CMSC 611: Advanced Computer Architecture

Instruction Set Architecture
Lecture Overview

• Last Week
  – Different performance metrics (response time, throughput, CPU time)
  – Performance reports, summary and comparison (Experiment reproducibility, arithmetic and weighted arithmetic means)
  – Widely used benchmark programs (SPEC, Whetstone, and Dhrystone)
  – Example industry metrics (e.g. MIPS, MFLOP, etc.)

• This Week
  – Classifications of instruction set architectures
  – Different addressing modes
  – Instruction types, operands and operations
To command a computer's hardware, you must speak its language.

Instructions: the “words” of a machine's language

Instruction set: its “vocabulary

The MIPS instruction set is used as a case study.
Instruction Set Architecture

• Once you learn one machine language, it is easy to pick up others:
  – Common fundamental operations
  – All designer have the same goals: simplify building hardware, maximize performance, minimize cost

• Goals:
  – Introduce design alternatives
  – Present a taxonomy of ISA alternatives
    • + some qualitative assessment of pros and cons
  – Present and analyze some instruction set measurements
  – Address the issue of languages and compilers and their bearing on instruction set architecture
  – Show some example ISA’s
A good interface:
- Lasts through many implementations (portability, compatibility)
- Is used in many different ways (generality)
- Provides convenient functionality to higher levels
- Permits an efficient implementation at lower levels

Design decisions must take into account:
- Technology
- Machine organization
- Programming languages
- Compiler technology
- Operating systems
Memory ISAs

• Terms
  – Result = Operand <operation> Operand

• Stack
  – Operate on top stack elements, push result back on stack

• Memory-Memory
  – Operands (and possibly also result) in memory
Register ISAs

• Accumulator Architecture
  – Common in early stored-program computers when hardware was expensive
  – Machine has only one register (accumulator) involved in all math & logic operations
  – Accumulator = Accumulator op Memory

• Extended Accumulator Architecture (8086)
  – Dedicated registers for specific operations, e.g. stack and array index registers, added

• General-Purpose Register Architecture (MIPS)
  – Register flexibility
  – Can further divide these into:
    • Register-memory: allows for one operand to be in memory
    • Register-register (load-store): all operands in registers
ISA Operations

(a) Stack
(b) Accumulator
(c) Register-memory
(d) Register-register/load-store
Famous ISA

- Stack
- Memory-Memory
- Accumulator Architecture
- Extended Accumulator Architecture
- General-Purpose Register Architecture

<table>
<thead>
<tr>
<th>Machine</th>
<th># general-purpose registers</th>
<th>Architecture style</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>Motorola 6800</td>
<td>2</td>
<td>Accumulator</td>
<td>1974</td>
</tr>
<tr>
<td>DEC VAX</td>
<td>16</td>
<td>Register-memory, memory-memory</td>
<td>1977</td>
</tr>
<tr>
<td>Intel 8086</td>
<td>1</td>
<td>Extended accumulator</td>
<td>1978</td>
</tr>
<tr>
<td>Motorola 68000</td>
<td>16</td>
<td>Register-memory</td>
<td>1980</td>
</tr>
<tr>
<td>Intel 80386</td>
<td>32</td>
<td>Register-memory</td>
<td>1985</td>
</tr>
<tr>
<td>PowerPC</td>
<td>32</td>
<td>Load-store</td>
<td>1992</td>
</tr>
<tr>
<td>DEC Alpha</td>
<td>32</td>
<td>Load-store</td>
<td>1992</td>
</tr>
</tbody>
</table>
High-Level-Language Architecture

- In the 1960s, systems software was rarely written in high-level languages
  - virtually every commercial operating system before Unix was written in assembly
- Some people blamed the code density on the instruction set rather than the programming language
- A machine design philosophy advocated making the hardware more like high-level languages
- The effectiveness of high-level languages, memory size limitation and lack of efficient compilers doomed this philosophy to a historical footnote
Other types of Architecture

• Reduced Instruction Set Architecture
  – With the recent development in compiler technology and expanded memory sizes less programmers are using assembly level coding
  – Drives ISA to favor benefit for compilers over ease of manual programming

• RISC architecture favors simplified hardware design over rich instruction set
  – Rely on compilers to perform complex operations

• Virtually all new architecture since 1982 follows the RISC philosophy:
  – fixed instruction lengths, load-store operations, and limited addressing mode
Compact Code

• Scarce memory or limited transmit time (JVM)
• Variable-length instructions (Intel 80x86)
  – Match instruction length to operand specification
  – Minimize code size
• Stack machines abandon registers altogether
  – Stack machines simplify compilers
  – Lend themselves to a compact instruction encoding
  – BUT limit compiler optimization
Evolution of Instruction Sets

Single Accumulator (*EDSAC 1950*)

Accumulator + Index Registers
(*Manchester Mark I, IBM 700 series 1953*)

Separation of Programming Model from Implementation

High-level Language Based
(*B5000 1963*)

Concept of a Family
(*IBM 360 1964*)

General Purpose Register Machines

Complex Instruction Sets
(*Vax, Intel 432 1977-80*)

Load/Store Architecture
(*CDC 6600, Cray 1 1963-76*)

RISC
(*MIPS, SPARC, IBM RS6000, ... 1987*)
# Register-Memory Arch

<table>
<thead>
<tr>
<th># memory addresses</th>
<th>Max. number of operands</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>SPARC, MIPS, PowerPC, ALPHA</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>Intel 60X86, Motorola 68000</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>VAX (also has 3 operands format)</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>VAX (also has 2 operands format)</td>
</tr>
</tbody>
</table>

**Effect of the number of memory operands:**

<table>
<thead>
<tr>
<th>Type</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reg-Reg (0,3)</td>
<td>- Fixed length instruction encoding</td>
<td>- Higher instruction count</td>
</tr>
<tr>
<td></td>
<td>- Simple code generation model</td>
<td>- Some instructions are short leading to wasteful bit encoding</td>
</tr>
<tr>
<td></td>
<td>- Similar execution time (pipeline)</td>
<td></td>
</tr>
<tr>
<td>Reg-Mem (1,2)</td>
<td>- Direct access without loading</td>
<td>- Can restrict # register available for use</td>
</tr>
<tr>
<td></td>
<td>- Easy instruction encoding</td>
<td>- Clocks per instr. varies by operand type</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Source operands are destroyed</td>
</tr>
<tr>
<td>Mem-Mem (3,3)</td>
<td>- No temporary register usage</td>
<td>- Less potential for compiler optimization</td>
</tr>
<tr>
<td></td>
<td>- Compact code</td>
<td>- Can create memory access bottleneck</td>
</tr>
</tbody>
</table>
Memory Addressing

- The address of a word matches the byte address of one of its 4 bytes
- The addresses of sequential words differ by 4 (word size in byte)
- Words' addresses are multiple of 4 (alignment restriction)
  - Misalignment (if allowed) complicates memory access and causes programs to run slower

<table>
<thead>
<tr>
<th>Object addressed</th>
<th>Aligned at byte offsets</th>
<th>Misaligned at byte offsets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte</td>
<td>1,2,3,4,5,6,7</td>
<td>Never</td>
</tr>
<tr>
<td>Half word</td>
<td>0,2,4,6</td>
<td>1,3,5,7</td>
</tr>
<tr>
<td>Word</td>
<td>0,4</td>
<td>1,2,3,5,6,7</td>
</tr>
<tr>
<td>Double word</td>
<td>0</td>
<td>1,2,3,4,5,6,7</td>
</tr>
</tbody>
</table>
Byte Order

- Given N bytes, which is the most significant, which is the least significant?
  - “Big Endian”
    • Leftmost / most significant byte = word address
  - “Little Endian”
    • Rightmost / least significant byte = word address

- Byte ordering can be a problem when exchanging data among different machines

- Can also affect array index calculation or any other operation that treat the same data as both byte and word.
Addressing Modes

• How to specify the location of an operand (effective address)

• Addressing modes have the ability to:
  – Significantly reduce instruction counts
  – Increase the average CPI
  – Increase the complexity of building a machine

• VAX machine is used for benchmark data since it supports wide range of memory addressing modes

• Can classify based on:
  – source of the data (register, immediate or memory)
  – the address calculation (direct, indirect, indexed)
## Example of Addressing Modes

<table>
<thead>
<tr>
<th>Address. mode</th>
<th>Example</th>
<th>Meaning</th>
<th>When used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>ADD R4, #3</td>
<td>Regs[R4] = Regs[R4] + 3</td>
<td>For constants</td>
</tr>
<tr>
<td>Register indirect</td>
<td>ADD R4, (R1)</td>
<td>Regs[R4] = Regs[R4] + Mem[Regs[R1]]</td>
<td>Accessing using a pointer or a computed address</td>
</tr>
<tr>
<td>Direct or absolute</td>
<td>ADD R4, (1001)</td>
<td>Regs[R4] = Regs[R4] + Mem[1001]</td>
<td>Sometimes useful for accessing static data; address constant may need to be large</td>
</tr>
<tr>
<td>Displacement</td>
<td>ADD R4, 100 (R1)</td>
<td>Regs[R4] = Regs[R4] + Mem[100 + Regs[R1]]</td>
<td>Accessing local variables</td>
</tr>
<tr>
<td>Indexed</td>
<td>ADD R4, (R1 + R2)</td>
<td>Regs[R4] = Regs[R4] + Mem[Regs[R1] + Regs[R2]]</td>
<td>Sometimes useful in array addressing: R1 = base of the array; R2 = index amount</td>
</tr>
<tr>
<td>Autoincrement</td>
<td>ADD R4, (R2) +</td>
<td>Regs[R4] = Regs[R4] + Mem[Regs[R2]]</td>
<td>Useful for stepping through arrays within a loop. R2 points to start of the array; each reference increments R2 by d.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Regs[R2] = Regs[R2] + d</td>
<td></td>
</tr>
<tr>
<td>Autodecrement</td>
<td>ADD R4, -(R2)</td>
<td>Regs[R2] = Regs[R2] – d</td>
<td>Same use as autoincrement. Autodecrement/increment can also act as push/pop to implement a stack</td>
</tr>
</tbody>
</table>
Focus on immediate and displacement modes since they are used the most.

Based on SPEC89 on VAX.
Displacement Addressing Modes

- The range of displacement supported affects the length of the instruction.

Data is based on SPEC2000 on Alpha (only 16 bit displacement allowed)
Immediate Addressing Modes

- Immediate values for what operations?

Statistics are based on SPEC2000 benchmark on Alpha
Measurements were taken on Alpha (only 16 bit immediate value allowed). Percentage of Immediate Values:

- Range affects instruction length
  - Similar measurements on the VAX (with 32-bit immediate values) showed that 20-25% of immediate values were longer than 16-bits.

Measurements were taken on Alpha (only 16 bit immediate value allowed).

Distribution of Immediate Values:

- Integer average
- Floating-point average

Number of bits needed for a immediate values in SPEC2000 benchmark.
Addressing Mode for Signal Processing

- DSP offers special addressing modes to better serve popular algorithms
- Special features requires either hand coding or a compiler that uses such features
Addressing Mode for Signal Processing

- **Modulo addressing:**
  - Since DSP deals with continuous data streams, circular buffers common
  - Circular or modulo addressing: automatic increment and decrement / reset pointer at end of buffer

- **Reverse addressing:**
  - Address is the reverse order of the current address
  - Expedites access / otherwise require a number of logical instructions or extra memory accesses

<table>
<thead>
<tr>
<th>Fast Fourier Transform</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (000₂) ➔ 0 (000₂)</td>
</tr>
<tr>
<td>1 (001₂) ➔ 4 (100₂)</td>
</tr>
<tr>
<td>2 (010₂) ➔ 2 (010₂)</td>
</tr>
<tr>
<td>3 (011₂) ➔ 6 (110₂)</td>
</tr>
<tr>
<td>4 (100₂) ➔ 1 (001₂)</td>
</tr>
<tr>
<td>5 (101₂) ➔ 5 (101₂)</td>
</tr>
<tr>
<td>6 (110₂) ➔ 3 (011₂)</td>
</tr>
<tr>
<td>7 (111₂) ➔ 7 (111₂)</td>
</tr>
</tbody>
</table>
Summary of MIPS Addressing Modes

1. Immediate addressing
   \[
   \begin{array}{c|c|c|c|c|c}
   \text{op} & \text{rs} & \text{rt} & \text{Immediate} \\
   \end{array}
   \]

2. Register addressing
   \[
   \begin{array}{c|c|c|c|c|c|c|c}
   \text{op} & \text{rs} & \text{rt} & \text{rd} & \ldots & \text{funct} & \text{Registers} \\
   \end{array}
   \]
   \[
   \begin{array}{c|c|c|c}
   \text{Register} & \text{Memory} \\
   \end{array}
   \]
   \[
   \begin{array}{c|c|c|c|c}
   \text{Byte} & \text{Halfword} & \text{Word} \\
   \end{array}
   \]

3. Base addressing
   \[
   \begin{array}{c|c|c|c|c|c}
   \text{op} & \text{rs} & \text{rt} & \text{Address} \\
   \end{array}
   \]
   \[
   \begin{array}{c|c|c|c}
   \text{Register} & \text{Memory} \\
   \end{array}
   \]
   \[
   \begin{array}{c|c|c|c|c}
   \text{Byte} & \text{Halfword} & \text{Word} \\
   \end{array}
   \]

4. PC-relative addressing
   \[
   \begin{array}{c|c|c|c|c|c}
   \text{op} & \text{rs} & \text{rt} & \text{Address} \\
   \end{array}
   \]
   \[
   \begin{array}{c|c|c|c}
   \text{PC} & \text{Memory} \\
   \end{array}
   \]
   \[
   \begin{array}{c|c|c|c|c}
   \text{Word} \\
   \end{array}
   \]

5. Pseudodirect addressing
   \[
   \begin{array}{c|c|c|c|c|c}
   \text{op} & \text{Address} \\
   \end{array}
   \]
   \[
   \begin{array}{c|c|c|c}
   \text{PC} & \text{Memory} \\
   \end{array}
   \]
   \[
   \begin{array}{c|c|c|c|c}
   \text{Word} \\
   \end{array}
   \]

\[+\text{Concatenation}\]
**Operations of the Computer Hardware**

“There must certainly be instructions for performing the fundamental arithmetic operations.”

Burkes, Goldstine and Von Neumann, 1947

MIPS assembler allows only one instruction/line and ignore comments following # until end of line

**Example:**

Translation of a segment of a C program to MIPS assembly instructions:

C: \[ f = (g + h) - (i + j) \]

(pseudo)MIPS:

- add \( t0, g, h \) \# temp. variable \( t0 \) contains "g + h"
- add \( t1, i, j \) \# temp. variable \( t1 \) contains "i + j"
- sub \( f, t0, t1 \) \# \( f = t0 - t1 = (g + h) - (i + j) \)
### Operations in the Instruction Set

<table>
<thead>
<tr>
<th>Operator type</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic and logical</td>
<td>Integer arithmetic and logical operations: add, and, subtract, or</td>
</tr>
<tr>
<td>Data Transfer</td>
<td>Loads-stores (move instructions on machines with memory addressing)</td>
</tr>
<tr>
<td>Control</td>
<td>Branch, jump, procedure call and return, trap</td>
</tr>
<tr>
<td>System</td>
<td>Operating system call, Virtual memory management instructions</td>
</tr>
<tr>
<td>Floating point</td>
<td>Floating point instructions: add, multiply</td>
</tr>
<tr>
<td>Decimal</td>
<td>Decimal add, decimal multiply, decimal to character conversion</td>
</tr>
<tr>
<td>String</td>
<td>String move, string compare, string search</td>
</tr>
<tr>
<td>Graphics</td>
<td>Pixel operations, compression/decompression operations</td>
</tr>
</tbody>
</table>

- Arithmetic, logical, data transfer and control are almost standard categories for all machines.
- System instructions are required for multi-programming environment although support for system functions varies.
- Others can be primitives (e.g. decimal and string on IBM 360 and VAX), provided by a co-processor, or synthesized by compiler.

- **Partitioned Add:**
  - Partition a single register into multiple data elements (e.g. 4 16-bit words in 1 64-bit register)
  - Perform the same operation independently on each
  - Increases ALU throughput for multimedia applications

- **Paired single operations**
  - Perform multiple independent narrow operations on one wide ALU (e.g. 2 32-bit float ops)
  - Handy in dealing with vertices and coordinates

- **Multiply and accumulate**
  - Very handy for calculating dot products of vectors (signal processing) and matrix multiplication
Frequency of Operations

Usage

- The most widely executed instructions are the simple operations of an instruction set
- Average usage in SPECint92 on Intel 80x86:

<table>
<thead>
<tr>
<th>Rank</th>
<th>80x86 Instruction</th>
<th>Integer Average (% total executed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Load</td>
<td>22%</td>
</tr>
<tr>
<td>2</td>
<td>Conditional branch</td>
<td>20%</td>
</tr>
<tr>
<td>3</td>
<td>Compare</td>
<td>16%</td>
</tr>
<tr>
<td>4</td>
<td>Store</td>
<td>12%</td>
</tr>
<tr>
<td>5</td>
<td>Add</td>
<td>8%</td>
</tr>
<tr>
<td>6</td>
<td>And</td>
<td>6%</td>
</tr>
<tr>
<td>7</td>
<td>Sub</td>
<td>5%</td>
</tr>
<tr>
<td>8</td>
<td>Move register-register</td>
<td>4%</td>
</tr>
<tr>
<td>9</td>
<td>Call</td>
<td>1%</td>
</tr>
<tr>
<td>10</td>
<td>Return</td>
<td>1%</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>96%</td>
</tr>
</tbody>
</table>

Make the common case fast by focusing on these operations
Control Flow Instructions

- Jump: unconditional change in the control flow
- Branch: conditional change in the control flow
- Procedure calls and returns

Data is based on SPEC2000 on Alpha
• PC-relative addressing
  – Good for short position-independent forward & backward jumps
• Register indirect addressing
  – Good for dynamic libraries, virtual functions & packed case statements

Data is based SPEC2000 on Alpha
## Condition Evaluation

<table>
<thead>
<tr>
<th>Name</th>
<th>How condition is tested</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Condition Code (CC)</td>
<td>Special bits are set by ALU operations, possibly under program control</td>
<td>Sometimes condition is set for free</td>
<td>CC is extra state. Condition codes constrain instructions’ ordering since they pass info. from one instruction to a branch</td>
</tr>
<tr>
<td>Condition register</td>
<td>Test arbitrary register with the result of a comparison</td>
<td>Simple</td>
<td>Uses up a register</td>
</tr>
<tr>
<td>Compare &amp; branch</td>
<td>Compare is part of the branch.</td>
<td>One instruction rather than two for a branch</td>
<td>May be too much work per instruction</td>
</tr>
</tbody>
</table>

### Based on SPEC92 on MIPS

<table>
<thead>
<tr>
<th>Comparison Type</th>
<th>Integer Average</th>
<th>Floating-point Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than/ greater than or equal</td>
<td>7%</td>
<td>40%</td>
</tr>
<tr>
<td>Greater than/ less than or equal</td>
<td>7%</td>
<td>23%</td>
</tr>
<tr>
<td>Equal/ not equal</td>
<td></td>
<td>37%</td>
</tr>
</tbody>
</table>

![Graph showing frequency of comparison types in branches.](image)

Remember to focus on the common case
Data is based on SPEC2000 on Alpha

Different benchmark and machine set new design priority

DSPs support repeat instruction for for loops (vectors) using 3 registers
Type and Size of Operands

• Operand type encoded in instruction opcode
  – The type of an operand effectively gives its size
• Common types include character, half word and word size integer, single- and double-precision floating point
  – Characters are almost always in ASCII, though 16-bit Unicode (for international characters) is gaining popularity
  – Integers in 2’s complement
  – Floating point in IEEE 754
Unusual Types

- **Business Applications**
  - Binary Coded Decimal (BCD)
    - Exactly represents all decimal fractions (binary doesn’t!)
- **DSP**
  - Fixed point
    - Good for limited range numbers: more mantissa bits
  - Block floating point
    - Single shared exponent for multiple numbers
- **Graphics**
  - 4-element vector operations (RGBA or XYZW)
    - 8-bit, 16-bit or single-precision floating point
Size of Operands

- Double-word: double-precision floating point + addresses in 64-bit machines
- Words: most integer operations + addresses in 32-bit machines
- For the mix in SPEC, word and double-word data types dominates
Instruction Representation

- All data in computer systems is represented in binary
- Instructions are no exception
- The program that translates the human-readable code to numeric form is called an Assembler
- Hence *machine-language* or *assembly-language*

Example:

**Assembly:**  
ADD $t0, $s1, $s2

**M/C language (binary):**  
000000 00001 00010 00000 00000 100000 0000 0000 0010 0010 0000 0000 0010 0000

**M/C language (hex):**  
0x00220020

*Note:* MIPS compiler by default maps $s0,…,$s7 to reg. 16-23 and $t0,…,$t7 to reg. 8-15
Encoding an Instruction Set

- Affects the size of the compiled program
- Also complexity of the CPU implementation
- Operation in one field called opcode
- Addressing mode in opcode or separate field
- Must balance:
  - Desire to support as many registers and addressing modes as possible
  - Effect of operand specification on the size of the instruction (and program)
  - Desire to simplify instruction fetching and decoding during execution
- Fixed size instruction encoding simplifies CPU design but limits addressing choices
### 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)
MIPS Instruction Formats

I-type instruction

<table>
<thead>
<tr>
<th>Opcode</th>
<th>6</th>
<th>5</th>
<th>5</th>
<th>16</th>
</tr>
</thead>
</table>

Encodes: Loads and stores of bytes, half words, words, double words. All immediates (rt ← rs op immediate)

Conditional branch instructions (rs is register, rd unused)
Jump register, jump and link register
(rd = 0, rs = destination, immediate = 0)

R-type instruction

<table>
<thead>
<tr>
<th>Opcode</th>
<th>6</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
</table>

Register-register ALU operations: rd ← rs funct rt
Function encodes the data path operation: Add, Sub, ...
Read/write special registers and moves

J-type instruction

<table>
<thead>
<tr>
<th>Opcode</th>
<th>6</th>
<th>26</th>
</tr>
</thead>
</table>

Jump and jump and link
Trap and return from exception

opcodes

<table>
<thead>
<tr>
<th></th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>j</td>
<td>jal</td>
<td>beq</td>
<td>bne</td>
<td>blez</td>
<td>bgtz</td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi</td>
<td>addiu</td>
<td>slti</td>
<td>sltiu</td>
<td>andi</td>
<td>ori</td>
<td>xori</td>
<td></td>
<td></td>
</tr>
<tr>
<td>llo</td>
<td>lhi</td>
<td>trap</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw</td>
<td>lbu</td>
<td>lhu</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sb</td>
<td>sh</td>
<td>sw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

func codes

<table>
<thead>
<tr>
<th></th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>sll</td>
<td></td>
<td>srl</td>
<td>sra</td>
<td>sllv</td>
<td>srlv</td>
<td>srav</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jr</td>
<td></td>
<td>jaf</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mfh</td>
<td>mthi</td>
<td>mfhi</td>
<td>mfl</td>
<td>mflo</td>
<td>mfl</td>
<td>mfl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mult</td>
<td>multu</td>
<td></td>
<td>div</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>addu</td>
<td>sub</td>
<td>subu</td>
<td>and</td>
<td>or</td>
<td>xor</td>
<td>nor</td>
<td></td>
</tr>
<tr>
<td>slt</td>
<td>sltu</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Stored Program Concept

- Today’s computers are build on two key principles:
  - Instructions are represented as numbers
  - Programs can be stored in memory to be read or written just like numbers

- Memory can contain:
  - the source code for an editor
  - the compiled m/c code for the editor
  - the text that the compiled program is using
  - the compiler that generated the code
Conclusion

• Summary
  – Type and size of operands
    • (common data types, effect of operand size on complexity)
  – Encoding the instruction set
    • (Fixed, variable and hybrid encoding, stored program)

• Next Week
  – Role and effect of compilers on ISA
  – Pipelined execution of instructions
  – Pipeline hazards