10</th>
<th>Field 11</th>
<th>Field 12</th>
<th>Field 13</th>
<th>Field 14</th>
<th>Field 15</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD*</td>
<td>0001</td>
<td>DR</td>
<td>SR1</td>
<td>A</td>
<td>op.spec</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND*</td>
<td>0101</td>
<td>DR</td>
<td>SR1</td>
<td>A</td>
<td>op.spec</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BR</td>
<td>0000</td>
<td>n</td>
<td>z</td>
<td>p</td>
<td>PCoffset9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JMP</td>
<td>1100</td>
<td>000</td>
<td>BaseR</td>
<td>000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JSR(R)</td>
<td>0100</td>
<td>A</td>
<td></td>
<td></td>
<td>operand specifier</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDB*</td>
<td>0010</td>
<td>DR</td>
<td>BaseR</td>
<td>boffset6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDW*</td>
<td>0110</td>
<td>DR</td>
<td>BaseR</td>
<td>offset6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LEA*</td>
<td>1110</td>
<td>DR</td>
<td></td>
<td>PCoffset9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RTI</td>
<td>1000</td>
<td></td>
<td></td>
<td>000000000000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHF*</td>
<td>1101</td>
<td>DR</td>
<td>SR</td>
<td>A</td>
<td>D</td>
<td>amount4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STB</td>
<td>0011</td>
<td>SR</td>
<td>BaseR</td>
<td>boffset6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STW</td>
<td>0111</td>
<td>SR</td>
<td>BaseR</td>
<td>offset6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRAP</td>
<td>1111</td>
<td>0000</td>
<td></td>
<td>trapvect8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XOR*</td>
<td>1001</td>
<td>DR</td>
<td>SR1</td>
<td>A</td>
<td>op.spec</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>not used</td>
<td>1010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>not used</td>
<td>1011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Complex vs. Simple Instructions

- Complex instruction: An instruction does a lot of work, e.g. many operations
  - Insert in a doubly linked list
  - Compute FFT
  - String copy

- Simple instruction: An instruction does small amount of work, it is a primitive
  - Add
  - XOR
  - Multiply
Complex vs. Simple Instructions

- Advantages of Complex instructions
  + Denser encoding → smaller code size → better memory utilization, saves off-chip bandwidth, better cache hit rate (better packing of instructions)
  + Simpler compiler: no need to optimize small instructions as much

- Disadvantages of Complex Instructions
  - Larger chunks of work → compiler has less opportunity to optimize
  - More complex hardware → translation from a high level to control signals and optimization needs to be done by hardware
  - Compiler is limited in fine-grained optimizations it can do
ISA-level Tradeoffs: Semantic Gap

- **Where to place the ISA?** Semantic gap
  - Closer to high-level language (HLL) → Small semantic gap, complex instructions
  - Closer to hardware control signals? → Large semantic gap, simple instructions

- **RISC vs. CISC machines**
  - FFT, QUICKSORT, POLY, FP instructions?
  - VAX INDEX instruction (array access with bounds checking)
ISA-level Tradeoffs: Semantic Gap

- Some tradeoffs (for you to think about)

- 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
An instruction operates on a string

- Move one string of arbitrary length to another location
- Compare two strings

Enabled by the ability to specify repeated execution of an instruction (in the ISA)

- Using a “prefix” called REP prefix

Example: REP MOVS instruction

- Only two bytes: REP prefix byte and MOVS opcode byte (F2 A4)
- Implicit source and destination registers pointing to the two strings (ESI, EDI)
- Implicit count register (ECX) specifies how long the string is
REP MOVS (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 MIPS?
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
      - 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
How High or Low Can You Go?

- **Very large semantic gap**
  - Each instruction specifies the complete set of control signals in the machine
  - Compiler generates control signals
  - Open microcode (John Cocke, circa 1970s)
    - Gave way to optimizing compilers

- **Very small semantic gap**
  - ISA is the same as high-level language
  - Java machines, LISP machines, object-oriented machines, capability-based machines
A Note on ISA Evolution

- ISAs have evolved to reflect/satisfy the concerns of the day

- Examples:
  - Limited memory size
  - Limited compiler optimization technology
  - Limited memory bandwidth
  - Need for specialization in important applications (e.g., MMX)

- Use of translation (in HW and SW) enabled underlying implementations to be similar, regardless of the ISA
  - Concept of dynamic/static interface
  - Contrast it with hardware/software interface
Effect of Translation

- One can translate from one ISA to another ISA to change the semantic gap tradeoffs

- Examples
  - Intel’s and AMD’s x86 implementations translate x86 instructions into programmer-invisible microoperations (simple instructions) in hardware
  - Transmeta’s x86 implementations translated x86 instructions into “secret” VLIW instructions in software (code morphing software)

- Think about the tradeoffs
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
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, MIPS, 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
x86 vs. Alpha Instruction Formats

- **x86:**

<table>
<thead>
<tr>
<th>Instruction Prefixes</th>
<th>Opcode</th>
<th>ModR/M</th>
<th>SIB</th>
<th>Displacement</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1-, 2-, or 3-byte opcode</td>
<td>1 byte (if required)</td>
<td>1 byte (if required)</td>
<td>Address displacement of 1, 2, or 4 bytes or none</td>
<td>Immediate data of 1, 2, or 4 bytes or none</td>
</tr>
</tbody>
</table>

- **Alpha:**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>RA</td>
<td>Disp</td>
</tr>
<tr>
<td>RA</td>
<td>RB</td>
</tr>
<tr>
<td>RA</td>
<td>RB</td>
</tr>
</tbody>
</table>
MIPS Instruction Format

- **R-type**, 3 register operands

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6-bit</td>
<td>5-bit</td>
<td>5-bit</td>
<td>5-bit</td>
<td>5-bit</td>
<td>6-bit</td>
</tr>
</tbody>
</table>
```

- **I-type**, 2 register operands and 16-bit immediate operand

```
<table>
<thead>
<tr>
<th></th>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>6-bit</td>
<td>5-bit</td>
<td>5-bit</td>
<td>16-bit</td>
<td></td>
</tr>
</tbody>
</table>
```

- **J-type**, 26-bit immediate operand

```
<table>
<thead>
<tr>
<th></th>
<th>opcode</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>6-bit</td>
<td>26-bit</td>
<td></td>
</tr>
</tbody>
</table>
```

- **Simple Decoding**
  - 4 bytes per instruction, regardless of format
  - must be 4-byte aligned (2 lsb of PC must be 2b’00)
  - format and fields easy to extract in hardware
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
Review: 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:**

<table>
<thead>
<tr>
<th>Instruction Prefixes</th>
<th>Opcode</th>
<th>ModR/M</th>
<th>SIB</th>
<th>Displacement</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Up to four prefixes of 1 byte each (optional)</td>
<td>1-, 2-, or 3-byte opcode</td>
<td>1 byte (if required)</td>
<td>1 byte (if required)</td>
<td>Address displacement of 1, 2, or 4 bytes or none</td>
<td>Immediate data of 1, 2, or 4 bytes or none</td>
</tr>
</tbody>
</table>

- **Alpha:**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>RA</td>
<td>Disp</td>
</tr>
<tr>
<td>RA</td>
<td>RB</td>
</tr>
<tr>
<td>RA</td>
<td>RB</td>
</tr>
</tbody>
</table>
### 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 39</td>
</tr>
<tr>
<td>[EDX]</td>
<td>02</td>
<td>02</td>
<td>0A 12 1A 22 2A 32 3A</td>
</tr>
<tr>
<td>[EBX]</td>
<td>03</td>
<td>03</td>
<td>0B 13 1B 23 2B 33 3B</td>
</tr>
<tr>
<td>[-][I][-]</td>
<td>04</td>
<td>04</td>
<td>0C 14 1C 24 2C 34 3C</td>
</tr>
<tr>
<td>disp32</td>
<td>05</td>
<td>05</td>
<td>0D 15 1D 25 2D 35 3D</td>
</tr>
<tr>
<td>[ESI]</td>
<td>06</td>
<td>06</td>
<td>0E 16 1E 26 2E 36 3E</td>
</tr>
<tr>
<td>[EDI]</td>
<td>07</td>
<td>07</td>
<td>0F 17 1F 27 2F 37 3F</td>
</tr>
<tr>
<td>[EAX]+disp8</td>
<td>08</td>
<td>08</td>
<td>09 10 18 20 28 30 38</td>
</tr>
<tr>
<td>[ECX]+disp8</td>
<td>09</td>
<td>09</td>
<td>0A 11 19 21 29 31 39</td>
</tr>
<tr>
<td>[EDX]+disp8</td>
<td>0A</td>
<td>0A</td>
<td>0B 12 1A 22 2A 32 3A</td>
</tr>
<tr>
<td>[EBX]+disp8</td>
<td>0B</td>
<td>0B</td>
<td>0C 13 1B 23 2B 33 3B</td>
</tr>
<tr>
<td>[-][I][-]+disp8</td>
<td>0C</td>
<td>0C</td>
<td>0D 14 1C 24 2C 34 3C</td>
</tr>
<tr>
<td>disp32+disp8</td>
<td>0D</td>
<td>0D</td>
<td>0E 15 1E 25 2E 35 3E</td>
</tr>
<tr>
<td>[ESI]+disp8</td>
<td>0E</td>
<td>0E</td>
<td>0F 16 1F 26 2F 36 3F</td>
</tr>
<tr>
<td>[EDI]+disp8</td>
<td>0F</td>
<td>0F</td>
<td>10 17 18 27 28 37 38</td>
</tr>
<tr>
<td>[EAX]+disp32</td>
<td>10</td>
<td>10</td>
<td>0A 12 14 16 18 20 22 24 26 28 30 32 34 36 38</td>
</tr>
<tr>
<td>[ECX]+disp32</td>
<td>11</td>
<td>11</td>
<td>0B 13 15 17 19 21 23 25 27 29 31 33 35 37 39</td>
</tr>
<tr>
<td>[EDX]+disp32</td>
<td>12</td>
<td>12</td>
<td>0C 14 16 18 20 22 24 26 28 30 32 34 36 38</td>
</tr>
<tr>
<td>[EBX]+disp32</td>
<td>13</td>
<td>13</td>
<td>0D 15 17 19 21 23 25 27 29 31 33 35 37 39</td>
</tr>
<tr>
<td>[-][I][-]+disp32</td>
<td>14</td>
<td>14</td>
<td>0E 16 18 20 22 24 26 28 30 32 34 36 38</td>
</tr>
<tr>
<td>disp32+disp32</td>
<td>15</td>
<td>15</td>
<td>0F 17 19 21 23 25 27 29 31 33 35 37 39</td>
</tr>
<tr>
<td>[ESI]+disp32</td>
<td>16</td>
<td>16</td>
<td>10 18 20 22 24 26 28 30 32 34 36 38</td>
</tr>
<tr>
<td>[EDI]+disp32</td>
<td>17</td>
<td>17</td>
<td>11 19 21 23 25 27 29 31 33 35 37 39</td>
</tr>
</tbody>
</table>

**NOTES:**
1. The [-][I] 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></th>
<th>SS</th>
<th>Index</th>
<th>Value of SIB Byte (in Hexadecimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[EAX*2]</td>
<td>01</td>
<td>00 01 02 03 04 05 06 07</td>
<td></td>
</tr>
<tr>
<td>[ECX*2]</td>
<td>01</td>
<td>08 09 0A 0B 0C 0D 0E 0F</td>
<td></td>
</tr>
<tr>
<td>[EDX*2]</td>
<td>01</td>
<td>10 11 12 13 14 15 16 17</td>
<td></td>
</tr>
<tr>
<td>[EBX*2]</td>
<td>01</td>
<td>18 19 1A 1B 1C 1D 1E 1F</td>
<td></td>
</tr>
<tr>
<td>none</td>
<td>10</td>
<td>20 21 22 23 24 25 26 27</td>
<td></td>
</tr>
<tr>
<td>[EBP*2]</td>
<td>10</td>
<td>28 29 2A 2B 2C 2D 2E 2F</td>
<td></td>
</tr>
<tr>
<td>[ESI*2]</td>
<td>10</td>
<td>30 31 32 33 34 35 36 37</td>
<td></td>
</tr>
<tr>
<td>[EDI*2]</td>
<td>10</td>
<td>38 39 3A 3B 3C 3D 3E 3F</td>
<td></td>
</tr>
<tr>
<td>[EAX*4]</td>
<td>10</td>
<td>40 41 42 43 44 45 46 47</td>
<td></td>
</tr>
<tr>
<td>[ECX*4]</td>
<td>10</td>
<td>48 49 4A 4B 4C 4D 4E 4F</td>
<td></td>
</tr>
<tr>
<td>[EDX*4]</td>
<td>10</td>
<td>50 51 52 53 54 55 56 57</td>
<td></td>
</tr>
<tr>
<td>[EBX*4]</td>
<td>10</td>
<td>58 59 5A 5B 5C 5D 5E 5F</td>
<td></td>
</tr>
<tr>
<td>none</td>
<td>10</td>
<td>60 61 62 63 64 65 66 67</td>
<td></td>
</tr>
<tr>
<td>[EBP*4]</td>
<td>10</td>
<td>68 69 6A 6B 6C 6D 6E 6F</td>
<td></td>
</tr>
<tr>
<td>[ESI*4]</td>
<td>10</td>
<td>70 71 72 73 74 75 76 77</td>
<td></td>
</tr>
<tr>
<td>[EDI*4]</td>
<td>10</td>
<td>78 79 7A 7B 7C 7D 7E 7F</td>
<td></td>
</tr>
<tr>
<td>[EAX*8]</td>
<td>11</td>
<td>80 81 82 83 84 85 86 87</td>
<td></td>
</tr>
<tr>
<td>[ECX*8]</td>
<td>11</td>
<td>88 89 8A 8B 8C 8D 8E 8F</td>
<td></td>
</tr>
<tr>
<td>[EDX*8]</td>
<td>11</td>
<td>90 91 92 93 94 95 96 97</td>
<td></td>
</tr>
<tr>
<td>[EBX*8]</td>
<td>11</td>
<td>98 99 9A 9B 9C 9D 9E 9F</td>
<td></td>
</tr>
<tr>
<td>none</td>
<td>11</td>
<td>A0 A1 A2 A3 A4 A5 A6 A7</td>
<td></td>
</tr>
<tr>
<td>[EBP*8]</td>
<td>11</td>
<td>A8 A9 AA AB AC AD AE AF</td>
<td></td>
</tr>
<tr>
<td>[ESI*8]</td>
<td>11</td>
<td>B0 B1 B2 B3 B4 B5 B6 B7</td>
<td></td>
</tr>
<tr>
<td>[EDI*8]</td>
<td>11</td>
<td>B8 B9 BA BB BC BD BE BF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>C0 C1 C2 C3 C4 C5 C6 C7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>C8 C9 CA CB CC CD CE CF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>D0 D1 D2 D3 D4 D5 D6 D7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>DA DC DD DE DF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>E0 E1 E2 E3 E4 E5 E6 E7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>EA EB EC ED EE EF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>FA FB FC FD FE FF</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:

   MOD bits Effective Address
   00      [scaled index] + disp32
   01      [scaled index] + disp8 + [EBP]
   10      [scaled index] + disp32 + [EBP]
X86 SIB-D Addressing Mode

Figure 3-11. Offset (or Effective Address) Computation

Offset = Base + (Index * Scale) + Displacement

x86 Manual Vol. 1, page 3-31 -- see course resources on website
• **(Index * Scale) + Displacement** — This address mode offers an efficient way to index into a static array when the element size is 2, 4, or 8 bytes. The displacement locates the beginning of the array, the index register holds the subscript of the desired array element, and the processor automatically converts the subscript into an index by applying the scaling factor.

• **Base + Index + Displacement** — Using two registers together supports either a two-dimensional array (the displacement holds the address of the beginning of the array) or one of several instances of an array of records (the displacement is an offset to a field within the record).

• **Base + (Index * Scale) + Displacement** — Using all the addressing components together allows efficient indexing of a two-dimensional array when the elements of the array are 2, 4, or 8 bytes in size.
Other Example ISA-level Tradeoffs

- Condition codes vs. not
- VLIW vs. single instruction
- Precise vs. imprecise exceptions
- Virtual memory vs. not
- Unaligned access vs. not
- Hardware interlocks vs. software-guaranteed interlocking
- Software vs. hardware managed page fault handling
- Cache coherence (hardware vs. software)
- ...

...
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 fitting physical memory?

Addressing modes

Unaligned memory access
- Compile/programmer needs to align data
MIPS: Aligned Access

- LW/SW alignment restriction: 4-byte word-alignment
  - not designed to fetch memory bytes not within a word boundary
  - not designed to rotate unaligned bytes into registers
- Provide separate opcodes for the “infrequent” case

- LWL  rd 6(r0) → byte-6 byte-5 byte-4 D
- LWR  rd 3(r0) → byte-6 byte-5 byte-4 byte-3

- LWL/LWR is slower
- Note LWL and LWR still fetch within word boundary
X86: Unaligned Access

- LD/ST instructions automatically align data that spans a “word” boundary
- Programmer/compiler does not need to worry about where data is stored (whether or not in a word-aligned location)

4.1.1 Alignment of Words, Doublewords, Quadwords, and Double Quadwords

Words, doublewords, and quadwords do not need to be aligned in memory on natural boundaries. The natural boundaries for words, double words, and quadwords are even-numbered addresses, addresses evenly divisible by four, and addresses evenly divisible by eight, respectively. However, to improve the performance of programs, data structures (especially stacks) should be aligned on natural boundaries whenever possible. The reason for this is that the processor requires two memory accesses to make an unaligned memory access; aligned accesses require only one memory access. A word or doubleword operand that crosses a 4-byte boundary or a quadword operand that crosses an 8-byte boundary is considered unaligned and requires two separate memory bus cycles for access.
Figure 4-2. Bytes, Words, Doublewords, Quadwords, and Double Quadwords in Memory

X86: Unaligned Access
Aligned vs. Unaligned Access

- Pros of having no restrictions on alignment

- Cons of having no restrictions on alignment

- Filling in the above: an exercise for you...
Implementing the ISA: Microarchitecture Basics
How Does a Machine Process Instructions?

- What does processing an instruction mean?
- Remember the von Neumann model

\[
\text{AS} = \text{Architectural (programmer visible) state before an instruction is processed}
\]

\[
\text{Process instruction}
\]

\[
\text{AS}' = \text{Architectural (programmer visible) state after an instruction is processed}
\]

- Processing an instruction: Transforming A to A’ according to the ISA specification of the instruction
The “Process instruction” Step

- ISA specifies abstractly what $A'$ should be, given an instruction and $A$
  - It defines an abstract finite state machine where
    - State = programmer-visible state
    - Next-state logic = instruction execution specification
  - From ISA point of view, there are no “intermediate states” between $A$ and $A'$ during instruction execution
    - One state transition per instruction

- Microarchitecture implements how $A$ is transformed to $A'$
  - There are many choices in implementation
  - We can have programmer-invisible state to optimize the speed of instruction execution: multiple state transitions per instruction
    - Choice 1: $AS \rightarrow AS'$ (transform $A$ to $A'$ in a single clock cycle)
    - Choice 2: $AS \rightarrow AS+MS1 \rightarrow AS+MS2 \rightarrow AS+MS3 \rightarrow AS'$ (take multiple clock cycles to transform $AS$ to $AS'$)
A Very Basic Instruction Processing Engine

- Each instruction takes a single clock cycle to execute
- Only combinational logic is used to implement instruction execution
  - No intermediate, programmer-invisible state updates

\[
AS = \text{Architectural (programmer visible) state at the beginning of a clock cycle}
\]

\[
\text{Process instruction in one clock cycle}
\]

\[
AS' = \text{Architectural (programmer visible) state at the end of a clock cycle}
\]
A Very Basic Instruction Processing Engine

- Single-cycle machine

- What is the clock cycle determined by?