Plan for Today

- Announcements
  - Readings are extremely important!
  - No class meeting next Monday
- Questions
- Commentaries
- A few remaining thoughts from last week
- Instruction set architecture discussion
- Assignment
Commentaries

• Are brief essays that:
  – Demonstrate that you’ve read and understood the assigned reading
  – Ideally: express a strong opinion, either critical or supportive

• Should be approximately one page in length

• Need to be submitted to discussion site before class
ISA: The Programmer’s Interface

• Designer’s Question: *what does the programmer need?*

• Instruction/command Set
  – Available operations (RISC vs. CISC)
  – Branching behavior
  – Valid instruction sequences

• Registers
  – Sizes
  – Data types
  – Organization (flat, windows, hierarchical)

• Addressing Modes
  – Register, Immediate, Indirect
  – Offset, scaling
Interface vs. Implementation

- **ISA**: the programmer’s interface

- *Implementation*: the underlying resources and organization

- Ideally:
  - Programmer needs only ISA to write good programs

- In reality:
  - Programmers look into the implementation to improve performance

- Consequences?
• **Programmer-Visible State**
  – EIP (Program Counter)
    • Address of next instruction
  – Register File
    • Heavily used program data
  – Condition Codes
    • Store status information about most recent arithmetic operation
    • Used for conditional branching

• Memory
  • Byte addressable array
  • Code, user data, (some) OS data
  • Includes stack used to support procedures
Turning C into Object Code

- Code in files p1.c p2.c
- Compile with command: gcc -O p1.c p2.c -o p
  - Use optimizations (-O)
  - Put resulting binary in file p

![Diagram of the compilation process]

- C program (p1.c p2.c)
- Compiler (gcc -S)
- Asm program (p1.s p2.s)
- Assembler (gcc or as)
- Object program (p1.o p2.o)
- Linker (gcc or ld)
- Executable program (p)
- Static libraries (.a)
Compiling Into Assembly

**C Code**

```c
int sum(int x, int y) {
    int t = x+y;
    return t;
}
```

**Generated Assembly**

```assembly
_sum:
pushl %ebp
movl %esp,%ebp
movl 12(%ebp),%eax
addl 8(%ebp),%eax
movl %ebp,%esp
popl %ebp
ret
```

Obtain with command

```
gcc -O -S code.c
```

Produces file `code.s`
Assembly Characteristics

• Minimal Data Types
  – “Integer” data of 1, 2, or 4 bytes
    • Data values
    • Addresses (untyped pointers)
  – Floating point data of 4, 8, or 10 bytes
  – No aggregate types such as arrays or structures
    • Just contiguously allocated bytes in memory

• Primitive Operations
  – Perform arithmetic function on register or memory data
  – Transfer data between memory and register
    • Load data from memory into register
    • Store register data into memory
  – Transfer control
    • Unconditional jumps to/from procedures
    • Conditional branches
Object Code

Code for `sum`

0x401040 <sum>:

- **Assembler**
  - Translates `.s` into `.o`
  - Binary encoding of each instruction
  - Nearly-complete image of executable code
  - Missing linkages between code in different files

- **Linker**
  - Resolves references between files
  - Combines with static run-time libraries
    - E.g., code for `malloc`, `printf`
  - Some libraries are *dynamically linked*
    - Linking occurs when program begins execution

- Total of 13 bytes
- Each instruction 1, 2, or 3 bytes
- Starts at address 0x401040

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Hex Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x55</td>
<td></td>
</tr>
<tr>
<td>0x89</td>
<td></td>
</tr>
<tr>
<td>0xe5</td>
<td></td>
</tr>
<tr>
<td>0x8b</td>
<td></td>
</tr>
<tr>
<td>0x45</td>
<td></td>
</tr>
<tr>
<td>0x0c</td>
<td></td>
</tr>
<tr>
<td>0x03</td>
<td></td>
</tr>
<tr>
<td>0x45</td>
<td></td>
</tr>
<tr>
<td>0x08</td>
<td></td>
</tr>
<tr>
<td>0x89</td>
<td></td>
</tr>
<tr>
<td>0xec</td>
<td></td>
</tr>
<tr>
<td>0x5d</td>
<td></td>
</tr>
<tr>
<td>0xc3</td>
<td></td>
</tr>
</tbody>
</table>
**x86 Machine Instruction Example**

- **C Code**
  - Add two signed integers

  ```c
  int t = x+y;
  ```

- **Assembly**
  - Add 2 4-byte integers
  - “Long” words in GCC parlance
  - Same instruction whether signed or unsigned

  ```asm
  addl 8(%ebp),%eax
  ```

  Similar to expression:
  ```c
  x += y
  ```

  Or
  ```c
  int eax;
  int *ebp;
  eax += ebp[2]
  ```

- **Object Code**
  - 3-byte instruction
  - Stored at address 0x401046

  ```asm
  0x401046:   03 45 08
  ```
Disassembling Object Code

Disassembled

<table>
<thead>
<tr>
<th>Offset</th>
<th>Opcode</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>55</td>
<td>push %ebp</td>
</tr>
<tr>
<td>1</td>
<td>89 e5</td>
<td>mov %esp,%ebp</td>
</tr>
<tr>
<td>3</td>
<td>8b 45 0c</td>
<td>mov 0xc(%ebp),%eax</td>
</tr>
<tr>
<td>6</td>
<td>03 45 08</td>
<td>add 0x8(%ebp),%eax</td>
</tr>
<tr>
<td>9</td>
<td>89 ec</td>
<td>mov %ebp,%esp</td>
</tr>
<tr>
<td>b</td>
<td>5d</td>
<td>pop %ebp</td>
</tr>
<tr>
<td>c</td>
<td>c3</td>
<td>ret</td>
</tr>
<tr>
<td>d</td>
<td>8d 76 00</td>
<td>lea 0x0(%esi),%esi</td>
</tr>
</tbody>
</table>

- Dis assembler
  - objdump -d p
  - Useful tool for examining object code
  - Analyzes bit pattern of series of instructions
  - Produces approximate rendition of assembly code
  - Can be run on either a .out (complete executable) or .o file
Alternate Disassembly with gdb

Object

Disassembled

0x401040 <sum>: push %ebp
0x401041 <sum+1>: mov %esp,%ebp
0x401043 <sum+3>: mov 0xc(%ebp),%eax
0x401046 <sum+6>: add 0x8(%ebp),%eax
0x401049 <sum+9>: mov %ebp,%esp
0x40104b <sum+11>: pop %ebp
0x40104c <sum+12>: ret
0x40104d <sum+13>: lea 0x0(%esi),%esi

• Within gdb Debugger
  gdb p
disassemble sum
  – Disassemble procedure
  x/13b sum
  – Examine the 13 bytes starting at sum
What Can be Disassembled?

Anything that can be interpreted as executable code
Disassembler examines bytes and reconstructs assembly source
ALU-centric ISA Taxonomy
Simple vs. Complex Operations

• RISC: reduced instruction set computer (plus load/store arch)
  – Small number of instructions
  – Easy to encode
  – Poor code density
  – Compiler and implementation friendly

• CISC: complex instruction set computer
  – Many instructions
  – Good for special, hand-coded cases
  – Good code density
  – Complicates compilation and implementation
Framing Your Perspective

• If you were to design an ISA of your own, what decisions must be made?
<table>
<thead>
<tr>
<th>Type</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>Arithmetic, logic</td>
</tr>
<tr>
<td>Memory</td>
<td>Loads and stores</td>
</tr>
<tr>
<td>Control</td>
<td>Branch, jump, calls returns, traps</td>
</tr>
<tr>
<td>System</td>
<td>OS call, privileged system management</td>
</tr>
<tr>
<td>F.P.</td>
<td>Floating point arithmetic</td>
</tr>
<tr>
<td>Decimal</td>
<td>Decimal ops</td>
</tr>
<tr>
<td>String</td>
<td>Copy, compare, search</td>
</tr>
<tr>
<td>Multimedia</td>
<td>Pixel and vertex ops</td>
</tr>
</tbody>
</table>
Encoding Instructions

- **Flexibility**
  - Many registers
  - Multiple addressing options

- **Compactness**
  - Small instructions

- **Ease of implementation**
  - Instructions represent *control*
## Encoding Examples

<table>
<thead>
<tr>
<th>Operation and no. of operands</th>
<th>Address specifier 1</th>
<th>Address field 1</th>
<th>...</th>
<th>Address specifier</th>
<th>Address field</th>
</tr>
</thead>
</table>

(a) Variable (e.g., VAX, Intel 80x86)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Address field 1</th>
<th>Address field 2</th>
<th>Address field 3</th>
</tr>
</thead>
</table>

(b) Fixed (e.g., Alpha, ARM, MIPS, PowerPC, SPARC, SuperH)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Address specifier</th>
<th>Address field</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Address specifier 1</th>
<th>Address specifier 2</th>
<th>Address field</th>
</tr>
</thead>
</table>

(c) Hybrid (e.g., IBM 360/70, MIPS16, Thumb, TI TMS320C54x)

© 2003 Elsevier Science (USA). All rights reserved.
To design and encode, consider

• Characteristics of “representative” programs
  – What operations get used most?
  – What data types are most common?
  – What branch and displacement distances are used?
## x86 SPECInt92 Operation Frequency

<table>
<thead>
<tr>
<th>Operation</th>
<th>Avg Int %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>22</td>
</tr>
<tr>
<td>Cond branch</td>
<td>20</td>
</tr>
<tr>
<td>Compare</td>
<td>16</td>
</tr>
<tr>
<td>Store</td>
<td>12</td>
</tr>
<tr>
<td>Add</td>
<td>8</td>
</tr>
<tr>
<td>And</td>
<td>6</td>
</tr>
<tr>
<td>Sub</td>
<td>5</td>
</tr>
<tr>
<td>Move</td>
<td>4</td>
</tr>
<tr>
<td>Call</td>
<td>1</td>
</tr>
<tr>
<td>Return</td>
<td>1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>96</strong></td>
</tr>
</tbody>
</table>
Control Flow Operations

• Conditional branches (usually PC-relative)
  – Condition codes
  – Condition register
  – Compare and branch

• Jumps (e.g., unconditional branches)

• Procedure calls and returns
  – Who saves what (return address, registers, params, etc.),
    where (stack, register) and when (caller, callee)
  – Combination of HW and SW conventions
Control Flow Op. Frequency

- Call/return: 8% (Floating-point) vs. 19% (Integer)
- Jump: 10% (Floating-point) vs. 6% (Integer)
- Conditional branch: 82% (Floating-point) vs. 75% (Integer)

© 2003 Elsevier Science (USA). All rights reserved.
Addressing for Control Flow

- Register indirect (for dynamic targets)
- PC-relative displacement

![Graph showing percentage of distance vs. bits of branch displacement. The graph compares integer and floating-point averages.]
Unconventional Operations

- Decimal operations (HP-PA, F-P is used in x86)
- String operations (x86)
- Lack of byte instructions (early Alpha)
- Synchronization (atomic swap, “fence”)
- Predicated execution (conditional moves)
- Cache hints (prefetch, flush)
- TLB instructions (TLB miss handled by software in MIPS)
- Multimedia (Sparc, MMX)
- Bit manipulation (Intel IXP)
- Thread management (SMT)
Registers

• Types of registers
  – Integer (often 32 in number)
  – Floating-point (often 32)
  – Special GPRs
    • Stack pointer, frame pointer, the PC (VAX)
  – Special purpose registers
    • Control registers, segment registers (x86)

• Organization
  – Flat
  – Windows (Sparc)
  – Hierarchy (Cray)
Addressing and Byte Ordering

• How are bytes ordered in multi-byte data objects?

• Example: 32-bit integer at address 0x100
  
  - \([x_{31}, x_{30}, \ldots, x_0]\) when broken into bytes,
  
  - \([x_{31}, \ldots, x_{24}], [x_{23}, \ldots, x_{16}], [x_{15}, \ldots, x_8], [x_7, \ldots, x_0]\)

<table>
<thead>
<tr>
<th>Big-endian</th>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>([x_{31}, \ldots, x_{24}])</td>
<td>([x_{23}, \ldots, x_{16}])</td>
<td>([x_{15}, \ldots, x_8])</td>
<td>([x_7, \ldots, x_0])</td>
</tr>
<tr>
<td>Little-endian</td>
<td>([x_7, \ldots, x_0])</td>
<td>([x_{15}, \ldots, x_8])</td>
<td>([x_{23}, \ldots, x_{16}])</td>
<td>([x_{31}, \ldots, x_{24}])</td>
</tr>
</tbody>
</table>

“Big-endian: most significant byte at lower address.”
Taxonomy of Addressing Modes

- General purpose
  - Register, immediate, absolute and displacement

- Often useful
  - Indexed, scaled-index

- Special purpose or obsolete
  - Memory indirect, auto-increment and -decrement

- Immediate, displacement, and absolute have a distinct cost
  - i.e., occupy bits in instruction

<table>
<thead>
<tr>
<th>Mode</th>
<th>Example</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>Add R4,R3</td>
<td>Values in registers</td>
</tr>
<tr>
<td>Immediate</td>
<td>Add R4,#3</td>
<td>Static constants</td>
</tr>
<tr>
<td>Displacement</td>
<td>Add R4,100(R1)</td>
<td>Local variables</td>
</tr>
<tr>
<td>Reg indirect</td>
<td>Add R4,(R1)</td>
<td>Pointers, calc addresses</td>
</tr>
<tr>
<td>Indexed</td>
<td>Add R3,(R1+R2)</td>
<td>R1 array base, R2 index</td>
</tr>
<tr>
<td>Absolute</td>
<td>Add R1,(1001)</td>
<td>Static data</td>
</tr>
</tbody>
</table>
Displacement Distance for Data

© 2003 Elsevier Science (USA). All rights reserved.
Frequency of Immediates

- Loads: Floating-point average 22%, Integer average 23%
- ALU operations: Floating-point average 19%, Integer average 25%
- All instructions: Floating-point average 16%, Integer average 21%
Size of Immediates

Percentage of immediates

Number of bits needed for immediate

Floating-point average

Integer average

© 2003 Elsevier Science (USA). All rights reserved.
New Architectures

• Memory addressing
  – Modes:
    • Immediate of 8-16 bits
    • displacement (subsumes register indirect) of 12-16 bits

• Instructions
  – Simple: load, store, add, subtract, move register-register and shift
  –Operand types
    • Integer: 8-, 16-, 32- and 64-bit
    • Floating point: 32- and 64-bit
  – Control flow: PC-relative branch displacement of 8 bits, register indirect for procedure calls and returns
Additional Topics

• VLIW – very long instruction word architectures

• Multithreading

• Exception handling
Assignment

• No class meeting next Monday

• Readings
  – For Wednesday
    • H&P: Appendix A, sections A1- A.2
    • V&L: Ch. 3
  – For Monday (Sept 14)
    • HW1 will be assigned, available on the course calendar
    • Revisit VHDL materials