Instruction Categories

- Arithmetic
  - \( x = x + 1 \)
- Memory
  - \( \text{mem[addr]} = x; \)
- Control
  - \( \text{for(int i = 0; i < 10 ; i++)} \)
Arguments to Arithmetic Operations

- **Constant**: $x = x + 1$
  - Immediate addressing
    - Data appears in constant part of instruction
- **Data stored in register**
  - Register direct or register addressing
    - Data is in register specified in instruction
- **Data stored in memory**: $\text{mem[addr]} = x$
  - Direct addressing
    - Explicitly state address in instruction
  - Register indirect
    - Register holds address of data in memory
Memory Aside: Memory Hierarchy

• Registers
  ✷ Small, fast to access, right next to circuitry that performs calculation
  ✷ Few in number
  ✷ Expensive!
    ▪ Both in circuitry and real-estate!

• Memory (logical and physical)
  ✷ Larger, slower, further away than registers
Memory Aside: Data

- Each program has memory associated with it where data is stored
- That data is just like an array of memory cells
  - Smallest addressable component is a byte
  - Each cell is addressable by its byte location

Memory

0x0
0x7ff3fa8bc
0x7fffffff

argc is 4B
Memory Aside: Data

- Each program has memory associated with it where data is stored
- That data is just like an array of memory cells
  - Smallest addressable component is a byte
  - Each cell is addressable by its byte location

```
0x5ffffd420
8 char string
```

```
0x7fffffffff
Memory
```

```
0x0
```
Each program has memory associated with it where data is stored
That data is just like an array of memory cells
  - Smallest addressable component is a byte
  - Each cell is addressable by its byte location
Instructions are also allocated space in memory

Memory:
0x0
0xffffffff
0xf5fd780
0x0
Stored Program Concept

• Very important: effectively allows the development of general purpose computer
• Treat instructions the same way as data
• Simplifies memory hardware and the software of computers
• Ex. Memory can contain source code for editing app, corresponding machine code, text program is using, and compiler that generated machine code!
Instruction Set Architecture (ISA)

- Specifies the instructions that the architecture understands
- Represents the interface between HW and SW
- Classified by number of addresses included in instructions
- Differentiated by how registers / memory used to specify data
Different Addressing Architectures

- 0 address architecture: Stack
  - Push operands onto stack
  - Operations automatically take operands from top of stack (pop)
  - Push result on to stack

4 + 5

- Stack
  - push 4
  - push 5
  - add
  - Result: 9
Different Addressing Architectures

- Accumulator
  - Special register where results stored
  - \( \text{accum} = \text{accum} \text{ op addr}_1 \ldots \text{addr}_n \)
  - Advantage: don’t have to specify as many addresses in instructions

- 1 address architecture
  - Assume one operand is in accumulator
  - Other address is specified
  - \( \text{accum} = \text{accum} \text{ op addr}_1 \)
Different Addressing Architectures

- 3 address architecture: MIPS
  - 2 inputs to operation
  - 1 output from operation
  - `add addr_1, addr_2, addr_3`
    
    ```
    // addr_1 = addr_2 + addr_3
    ```

  Nothing is implicit!
Addresses are Registers or Memory?

- **Accumulator Architecture**
  - 1 register for arithmetic (the accumulator)
  - 1 operand for memory
    - `add addr` // acc = acc + mem[addr]
  - Accumulator both source and destination

- **Dedicated Register Architecture**
  - More registers
  - Each register with dedicated purpose
    - Stack pointer
    - Array index
  - Has both accumulator–like instructions and MIPS–like instructions
  - Multiply/div accumulator separate from other accumulators

- **General Purpose Register Architecture**
  - All registers used for any purpose
  - Two Kinds:
    - Register–memory: one operand in memory
    - Load–store: all operands in registers
From Source Code to Machine Language

High Level Language Program

Compiler

Assembly Language Program

Assembler

Machine Language Program

Machine Interpretation

Hardware

temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;

lw $15, 0($2)
lw $16, 4($2)
sw $16, 0($2)
sw $15, 4($2)

10001100011000100000000000000000
1000110011110010000000000000100
10101100111100100000000000000000
1010110001100010000000000000100

ALUOP[0:3] <= InstReg[9:11] & MASK
Classifying Instruction Types

- **CISC: Complex Instruction Set Computer**
  - Create instructions for specific operations
    - Square root
  - Create composite instructions
    - Load then add then store
  - Allow both memory and register addressing
  - Allow multiple operands in each instruction
  - Variable instruction length
  - Examples: VAX, Motorola 68K, Intel x86

- **RISC: Reduced Instruction Set Computer**
  - Include small number of basic instructions
    - Load, store, add, subtract
  - Small number of operands permitted to instructions
  - Fixed instruction length
  - Examples: MIPS, PowerPC, SPARC, HP PA-RISC
Classifying Instruction Types

• CISC:
  - Make hardware like software to make compilation easier
  - Also optimize program size

• RISC:
  - VERY much easier to decode instructions
    - Simple instructions have implications for program performance
  - Benefit of RISC is really when it comes to hardware
2.1 INSTRUCTION FORMAT FOR PROTECTED MODE, REAL-ADDRESS MODE, AND VIRTUAL-8086 MODE

IA-32 instruction encodings are subsets of the format shown in Figure 2-1. Instructions consist of optional instruction prefixes (in any order), primary opcode bytes (up to three bytes), an addressing-form specifier (if required) consisting of the ModR/M byte and sometimes the SIB (Scale-Index-Base) byte, a displacement (if required), and an immediate data field (if required).

Pentium 4: Note instruction length is variable!
### SQRTSS—Compute Square Root of Scalar Single-Precision Floating-Point Value

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 51 l/r</td>
<td>SQRTSS xmm1, xmm2/m32</td>
<td>Valid</td>
<td>Valid</td>
<td>Computes square root of the low single-precision floating-point value in xmm2/m32 and stores the results in xmm1.</td>
</tr>
</tbody>
</table>

#### Description

Computes the square root of the low single-precision floating-point value in the source operand (second operand) and stores the single-precision floating-point result in the destination operand. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. The three high-order doublewords of the destination operand remain unchanged. See Figure 10-6 in the IA-32 Intel Architecture Software Developer’s Manual, Volume I for an illustration of a scalar single-precision floating-point operation.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

#### Operation

DEST[31:0] ← SQRT (SRC[31:0]);
(* DEST[127:64] unchanged *)
RISC Example

I-type: used by immediate and data transfer functions

R-type: used for register functions
## ADD

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SPECIAL</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>ADD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>000000</td>
<td></td>
<td></td>
<td></td>
<td>00000</td>
<td>100000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Format:
ADD rd, rs, rt

### Purpose:
To add 32-bit integers. If overflow occurs, then trap.

### Description:
rd ← rs + rt

The 32-bit word value in GPR rt is added to the 32-bit value in GPR rs to produce a 32-bit result. If the addition results in 32-bit 2's complement arithmetic overflow then the destination register is not modified and an Integer Overflow exception occurs. If it does not overflow, the 32-bit result is placed into GPR rd.
Storing Numbers
Unsigned Numbers

- In general, value of $i^{th}$ digit $d$ is $d \times \text{Base}^i$
  - $1011_2 = 11_{10}$

- 32b allows you to represent $2^{32}$ different 32b patterns
  - 0 to $2^{32}-1$ (4,294,967,295)
Sign and Magnitude

- Single bit used to represent positive or negative
- Problems:
  - Which bit for sign? First? Last?
  - 2 representations for zero
  - Adders may need additional cycle to figure out sign
Signed Numbers: Two’s Complement

- Used in today’s computers
- Use leading bit to represent positive or negative
  - 0: positive
  - 1: negative
- Positive numbers have same representation but have one less bit to use
  - 32b: 0 to $2^{31}-1$ (2,147,483,647)
- Negative numbers
  - 32b: $-1$ to $-2^{31}$ (-2,147,483,648)
    - $-1$: 32 1s
    - $-2^{31}$: 1 (31 0s)
Signed Numbers: Two’s Complement

- Converting from two’s complement to decimal
  \[(x_{31} \times -2^{31}) + (x_{30} \times 2^{30}) + \ldots + (x_1 \times 2^1) + (x_0 \times 2^0)\]
  where \(x_i\) means the \(i^{th}\) bit of \(x\)

- Convert between decimal and two’s complement
  - 1(27 1s)1100
Converting from Positive to Negative

• Value 6 is represented in 8b as 0 0 0 0 0 1 1 0
• Value –6 is represented in 8b as 1 1 1 1 1 0 1 0
• Technique:
  ♦ Invert every bit
  ♦ Add one
• Works to convert negative to positive too!
Examples

• What is the 2’s complement representation of (assume 8b)
  ◦ –3
  ◦ –62

• What is the decimal value for these 2’s complement numbers?
  ◦ 10010011
  ◦ 11110100
Nifty Benefits of Two’s Complement

- Addition of negative number works correctly
  - Add 7 to -5
    \[
    \begin{array}{c}
    0111 \\
    + 1011 \\
    \hline
    0010
    \end{array}
    \]
  - Add 2 to -5
    \[
    \begin{array}{c}
    0010 \\
    + 1011 \\
    \hline
    1101
    \end{array}
    \]
Overflow

When can adding/subtracting two numbers result in overflow?

- **Addition**: (carry into sign bit)
  - Positive + Positive = Negative
    - 4b values: 0111 + 0111 = 1110
  - Negative + Negative = Positive
    - 4b values: 1001 + 1001 = 0010

- **Subtraction**: (borrow from sign bit)
  - Positive – Negative = Negative
    - 4b values: 0100 – 1001 = ??
  - Negative – Positive = Positive
    - 4b values: 1001 – 0110 = 0011
    - (This is -7–6, which should be -13, i.e., 10011)
Overflow w/ Unsigned Numbers

- Unsigned integers generally used for memory addresses
- Want to ignore overflow
- Two types of add/sub instructions
  - add/addi/sub – overflow causes exception
  - addu/addiu/subu – overflow does not cause exception
Comparisons

• Need to distinguish between signed and unsigned numbers for comparison
• slt / slti – assumes numbers are signed
• sltu / sltiu – assumes numbers are unsigned
Sign Extension

- Converting from smaller number of bits to larger number of bits
  - Immediate fields are 16b but registers are 32b
- Convert 16b to 32b
  - Copy most significant bit 16 times and prepend
  - 0000 0000 0000 0010
    - 0000 0000 0000 0000 0000 0000 0000 0010
    - Positive #s have infinite number of preceding 0s
  - 1111 1111 1111 1110
    - 1111 1111 1111 1111 1111 1111 1111 1110
    - Negative #s have infinite number of preceding 1s