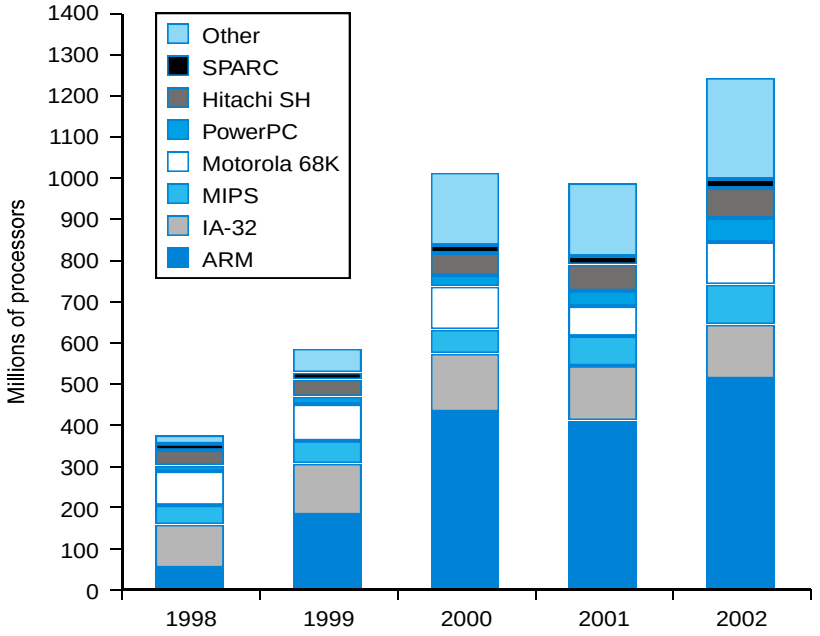
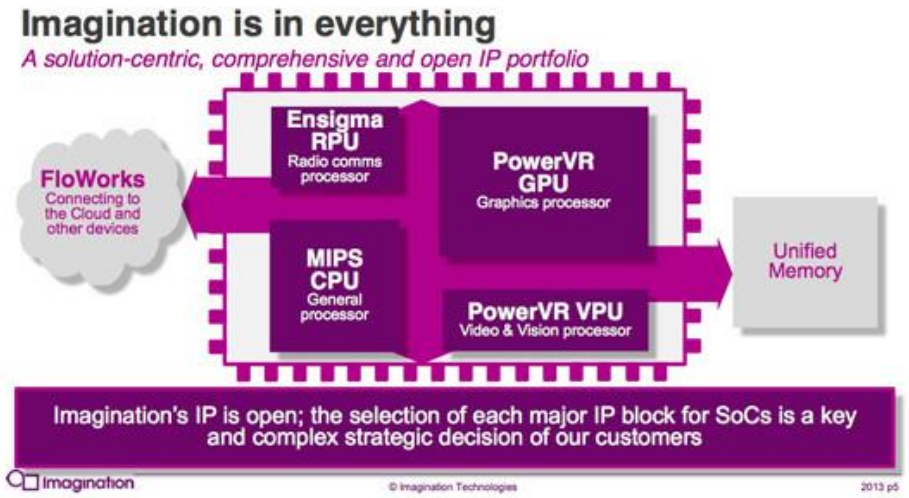
Computer Architecture
Chapter 2-1

Instructions: Language of the Computer

Instructions:

- **Language of the Machine**
  - Different computers have different instruction sets
  - But with many aspects in common
- **We'll be working with the MIPS instruction set architecture**
  - Stanford MIPS commercialized by MIPS Technologies (www.mips.com)
  - Large share of embedded core market: Applications in consumer electronics, network/storage equipment, cameras, printers, …
  - MIPS Technologies, Inc. was acquired by Imagination Technologies (www.imgtec.com), February 2013





MIPS-based SoC

MIPS

- **Learn one specific ISA: MIPS ISA.**
  - Assembly languages are all similar.
  - You will write some functions in MIPS assembly instructions and test your code using a MIPS software simulator, SPIM.
- **Familiarize MIPS instructions & instruction formats for later chapters.**
  - We will be using MIPS throughout the rest of the textbook.
- **See MIPS Reference Data tear-out card, and Appendixes A and E**

MIPS arithmetic

- **All instructions have 3 operands**
- **Operand order is fixed (destination first)**

Example:

C code: `a = b + c`

MIPS 'code': `add $s0, $s1, $s2`
(associated with variables by compiler)

"The natural number of operands for an operation like addition is three…requiring every instruction to have exactly three operands, no more and no less, conforms to the philosophy of keeping the hardware simple"

MIPS arithmetic

- **Design Principle 1: simplicity favors regularity.**
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

C code:

```
a = b + c + d;
e = f - a;
```

MIPS code:

```
add $t0, $s1, $s2
add $s0, $t0, $s3
sub $s4, $s5, $s0
```

- **Operands must be registers, only 32 registers provided (0~31)**
- **Each register contains 32 bits**

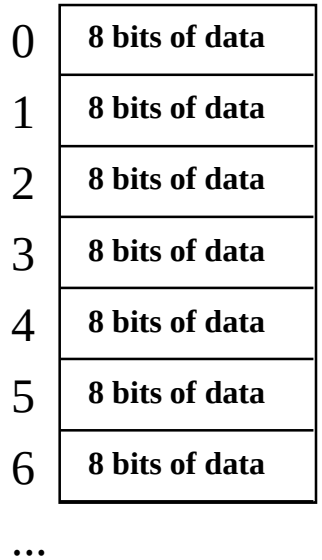
- **Design Principle 2: smaller is faster.** **Why?**
  - A very large number of registers may increase the clock cycle time simply because it takes electronic signals longer when they must travel farther.

MIPS Registers: Fast Locations for Data

- **32x32-bit registers:** `$0, $1, … , $31`
  - operands for integer arithmetic
  - address calculations
  - temporary locations
  - special-purpose functions defined by convention
  - Assembler names
    - $t0, $t1, …, $t9 for temporary values
    - $s0, $s1, …, $s7 for saved variables
- **1x32-bit Program Counter (`PC`)**
- **2x32-bit registers `HI & LO`:**
  - used for multiply & divide
- **32x32-bit registers:** `$f0, … $f31`
  - floating-point arithmetic (often used as 16 64-bit registers)

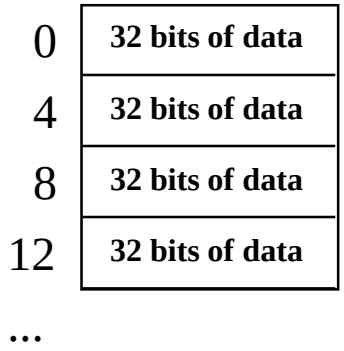
Memory Organization

- **Viewed as a large, single-dimension array, with an address.**
- **used for composite data**
  - Arrays, structures, dynamic data
- **To apply arithmetic operations**
  - Load values from memory into registers
  - Store result from register to memory
- **A memory address is an index into the array**
- **"Byte addressing" means that the index points to a byte of memory.**



Memory Organization

- **Bytes are nice, but most data items use larger "words"**
- **For MIPS, a word is 32 bits or 4 bytes.**



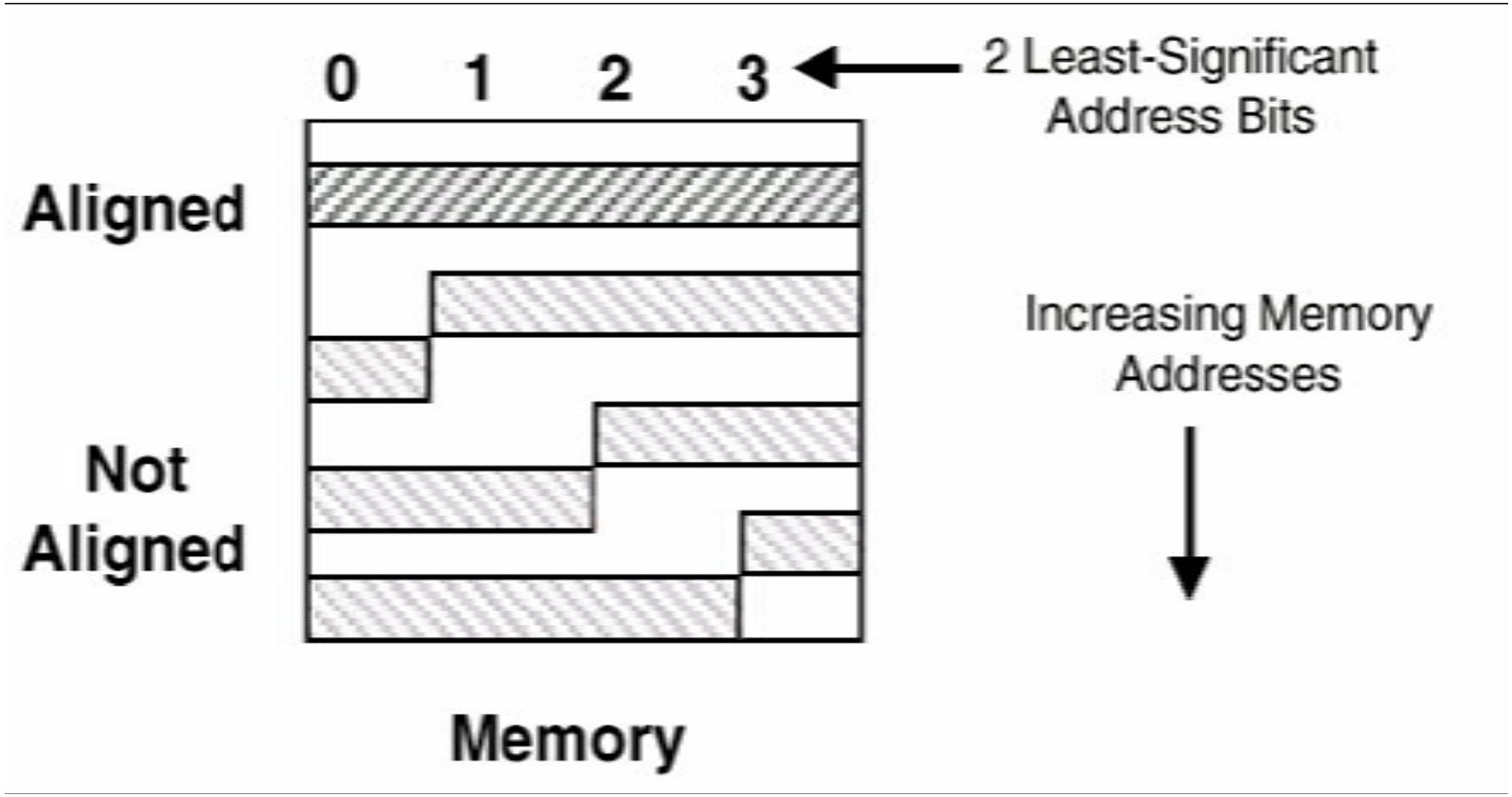
Registers hold 32 bits of data

- **2^{32} bytes with byte addresses from 0 to 2^{32}-1**
- **2^{30} words with byte addresses 0, 4, 8, ... 2^{32}-4**
- **Words are aligned**
  i.e., Address must be a multiple of 4
  - what are the least 2 significant bits of a word address?

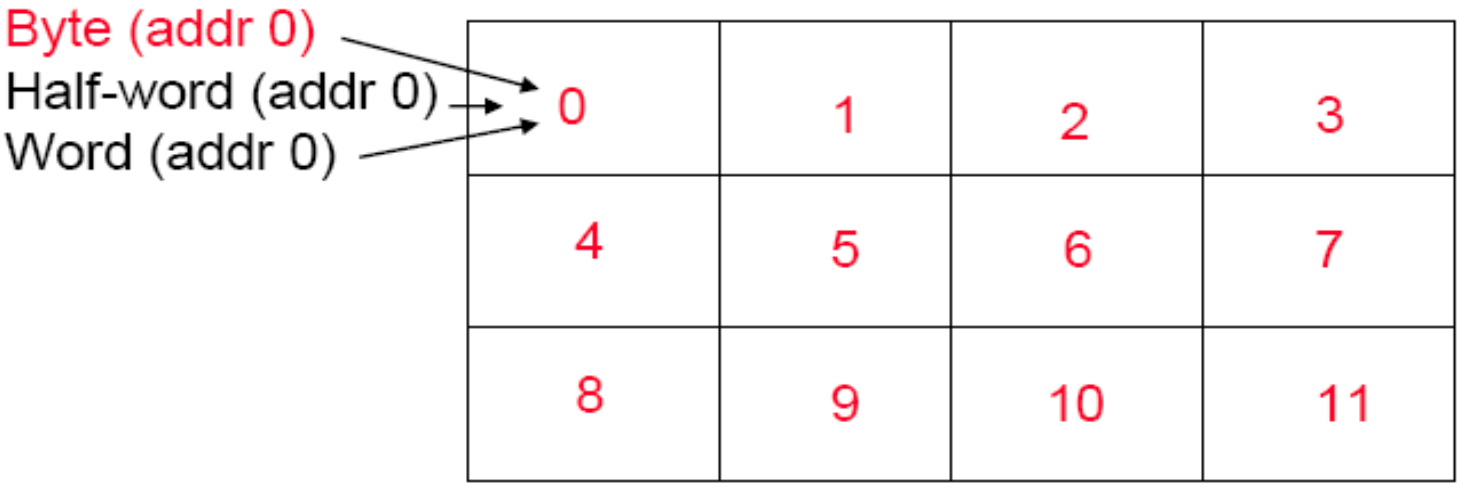
Alignment

- **Alignment**
  - require that each object be located at an address that is multiple of its size
- **Important performance effect**
  - also logical simplification
    - removes complexity of sequencing memory references
- **Historically**
  - early machines (IBM 360 in 1964) require alignment
  - restriction removed in 1970s
  - RISC machines: reintroduce restriction for performance and simplicity

Alignment Example: 32-Bit Word



Addressing Examples



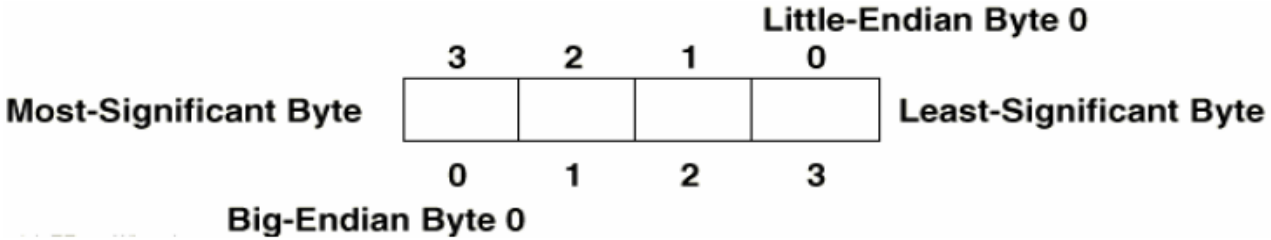
Little Endian vs. Big Endian

MIPS is Big Endian

Byte Ordering

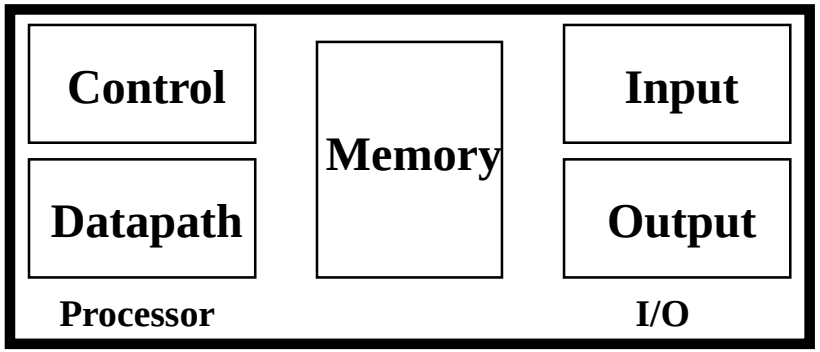
The terms derive from one of the satirical conflicts in Gulliver's Travels, in which two religious sects of Lilliputians are divided between those who prefer cracking open their soft-boiled eggs from the little end, and those who prefer the big end.

- **Two Conventions**
  - Big Endian
    - specify address of most significant byte
  - Little Endian
    - specify address of least significant byte
  - no technical significance to the distinction
    - just an arbitrary historical difference
      - Big Endian machines: Amiga, 68K Macs, IBM RS6K, SGI, Sun
      - Little Endian machines: Alpha, DEC, Vax, x86
    - recently many processors are "bimodal"
      - MIPS, PowerPC (both mostly Big Endian)



Registers vs. Memory

- **Arithmetic instructions operands must be registers,**
  — only 32 registers provided
- **Compiler associates variables with registers**
- **Registers are faster to access than memory**
- **Operating on memory data requires loads and stores**
  - More instructions to be executed
- **What about programs with lots of variables**
  - Spilling registers, Only spill to memory for less frequently used variables
  - For high performance, use registers efficiently!



Instructions

- **Load and store instructions**
- **Example:**

C code: `A[12] = h + A[8];`

MIPS code:

```
lw  $t0, 32($s3)
add $t0, $s2, $t0
sw  $t0, 48($s3)
```

base register: **$s3**
offset: 32

- **Can refer to registers by name (e.g., `$s2, $t2`) instead of number**
- **Store word has destination last**
- **Remember arithmetic operands are registers, not memory!**

Can't write: `add 48($s3), $s2, 32($s3)`

Example: Compiling using a Variable Array Index

- **C code: g = h + A[i]**
  - `$s3`: base register for A
  - g, h, i: `$s1, $s2, $s4`

- **MIPS code:**

```
add $t1, $s4, $s4       # $t1 = 2 * i
add $t1, $t1, $t1       # $t1 = 4 * i

add $t1, $t1, $s3       # $t1 = address of A[i]
lw  $t0, 0($t1)         # $t0 = A[i]

add $s1, $s2, $t0       # g = h + A[i]
```

Constants (Immediate Operands)

- **Small constants are used quite frequently (50% of operands)**

  e.g., A = A + 5;
  B = B + 1;
  C = C - 18;

- **Solutions? Why not?**
  - put 'typical constants' in memory and load them.
  - create hard-wired registers (like $zero) for constants like zero.

- **MIPS Instructions:**

```
addi $29, $29, 4
slti $8,  $18, 10
andi $29, $29, 6
ori  $29, $29, 4
```

No subtract immediate instruction

- Just use a negative constant

```
addi $s2, $s1, -1
```

MIPS register 0 ($zero) is the constant 0

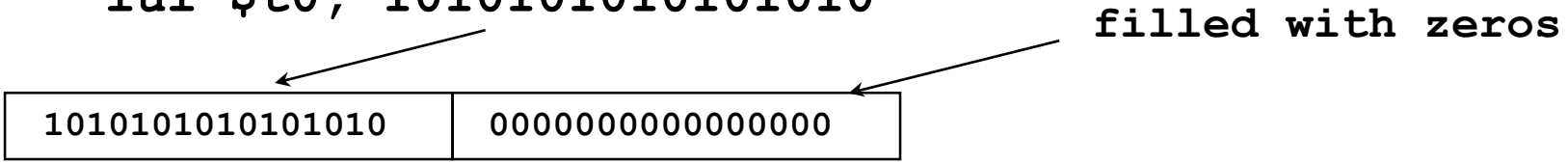
- Cannot be overwritten

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

How about larger constants?

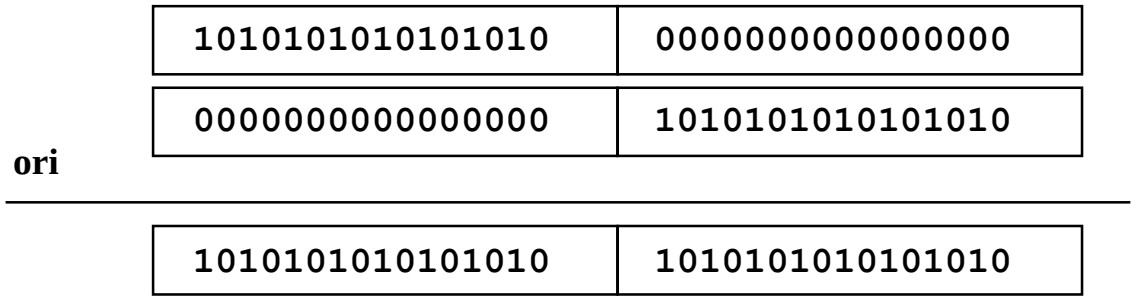
- **We'd like to be able to load a 32 bit constant into a register**
- **Must use two instructions, new "load upper immediate" instruction**

```
lui $t0, 1010101010101010
```



- **Then must get the lower order bits right, i.e.,**

```
ori $t0, $t0, 1010101010101010
```



Numbers

- **Binary numbers (base 2)**

  0000 0001 0010 0011 0100 0101 0110 0111 1000 1001...

  decimal: $0...2^n-1$

- **Of course it gets more complicated:**

  numbers are finite (overflow)

  fractions and real numbers

  negative numbers

  e.g., no MIPS subi instruction; addi can add a negative number

- **How do we represent negative numbers?**

  i.e., which bit patterns will represent which numbers?

Possible Representations

Sign Magnitude:	One's Complement	Two's Complement
000 = +0	000 = +0	000 = +0
001 = +1	001 = +1	001 = +1
010 = +2	010 = +2	010 = +2
011 = +3	011 = +3	011 = +3
100 = -0	100 = -3	100 = -4
101 = -1	101 = -2	101 = -3
110 = -2	110 = -1	110 = -2
111 = -3	111 = -0	111 = -1

- **Issues: balance, number of zeros, ease of operations**
- **Which one is best? Why?**

- **32 bit signed numbers:**

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = +\ 1_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = +\ 2_{ten}$
...
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = +\ 2{,}147{,}483{,}646_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = +\ 2{,}147{,}483{,}647_{ten}$ — *maxint*
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -\ 2{,}147{,}483{,}648_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -\ 2{,}147{,}483{,}647_{ten}$ — *minint*
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -\ 2{,}147{,}483{,}646_{ten}$
...
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = -\ 3_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -\ 2_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -\ 1_{ten}$

Two's Complement Operations

- **Negating a two's complement number: invert all bits and add 1**
  - remember: "negate" and "invert" are quite different!
- **Converting n bit numbers into numbers with more than n bits:**
  - MIPS 16 bit immediate gets converted to 32 bits for arithmetic
  - copy the most significant bit (the sign bit) into the other bits

    `+2: 0000 0010 => 0000 0000 0000 0010`

    `-2: 1111 1110 => 1111 1111 1111 1110`
  - "sign extension" (lbu vs. lb)

$$x + \bar{x} = 1111...111_2 = -1$$

$$\bar{x} + 1 = -x$$

So far we've learned:

- **MIPS**
  - **— loading words but addressing bytes**
  - **— arithmetic on registers only**

- **Instruction** **Meaning**

```
add $s1, $s2, $s3      $s1 = $s2 + $s3
sub $s1, $s2, $s3      $s1 = $s2 - $s3
lw $s1, 100($s2)       $s1 = Memory[$s2+100]
sw $s1, 100($s2)       Memory[$s2+100] = $s1
```

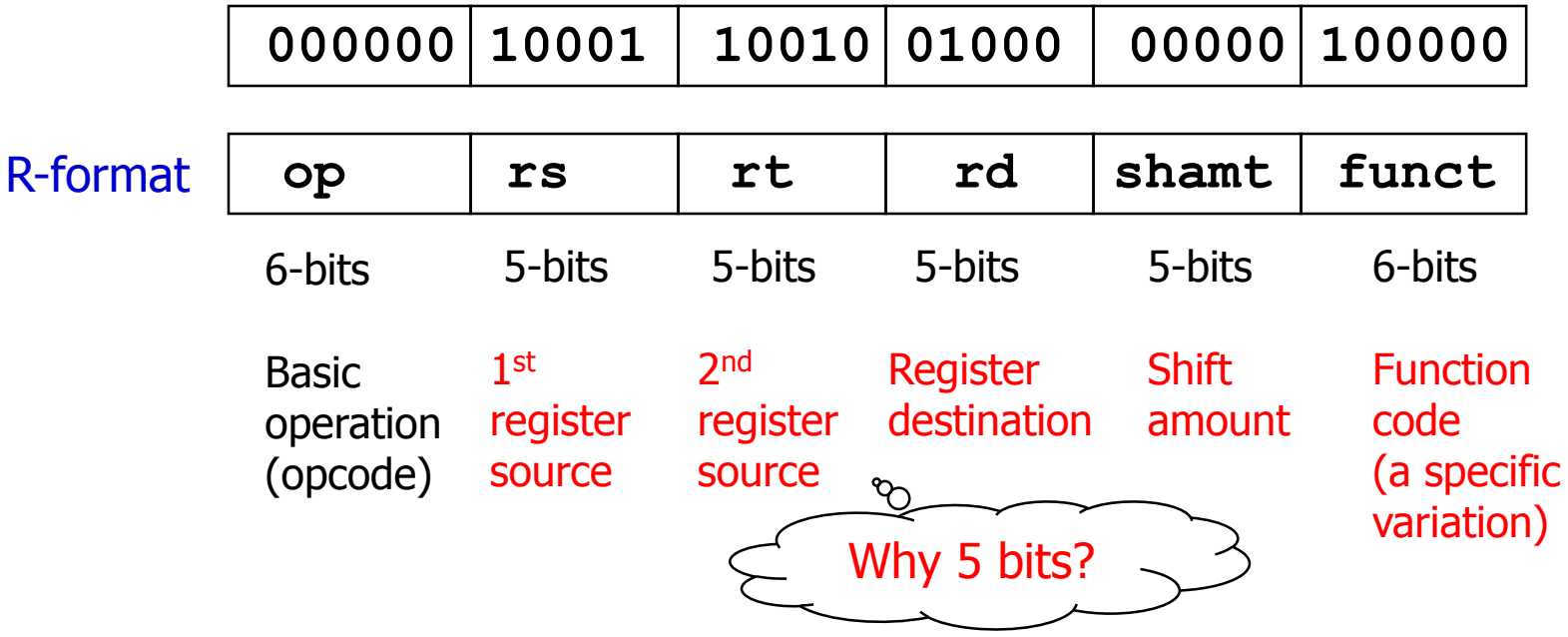
Register numbers

$t0 – $t7 are reg's 8 – 15
$t8 – $t9 are reg's 24 – 25
$s0 – $s7 are reg's 16 – 23

Representing Machine Instructions

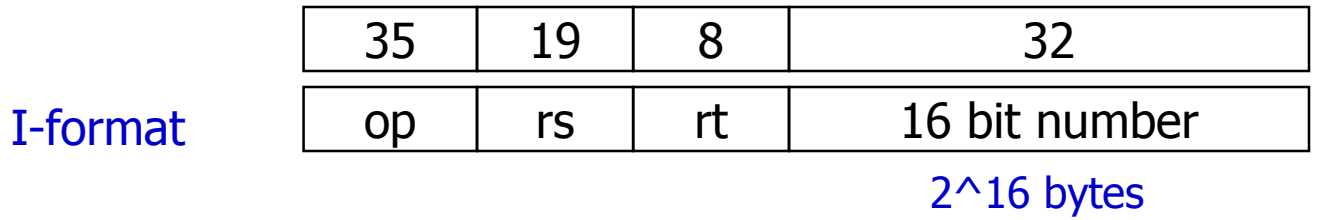
- **Instructions, like registers and words of data, are also 32 bits long**
  - Example: `add $t0, $s1, $s2`
  - registers have numbers, `$t0=8, $s1=17, $s2=18`

- **Instruction Format:**



Representing Machine Instructions

- **Consider the load-word and store-word instructions,**
  - What would the regularity principle have us do?
- **Design Principle 4: Good design demands a compromise.**
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible
- **Introduce a new type of instruction format**
  - I-type for data transfer instructions
  - other format was R-type for register
- **Example:** `lw $t0, 32($s3)  # A[8]`



Where's the compromise?

Instruction Format == Instruction Encoding

- **ISA defines the format of an instruction (syntax) and what it does (semantics).**
- **Each instruction format has different fields.**
- **Opcode: what the instruction does (semantics)**
  - 6 bits => 64 instructions only? Use funct.
    - add inst: op=0 funct = 32
    - sub inst: op=0 funct = 34

- **Operand fields: where to find input(s) and output(s) of the instruction**

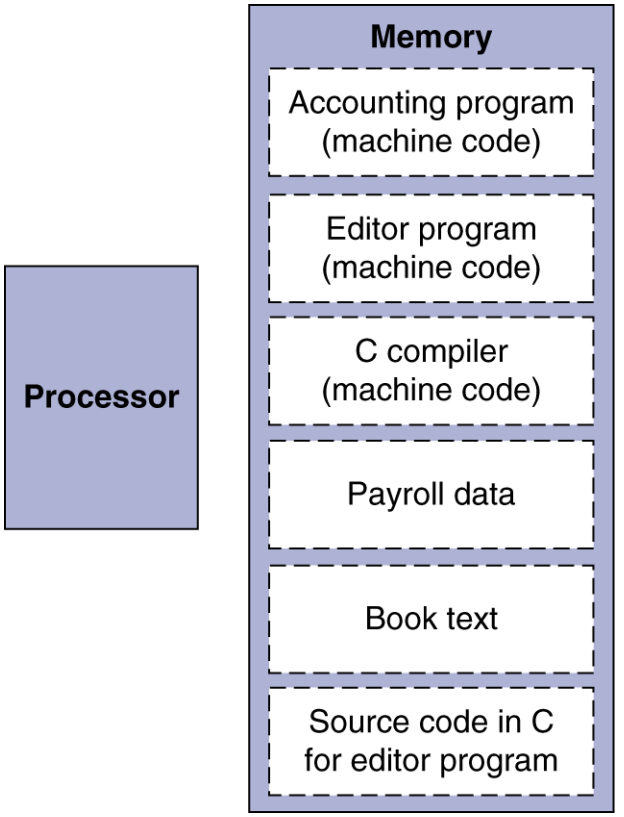
MIPS Instruction Encoding

- **MIPS is a RISC (Reduced Instruction Set Computer)**
  - vs. CISC (Complex Instruction Set Computer)
- **Main Motivation for RISC Architecture**
  - All instructions are of the same size, 32 bits.
  - The formats are consistent with each other.

Stored Program Concept

- **Instructions represented in binary, just like data**
- **Instructions stored in memory**
  - to be read or written just like data
- **Fetch & Execute Cycle**
  - Instructions are fetched and put into a special register
  - Bits in the register "control" the subsequent actions
  - Fetch the “next” instruction and continue

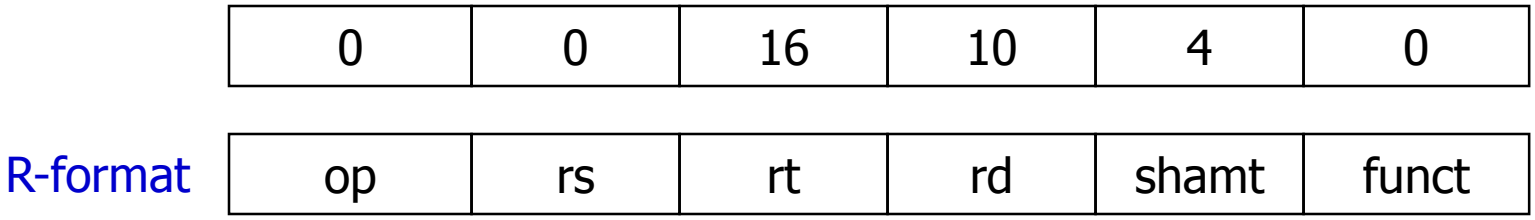
Treating Instructions in the same way as Data



Logical Operations

- `and  $s1, $s2, $3` `# $s1 = $s2 & $s3`
- `or  $s1, $s2, $3` `# $s1 = $s2 | $s3`
- `nor  $s1, $s2, $3` `# $s1 = ~($s2 | $s3)`
- `andi $s1, $s2, 100` `# $s1 = $s2 & 100`
- `ori  $s1, $s2, 100` `# $s1 = $s2 | 100`
- `sll  $s1, $s2, 10` `# $s1 = $s2 << 10`
- `srl  $s1, $s2, 10` `# $s1 = $s2 >> 10`

```
sll $t2, $s0, 4      # $t2 = $s0 << 4
```



Control

- **Decision making instructions**
  - alter the control flow,
  - i.e., change the "next" instruction to be executed

- **MIPS conditional branch instructions:**

```
bne $t0, $t1, Label
beq $t0, $t1, Label
```

- **Example:** `if (i==j) h = i + j;`

```
        bne $s0, $s1, Label
        add $s3, $s0, $s1
Label:      ....
```

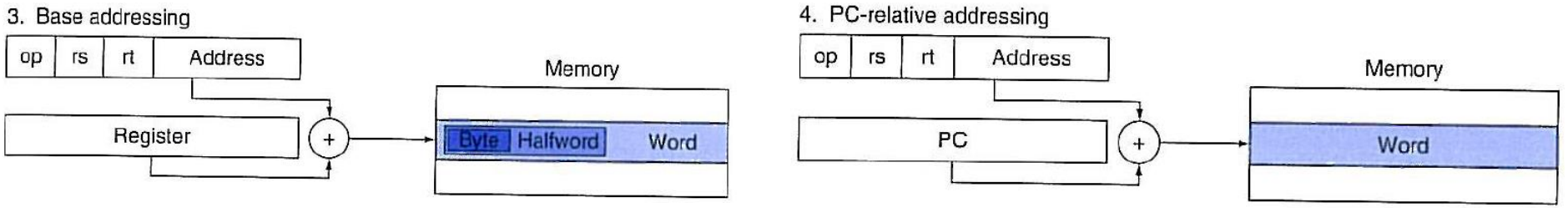
Addresses in Branches

- **bne $s0, $s1, Exit**

Branch address size problem:
No Program Can be Bigger Then 2^16

PC-relative Addressing

- **To solve the branch address size problem,**
  - PC = $\text{Register}_{\text{base}}$ + Branch Offset
  - Observation: branch targets tend to be nearby instructions.
- **PC-relative addressing**
  - $\text{Register}_{\text{base}}$ = PC
  - Actually relative to PC+4 (as opposed to the current PC)



Control

- **MIPS unconditional branch instructions:**

```
j  label
```

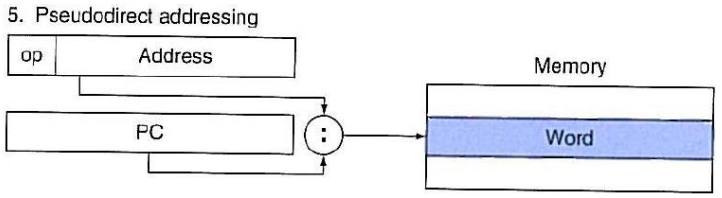
- **Example:**

```
if (i!=j)                    beq $s4, $s5, Lab1
    h=i+j;                   add $s3, $s4, $s5
else                         j Lab2
    h=i-j;           Lab1:   sub $s3, $s4, $s5
                     Lab2:   ...
```

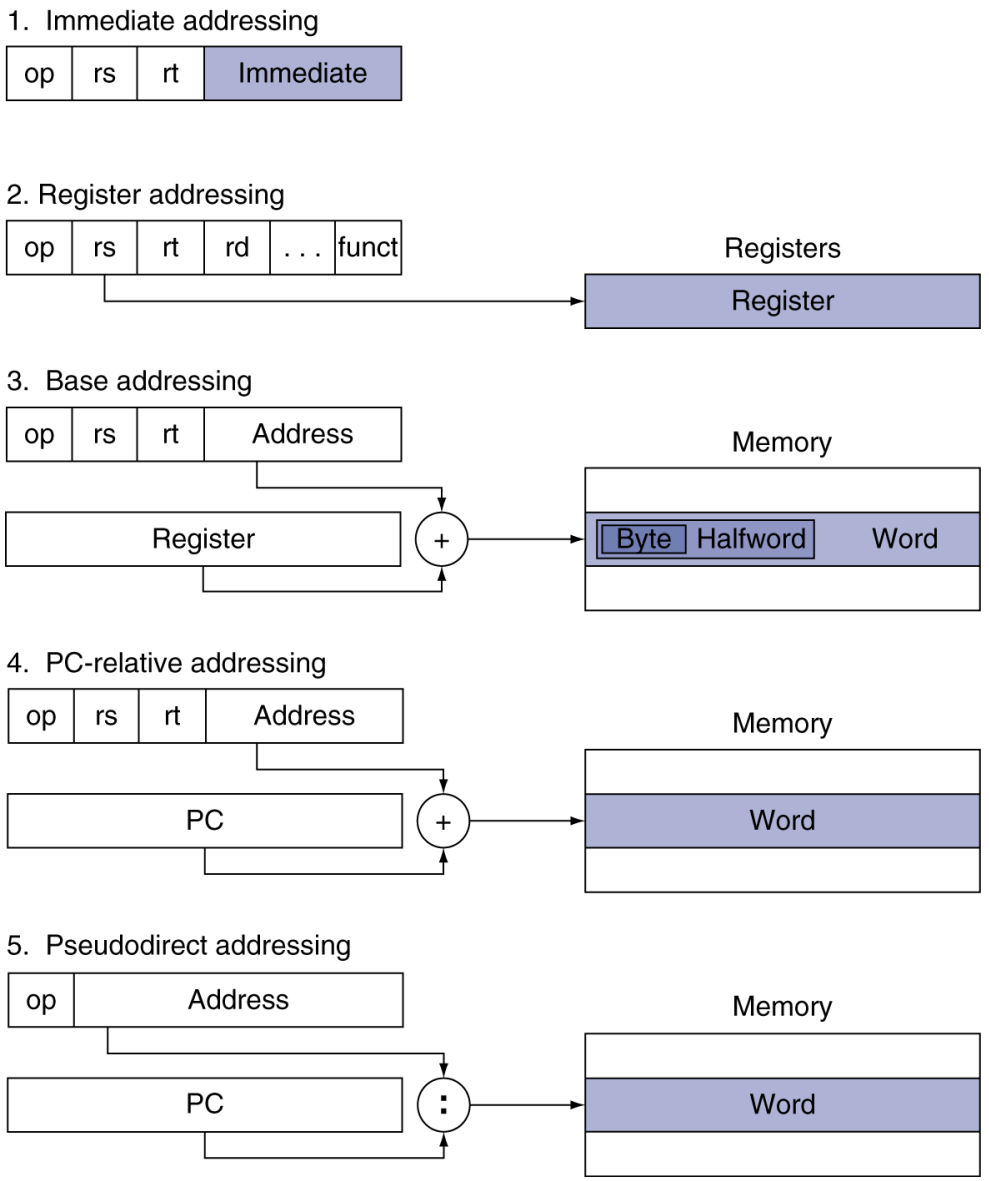
- **Jump instructions just use high order bits of PC**
  - address boundaries of 256 MB (== 2^26 WORDS)

J-format

2	10000
Op code	26 bits



MIPS Addressing Modes



Example: a while loop

- **C code:** while (save[i] == k)
  i = i + 1;

  the base addr. of save in **`$s6`**
  i, k: **`$s3, $s5`**

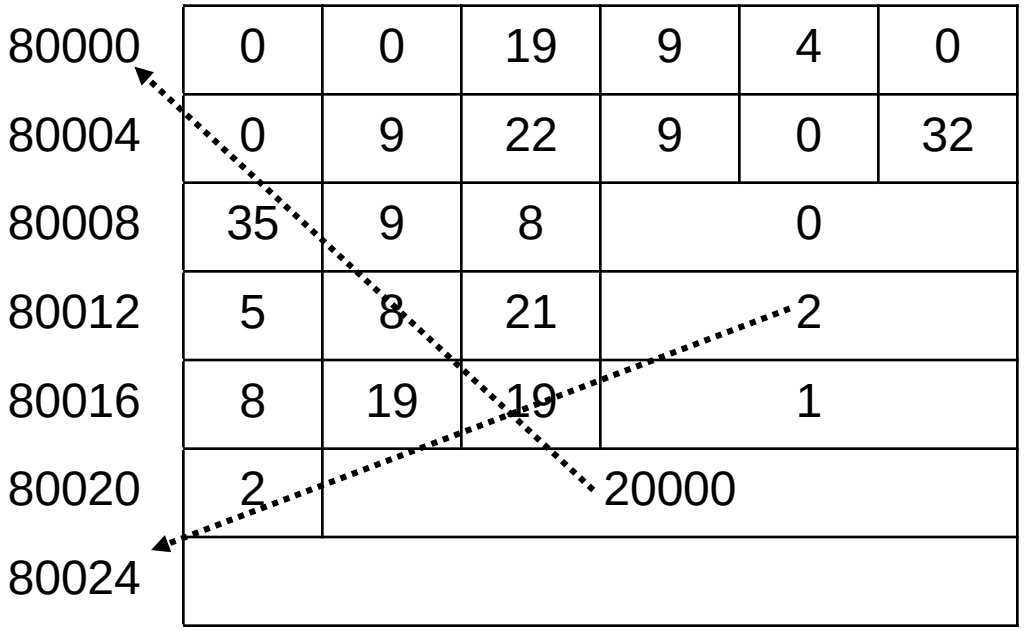
- **MIPS code:**

```
Loop:   sll $t1, $s3, 2       # $t1 = 4 * i
        add $t1, $t1, $s6     # $t1 = addr. of save[i]
        lw  $t0, 0($t1)       # $t0 = save[i]
        bne $t0, $s5, Exit
        addi $s3, $s3, 1      # i = i + 1
        j Loop                # go to Loop
Exit:
```

Target Addressing Example

- **Loop code from earlier example**
  - Assume Loop at location 80000

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

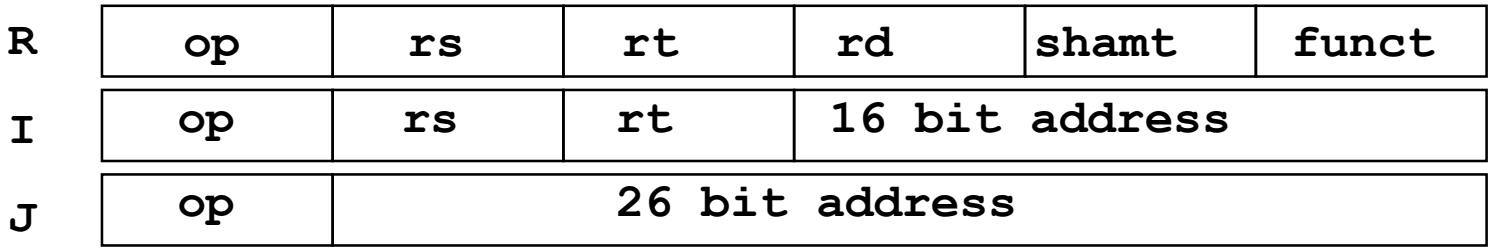


So far:

- **Instruction** **Meaning**

```
add $s1,$s2,$s3      $s1 = $s2 + $s3
sub $s1,$s2,$s3      $s1 = $s2 - $s3
lw $s1,100($s2)      $s1 = Memory[$s2+100]
sw $s1,100($s2)      Memory[$s2+100] = $s1
bne $s4,$s5,L        Next instr. is at Label if
                     $s4 ≠ $s5
beq $s4,$s5,L        Next instr. is at Label if
                     $s4 = $s5
j Label              Next instr. is at Label
```

- **Formats:**



Control Flow

- **We have: beq, bne, what about Branch-if-less-than?**
- **New instruction:**

```
slt $t0, $s1, $s2
```

```
if  $s1 < $s2 then
   $t0 = 1
   else
   $t0 = 0
```

- **Can use this instruction to build "`blt $s1, $s2, Label`"**
  - can now build general control structures
  - **`Use $zero($0)`** register ← **`if (i > 0)`** …
- **Why does not MIPS provide '`blt`' instruction ?**
  - Too complicated, so stretch the clock cycle time

```
slt $t0, $s1, $s2
bne $t0, $zero, Label1
```

Case/Switch Statement

- **A chain of if-then-else statements**
- **jump address table**
  - an array of words containing jump target addresses
  - New Instruction:

```
jr $t0          # jump based on register $t0
```