Lecture 3

Instruction Set Architecture

Outline

- Instruction set architecture
  - RISC vs. CISC
  - MIPS/ARM/x86
- Instructions:
  - Arithmetic instruction: add, sub, …
  - Data transfer instruction: lw, sw, lh, sh, …
  - Logical instruction: and, or, …
  - Conditional branch beq, bne, …
- Basic concepts:
  - Operands: register vs. memory vs. immediate
  - Numeric representation: signed, unsigned, sign extension
  - Instruction format: R-format vs. I-format

Instruction Set

- To command a computer's hardware, you must speak its language
  - ✓ Instructions: words of a computer's language
  - ✓ Instruction set: vocabulary of commands

- Two forms of instruction set:
  - ✓ Assembly language: written by people
  - ✓ Machine language: read by computer

- A program (in say, C) is compiled into an executable program that is composed of machine instructions
  - ✓ This executable program must also run on future machines
  - ✓ each Intel processor reads in the same x86 instructions, but each processor handles instructions differently

- Java programs are converted into portable bytecode that is converted into machine instructions during execution (just-in-time compilation)

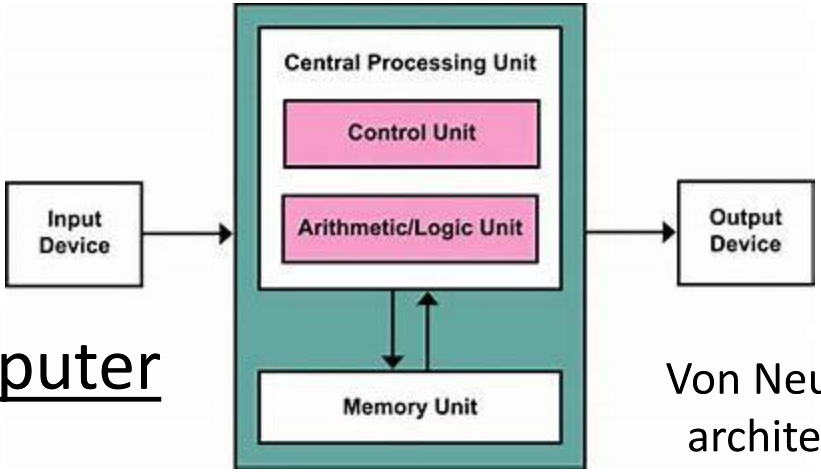
Instruction Set

- Instruction set of different machine are similar, because
  - ✓ They base on similar design principles
  - ✓ Several basic operations are provided
  - ✓ computer designers have a common goal
- Design target:
  - ✓ easy to build the hardware and compiler
  - ✓ maximizing performance and minimizing cost and energy
- Important design principles:
  - ✓ keep the hardware simple – the chip must only implement basic primitives and run fast
  - ✓ keep the instructions regular – simplifies the decoding/ scheduling of instructions

Von Neumann

It is easy to see by formal-logical methods that there exist certain [instruction sets] that are in abstract adequate to control and cause the execution of any sequence of operations . . . . The really decisive considerations from the present point of view, in selecting an [instruction set], are more of a practical nature: simplicity of the equipment demanded by the [instruction set], and the clarity of its application to the actually important problems together with the speed of its handling of those problems.

Burks, Goldstine, and von Neumann, 1947



Von Neumann architecture

Stored-program computer

Instruction Set

- All instruction set are similar
  - ✓ Once learn one, easy to pick up others
- We will use MIPS as an example
  - ✓ MIPS: Microprocessor without interlocked pipeline stages
  - ✓ History of MIPS
- We will later discuss RISC vs CISC
  - ✓ RISC: reduced instruction set computer, e.g. MIPS, ARM, PowerPC, RISC-V
  - ✓ CISC: complex instruction set computer, e.g. x86

John Cocke, IBM
The father of RISC

D. Patterson, UC Berkeley

A Basic MIPS Instruction

C code: a = b + c ;

Assembly code: (human-friendly machine instructions)

add a, b, c # a is the sum of b and c

Machine code: (hardware-friendly machine instructions)

00000010001100100100000000100000

Translate the following C code into assembly code:

a = b + c + d + e;

Example

C code   a = b + c + d + e;

translates into the following assembly code:

add a, b, c		add a, b, c
add a, a, d	or	add f, d, e
add a, a, e		add a, a, f

- Instructions are simple: fixed number of operands (unlike C)
- A single line of C code is converted into multiple lines of assembly code
- Some sequences are better than others… the second sequence needs one more (temporary) variable f

Subtract Example

C code f = (g + h) – (i + j);

Assembly code translation with only add and sub instructions:

Subtract Example

C code f = (g + h) – (i + j);
translates into the following assembly code:

```
add  t0, g, h            add  f, g, h
add  t1,  i, j      or   sub   f, f, i
sub  f,   t0, t1         sub   f, f, j
```

- Each version may produce a different result because floating-point operations are not necessarily associative and commutative… more on this later

Design Principle 1

- Simplicity favors regularity
  - ✓ Regularity makes implementation simpler
  - ✓ Simplicity enables higher performance at lower cost

Operands

- In C, each “variable” is a location in memory
- In hardware, each memory access is expensive – if variable *a* is accessed repeatedly, it helps to bring the variable into an on-chip scratchpad and operate on the scratchpad (registers)
- To simplify the instructions, we require that each instruction (add, sub) only operate on registers
- Note: the number of operands (variables) in a C program is very large; the number of operands in assembly is fixed… there can be only so limited number of registers

Registers

- The MIPS ISA has 32 registers (x86 has 8 registers) – Why not more? Why not less?

- Each register is 32-bit wide (modern 64-bit architectures have 64-bit wide registers)

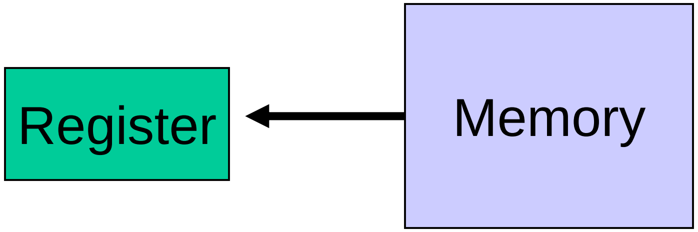
- A 32-bit entity (4 bytes) is referred to as a word

- To make the code more readable, registers are partitioned as $s0-$s7 (C/Java variables), $t0-$t9 (temporary variables)…

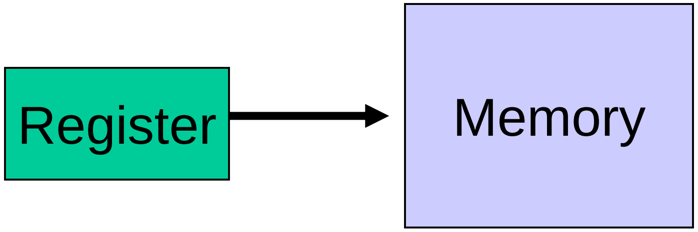
Memory Operands

- Values must be fetched from memory before (add and sub) instructions can operate on them

Load word
lw $t0, memory-address



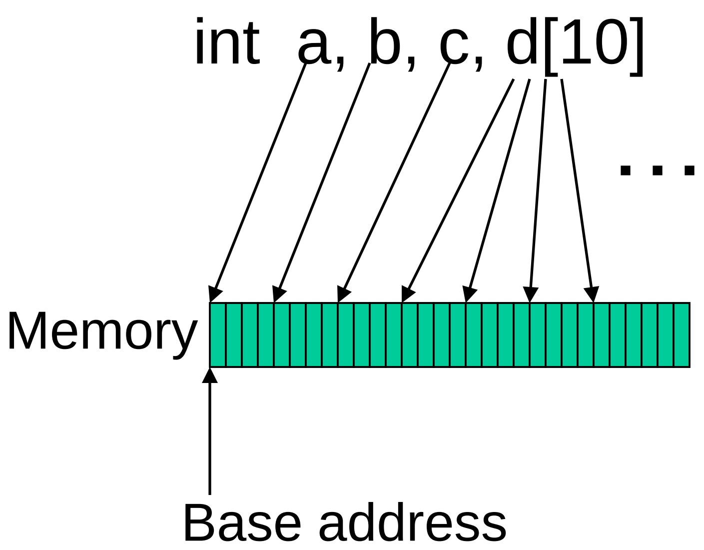
Store word
sw $t0, memory-address

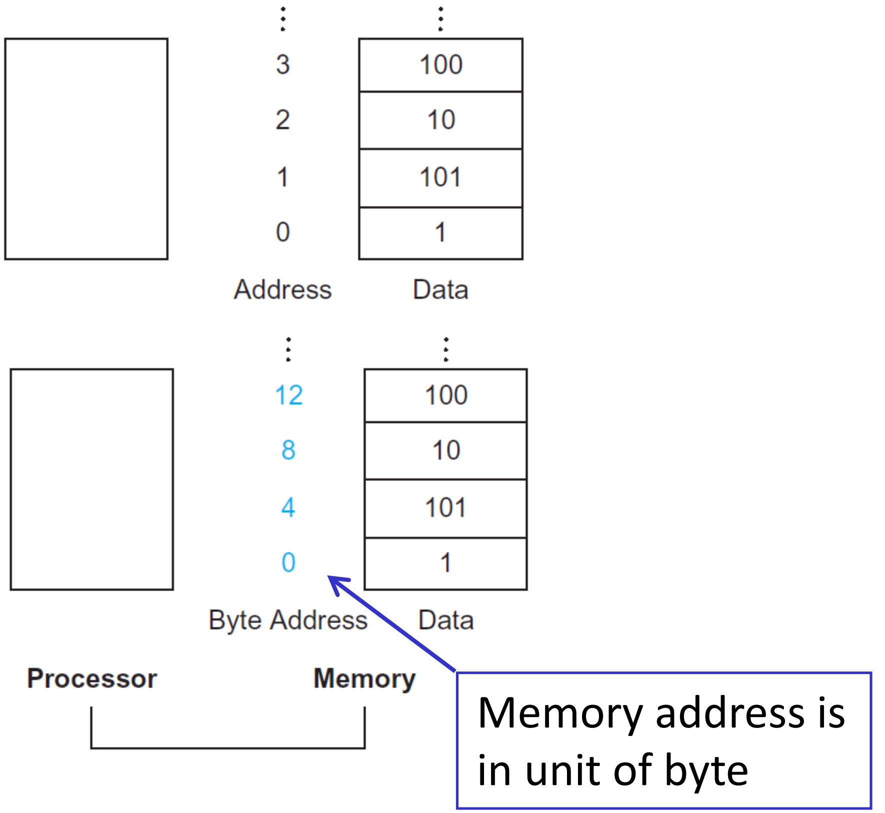


How is memory-address determined?

Memory Address

- The compiler organizes data in memory… it knows the location of every variable (saved in a table)… it can fill in the appropriate mem-address for load-store instructions





Immediate Operands

- An instruction may require a constant as input
- An immediate instruction uses a constant number as one of the inputs (instead of a register operand)

```
addi  $s0, $zero, 1000   # the program has base address
                         #  1000 and this is saved in $s0
                         # $zero is a register that always
                         # equals zero
addi  $s1, $s0, 0      # this is the address of variable a
addi  $s2, $s0, 4      # this is the address of variable b
addi  $s3, $s0, 8      # this is the address of variable c
addi  $s4, $s0, 12     # this is the address of variable d[0]
```

Memory Instruction Format

- The format of a load instruction:

destination register

source address

```
lw    $t0,   8($s0)
```

any register

a constant that is added to the register in brackets

Example

Convert to assembly:

C code: d[3] = d[2] + a;

Example

Convert to assembly:

C code: d[3] = d[2] + a;

Assembly: # addi instructions as before

```
lw    $t0, 8($s4)     #  d[2] is brought into $t0
lw    $t1, 0($s1)     #   a  is brought into $t1
add   $t0, $t0, $t1   #  the sum is in $t0
sw    $t0, 12($s4)    #  $t0 is stored into d[3]
```

Assembly version of the code continues to expand!

Registers vs. Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
  - More instructions to be executed
- Compiler must use registers for variables as much as possible
  - Only spill to memory for less frequently used variables
  - Register optimization is important!

Design Principles

- *Design Principle 2*: Smaller is faster
  - Register vs. memory
  - Number of registers is small

- *Design Principle 3:* Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction

Numeric Representations

- Assume that the bits in register $s0 are as follows, what is the value of s0?

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1

- How about this one?

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1

Numeric Representations

- Decimal 35_{10}
- Binary 00100011_2
- Hexadecimal (compact representation)

  0x 23 or 23_{hex}

  0-15 (decimal) → 0-9, a-f (hex)

Unsigned Binary Integers

- Given an n-bit number

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: 0 to $+2^n-1$
- Example
  - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0011_2$
    $= 0 + \ldots + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$
    $= 0 + \ldots + 0 + 0 + 2 + 1 = 3_{10}$
- Using 32 bits
  - 0 to +4,294,967,295

2s-Complement Signed Integers

- Given an n-bit number, define the value as follows:

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: -2^{n-1} to $+2^{n-1} - 1$
- Example
  - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
    $= -1 \times 2^{31} + 1 \times 2^{30} + \ldots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$
    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$
- Using 32 bits
  - $-2{,}147{,}483{,}648$ to $+2{,}147{,}483{,}647$

Signed Negation

- 2's complement = 1's complement + 1
  - 1's complement means 1 → 0, 0 → 1
  - Using $\overline{x}$ represent 1's complement

$$x + \overline{x} = 1111\ldots111_2 = -1$$
$$\overline{x} + 1 = -x$$

- Example: -2
  - $+2 = 0000\ 0000\ \ldots\ 0010_2$
  - $-2 = 1111\ 1111\ \ldots\ 1101_2 + 1$
    $= 1111\ 1111\ \ldots\ 1110_2$

Sign Extension

- Representing a number using more bits
  - Preserve the numeric value
  - E.g. we copy 8-bit to register and then want to extend it to be 16-bit or 32-bit
- In MIPS instruction set
  - `addi`: extend immediate value
  - `lb`, `lh`: extend loaded byte/halfword
  - `beq`, `bne`: extend the displacement
- Replicate the sign bit to the left
  - c.f. unsigned values: extend with 0s
- Examples: 8-bit to 16-bit
  - +2: 0000 0010 => 0000 0000 0000 0010
  - –2: 1111 1110 => 1111 1111 1111 1110

Instruction Formats

Instructions are represented as 32-bit numbers (one word), broken into 6 fields

R-type instruction add $t0, $s1, $s2

000000	10001	10010	01000	00000	100000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
op	rs	rt	rd	shamt	funct
opcode	source	source	dest	shift amt	function

I-type instruction lw $t0, 32($s3)

6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	constant

Design Principle 4

- *Design Principle 4:* Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

Logical Operations

<table>
<tr><th>Logical ops</th><th>C operators</th><th>Java operators</th><th>MIPS instr</th></tr>
<tr><td>Shift Left</td><td><<</td><td><<</td><td>sll</td></tr>
<tr><td>Shift Right</td><td>>></td><td>>>></td><td>srl</td></tr>
<tr><td>Bit-by-bit AND</td><td>&</td><td>&</td><td>and, andi</td></tr>
<tr><td>Bit-by-bit OR</td><td>|</td><td>|</td><td>or, ori</td></tr>
<tr><td>Bit-by-bit NOT</td><td>~</td><td>~</td><td>nor</td></tr>
</table>

Control Instructions

- Conditional branch: Jump to instruction L1 if register1 equals register2: beq register1, register2, L1
  Similarly, bne and slt (set-on-less-than)

- Unconditional branch:
  j L1
  jr $s0

Convert to assembly:

```
if  (i == j)
    f = g+h;
else
    f = g-h;
```

Control Instructions

- Conditional branch: Jump to instruction L1 if register1 equals register2: beq register1, register2, L1
  Similarly, bne and slt (set-on-less-than)

- Unconditional branch:
  j L1
  jr $s0

Convert to assembly:

```
if  (i == j)                     bne   $s3, $s4, Else
    f = g+h;                     add   $s0, $s1, $s2
else                             j     Exit
    f = g-h;              Else:  sub   $s0, $s1, $s2
                          Exit:
```

Example

Convert to assembly:

```
while   (save[i] == k)
    i += 1;
```

i and k are in $s3 and $s5 and
base of array save[] is in $s6

Example

Convert to assembly:

```
while   (save[i] == k)
     i += 1;
```

i and k are in $s3 and $s5 and base of array save[] is in $s6

```
Loop:  sll    $t1, $s3, 2
       add    $t1, $t1, $s6
       lw     $t0, 0($t1)
       bne    $t0, $s5, Exit
       addi   $s3, $s3, 1
       j      Loop
Exit:
```

Homework #2

- Chapter 2： 2.1, 2.3, 2.6, 2.12, 2.16
- Due on Mar. 12