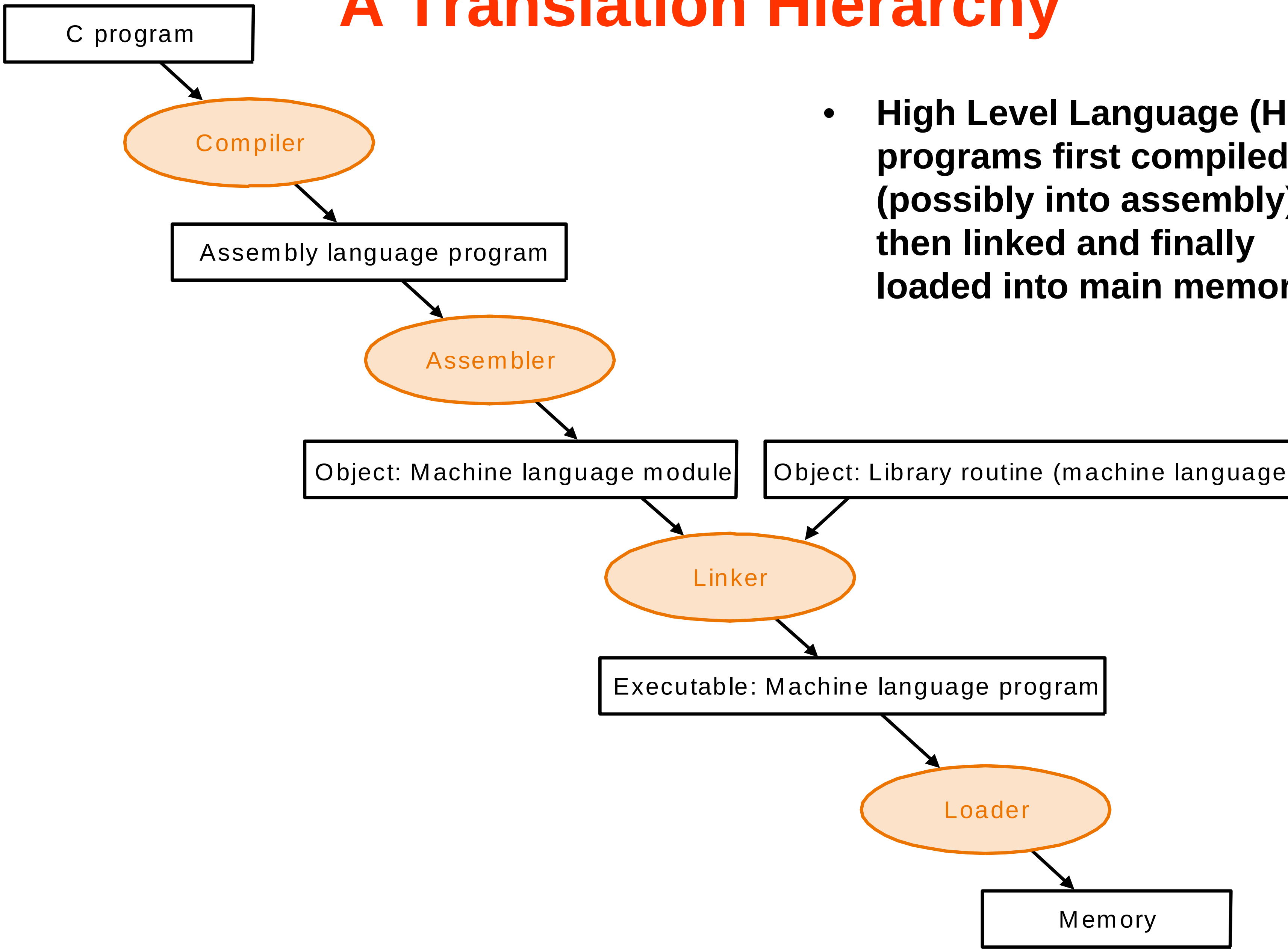
Part I
Introduction to MIPS
Instruction Set Architecture

A Translation Hierarchy

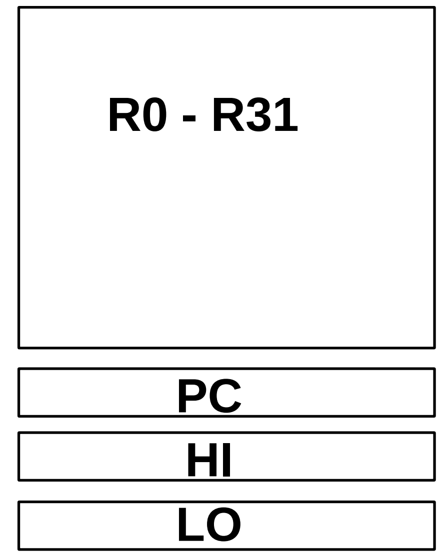


- **High Level Language (HLL) programs first compiled (possibly into assembly), then linked and finally loaded into main memory.**

MIPS R3000 Instruction Set Architecture (*Summary*)

- **Machine Environment Target**
- **Instruction Categories**
  - Load/Store
  - Computational
  - Jump and Branch
  - Floating Point (coprocessor)
  - Memory Management
  - Special

Registers



3 Instruction Formats: all 32 bits wide

R:	OP	Rs	Rt	rd	sa	funct
I:	OP	Rs	Rt	Immediate		
J:	OP	jump target				

Machine Language Instructions

- **Design Goals [Burks, Coldstine, Von Neuman, 1947]:**
  Simplicity in resources each instruction needs,
  Clarity in definition and application,
  Efficient implementation by underlying h/w.
- **More primitive than higher level languages.**
  e.*g.*, no sophisticated control flow.

- **Very restrictive**
  ***e.g.*, MIPS Arithmetic Instructions.**

- **We'll be working with the MIPS instruction set architecture (www.mips.org definitive site).**
  similar to other architectures developed since the 1980's.
  used by NEC, Nintendo, Silicon Graphics, Sony.

Review C Operators/operands

- **Operators: +, -, `*`, `/`, % (mod); (7/4==1, 7%4==3)**

- **Operands:**
  - *Variables*: `fahr`, `celsius`
  - *Constants*: `0`, `1000`, `-17`, `15.4`

- **In C (and most High Level Languages) variables declared and given a type first**
  - Example:

```
int fahr, celsius;
int a, b, c, d, e;
```

C Operators/operands

- **This is the Computation Model by ``State-Effects''.**

  Programs move from state to state by means of assignments changing their state

- **Assignment Statement:**

  Variable = expression, e.g.,

  celsius = 5*(fahr-32)/9;

  a = b+c+d-e;

Assembly Operators

- **Syntax of Assembly Operator**

  1) operation by name "Mnemonics"

  2) operand getting result Register or Memory

  3) 1st operand for operation

  4) 2nd operand for operation

- **Ex. add b to c and put the result in a:** **add a, b, c**

  Called an Assembly Language Instruction

- **Equivalent assignment statement in C:**

  a = b + c;

Assembly Operators/instructions

- **How to do the following C statement?**
  a = b + c + d - e;

- **Break into multiple instructions**

```
add  a, b, c    #  a = sum of b & c
add  a, a, d    #  a = sum of b,c,d
sub  a, a, e    #  a = b+c+d-e
```

- **To right of sharp sign (#) is a *comment* terminated by end of the line. Applies only to current line.**

  C comments have format /* comment */ ,   can span many lines

Assembly Operators/instructions

- **Note: Unlike C (and most other HLLs), each *line* of assembly contains at most one instruction**

add a,b,c add d,e,f **WRONG**

add a,b,c
add d,e,f **RIGHT**

Compilation

- **How to turn the notation that programmers prefer into notation computer understands?**
- **Program to translate C statements into Assembly Language instructions; called a <u>compiler</u>**
- **Example: compile by hand this C code:**

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

- **Easy:**

```
add   a, b, c
sub   d, a, e
```

- **<u>Big Idea: compiler translates notation from one level of computing abstraction to lower level</u>**

Compilation 2

- **Example: compile by hand this C code:**
  f = (g + h) - (i + j);

- **First sum of g and h. Where to put result?**

  Add f, g, h *# f contains g+h*

- **Now sum of i and j. Where to put result?**

  Cannot use f !

  Compiler creates temporary variable to hold sum: t1
  add t1, i, j *# t1 contains i+j*

- **Finally produce difference**
  sub f, f, t1 ***# f = (g+h)-(i+j)***

Compilation -- Summary

- **C statement (5 operands, 3 operators):**

  f = (g + h) - (i + j);

- **Becomes 3 assembly instructions**
  (6 unique operands, 3 operators):

  add f,g,h # *f contains g+h*

  add t1,i,j # *t1 contains i+j*

  sub f,f,t1 # *f*=(g+*h*)-(i+*j*)

- **In general, each line of C produces many assembly instructions**

  One reason why people program in C *vs*. Assembly; fewer lines of code

  Other reasons? (many!)

Assembly Design: Key Concepts

- **Assembly language is essentially directly supported in hardware, therefore ...**

- **It is kept very simple!**
  - Limit on the type of operands
  - Limit on the set operations that can be done to absolute minimum.
    - if an operation can be decomposed into a simpler operation, don't include it.

Assembly Variables: Registers (1/4)

- **Unlike HLL, assembly cannot use variables**
  - Why not? Keep Hardware Simple
- **Assembly Operands are <u>registers</u>**
  - limited number of special locations built directly into the hardware
  - operations can only be performed on these!
- **Benefit: Since registers are directly in hardware, they are very fast**

Assembly Variables: Registers (2/4)

- **Drawback: Since registers are in hardware, there are a predetermined number of them**
  – Solution: MIPS code must be very carefully put together to efficiently use registers
- **32 registers in MIPS**
  – Why 32? Smaller is faster
- **Each MIPS register is 32 bits wide**
  – Groups of 32 bits called a word in MIPS

Assembly Variables: Registers (3/4)

- **Registers are numbered from 0 to 31**
- **Each register can be referred to by number or name**
- **Number references:**

  $0, $1, $2, … $30, $31

Assembly Variables: Registers (4/4)

- **By convention, each register also has a name to make it easier to code**

- **For now:**

  $16 - $22 → $s0 - $s7

  (correspond to C variables)

  $8 - $15 → $t0 - $t7

  (correspond to temporary variables)

- **In general, use register names to make your code more readable**

Assembly Instructions

- **In assembly language, each statement (called an Instruction), executes exactly one of a short list of simple commands**

- **Unlike in C (and most other High Level Languages), where each line could represent multiple operations**

Addition and Subtraction (1/3)

- **Syntax of Instructions:**

  1 2, 3, 4

  where:

  1) operation by name

  2) operand getting result (destination)

  3) 1st operand for operation (source1)

  4) 2nd operand for operation (source2)

- **Syntax is rigid:**
  - 1 operator, 3 operands
  - Why? Keep Hardware simple via regularity

Addition and Subtraction (2/3)

- **Addition in Assembly**
  - Example (in MIPS): add $s0, $s1, $s2
  - Equivalent to (in C): a = b + c

  where registers **$s0, $s1, $s2** are associated with variables a, b, c

- **Subtraction in Assembly**
  - Example (in MIPS): sub $s3, $s4, $s5
  - Equivalent to (in C): d = e - f

  where registers **$s3, $s4, $s5** are associated with variables d, e, f

Addition and Subtraction (3/3)

- **How to do the following C statement?**

  a = b + c + d - e;

- **Break it into multiple instructions:**

```
add    $s0, $s1, $s2    # a = b + c
add    $s0, $s0, $s3    # a = a + d
sub    $s0, $s0, $s4    # a = a - e
```

- **Notice: A single line of C may break up into several lines of MIPS.**

- **Notice: Everything after the hash mark on each line is ignored.**

Immediates

- **Immediates are numerical constants.**
- **They appear often in code, so there are special instructions for them.**
- **``Add immediate'':**

  addi $s0, $s1, 10 (in MIPS)

  F = g + 10 (in C)

  Where registers **$s0, $s1** are associated with variables f, g
- **Syntax similar to add instruction, except that last argument is a number instead of a register.**

Register Zero

- **One particular immediate, the number zero (0), appears very often in code.**

- **So we define register zero (**$0 **or $zero) to always have the value 0.**

- **This is defined in hardware, so an instruction like.**

  addi $0, $0, 5.

  Will not do anything.

- **Use this register, it's very handy!**

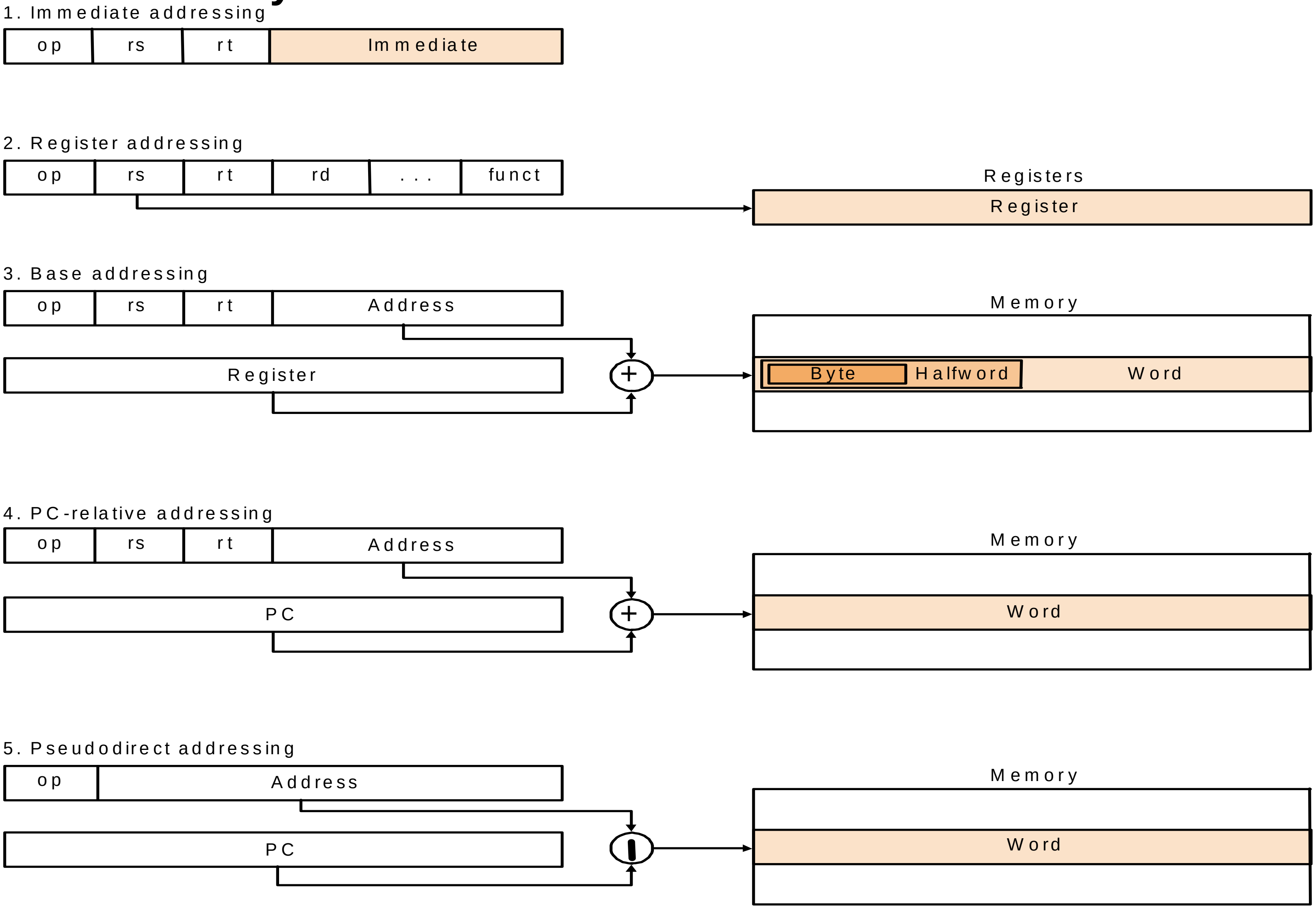
Assembly Operands: Memory

- **C variables map onto registers; What about large data structures like arrays?**
- **1 of 5 components of a computer: memory contains such data structures.**
- **But MIPS arithmetic instructions only operate on registers, never directly on memory.**

- **Data transfer instructions transfer data between registers and memory:**
  - Memory to register.
  - Register to memory.

MIPS Addressing Formats (*Summary*)

- **How memory can be addressed in MIPS**



Data Transfer: Memory to Reg (1/4)

- **To transfer a word of data, we need to specify two things:**
  - Register: specify this by number (0 - 31).
  - Memory address: more difficult.
    - Think of memory as a single one-dimensional array, so we can address it simply by supplying a pointer to a memory address.
    - Other times, we want to be able to offset from this pointer.

Data Transfer: Memory to Reg (2/4)

- **To specify a memory address to copy from, specify two things:**
  - A register which contains a pointer to memory.
  - A numerical offset (in bytes).
- **The desired memory address is the sum of these two values.**
- **Example: 8($t0).**
  - Specifies the memory address pointed to by the value in **$t0**, plus 8 bytes.

Data Transfer: Memory to Reg (3/4)

- **Load instruction syntax:**

  1   2, 3(4)

  – Where

  1) operation (instruction) name

  2) register that will receive value

  3) numerical offset in bytes

  4) register containing pointer to memory

- **Instruction name:**

  – lw (meaning load word, so 32 bits or one word are loaded at a time)

Data Transfer: Memory to Reg (4/4)

- **Example: lw $t0, 12($s0)**

  This instruction will take the pointer in $s0, add 12 bytes to it, and then load the value from the memory pointed to by this calculated sum into register $t0

- **Notes:**
  - **$S0** is called the base register
  - **12** is called the offset
  - Offset is generally used in accessing elements of array or structure: base register points to beginning of array or structure

Data Transfer: Reg to Memory

- **Also want to store value from a register into memory**
- **Store instruction syntax is identical to Load instruction syntax**
- **Instruction Name:**

sw **(meaning Store Word, so 32 bits or one word are loaded at a time)**

- **Example: sw $t0, 12($s0)**

This instruction will take the pointer in **$s0**, add 12 bytes to it, and then store the value from register $t0 into the memory address pointed to by the calculated sum

Pointers *Vs*. Values

- **Key Concept: A register can hold any 32-bit value. That value can be a (signed) int, an unsigned int, a pointer (memory address), *etc*.**

- **If you write**

  lw $t2, 0($t0)

  then, $t0 better contain a pointer

- **What if you write**

  add $t2, $t1, $t0

  then, $t0 and $t1 must contain?

Addressing: Byte *Vs*. Word

- **Every word in memory has an <u>address</u>, similar to an index in an array**
- **Early computers numbered words like C numbers elements of an array:**
  - Memory[0], memory[1], memory[2], …

Called the "<u>address</u>" of a word

- **Computers needed to access 8-bit <u>bytes</u> as well as words (4 bytes/word)**
- **Today machines address memory as bytes, hence word addresses differ by 4**

Memory[0], Memory[<u>4</u>], Memory[<u>8</u>],

Compilation With Memory

- **What offset in lw to select A[8] in C?**
- **4 x 8 = 32 to select** A[8]**: byte *vs*. Word**
- **Compile by hand using registers:**
  g = h + A[8];
  - g: $s1, h: $s2, base address of A: $s3
- **1st transfer from memory to register:**

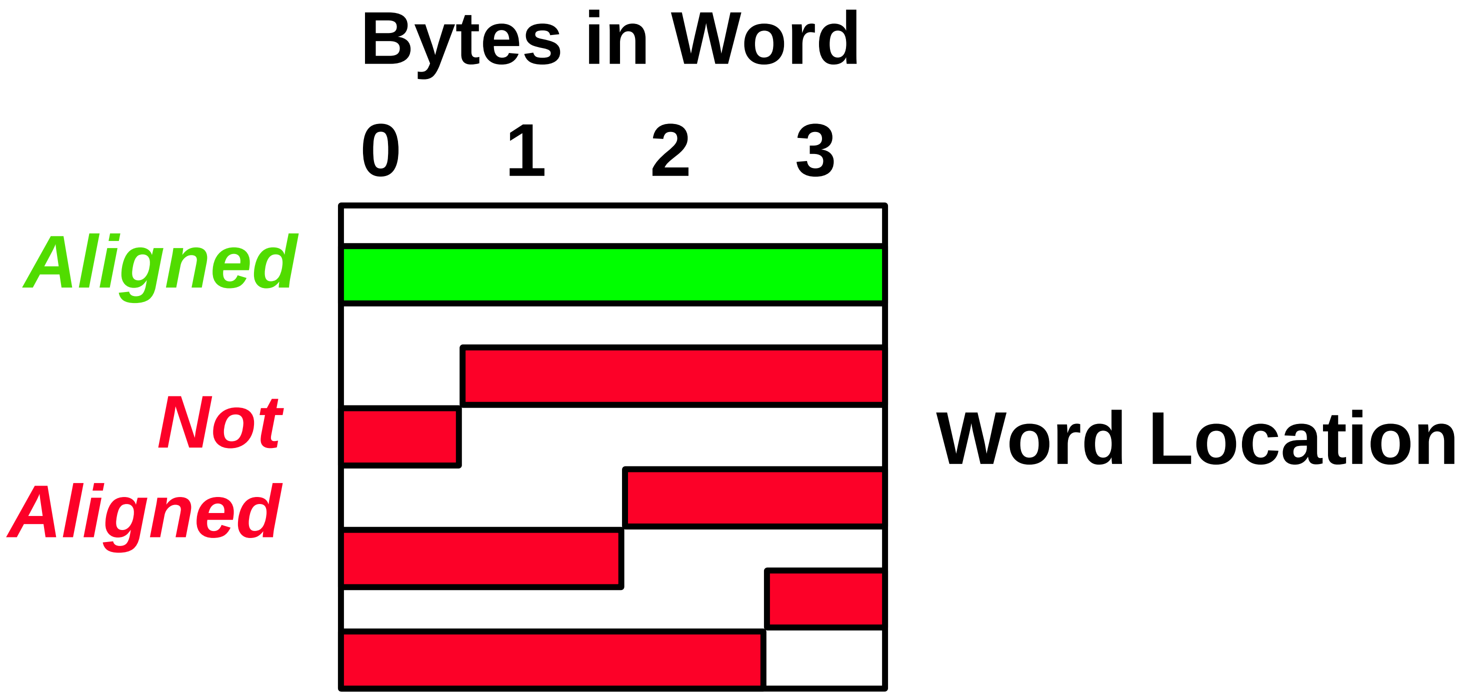
  lw $t0, 32($s3) # *$t*0 *gets A*[8]
  - Add **32** to $s3 to select A[8], put into $t0
- **Next add it to *h* and place in *g***
  add $s1, $s2, $t0 # *$s*1 = *h* + *A*[8]

Notes About Memory

- **Pitfall: forgetting that sequential word addresses in machines with byte addressing do not differ by 1.**
  - Many an assembly language programmer has toiled over errors made by assuming that the address of the next word can be found by incrementing the address in a register by 1 instead of by the word size in bytes.
  - So remember that for both lw and sw, the sum of the base address and the offset must be a multiple of 4 (to be word aligned).

More Notes About Memory: Alignment

- MIPS requires that all words start at addresses that are multiples of 4 bytes



- Called <u>Alignment</u>: objects must fall on address that is multiple of their size.

Role of Registers *Vs*. Memory

- **What if more variables than registers?**
  - Compiler tries to keep most frequently used variable in registers
  - Writing less frequently used to memory: spilling
- **Why not keep all variables in memory?**
  - Smaller is faster:
    registers are faster than memory
  - Registers more versatile:
    - MIPS arithmetic instructions can read 2, operate on them, and write 1 per instruction
    - MIPS data transfer only read or write 1 operand per instruction, and no operation

Summary (1/2)

- **In MIPS assembly language:**
  - Registers replace C variables.
  - One instruction (simple operation) per line.
  - Simpler is better.
  - Smaller is faster.
- **Memory is byte-addressable, but lw and sw access one word at a time.**
- **A pointer (used by lw and sw) is just a memory address, so we can add to it or subtract from it (using offset).**

Summary (2/2)

- **New Instructions:**

  add, addi,

  sub

  lw, sw

- **New Registers:**

  C Variables: **$s0** - $s7

  Temporary Variables: **$t0** - $t9

  Zero: **$zero**