

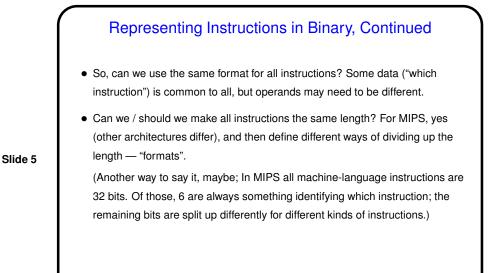
1

Sidebar: Hexadecimal

- Binary is well and good but long strings of it difficult for humans.
- More compact representation is hexadecimal (base 16). Why that one? Any base that's a power of 2 is trivially easy to convert from and to binary: Each digit in base 2^n represents n bits. (Not hard to convince yourself that this works to binary, just write down definition for other-base number and expand, while from binary, can write each group of n bits and observe that each can be written as 2^m times an n-bit number in the range 0 through $2^n 1$.
- Base 16 attractive because 4 divides typical word sizes. Base 8 (octal) once popular, and easier for most humans, and made sense when 36-bit words were more common, but not so much now.
- I'll use C convention of prefixing hexadecimal values with 0 x.
- (Short example.)

Representing Instructions in Binary, Continued

- First consider what we have to represent:
 - For all instructions, which instruction it is.
 - For add and sub, three operands (all register numbers).
 - For lw and sw, three operands (two register numbers and a "displacement").
 - And so forth ...
- So, each instruction will have "fields" consistent format for storing pieces of data, a little like a C struct.



MIPS Assembly Language Program Structure • Look at starter.s under "sample programs" on course Web site. • Overall structure mixes instructions and "directives" (things that start with .). Programs typically have two sections, one for code (starting with .text directive) and one for data (starting with . data). • For now, ignore "opening linkage" and "closing linkage" (which will make sense after we talk about procedure calls soon).

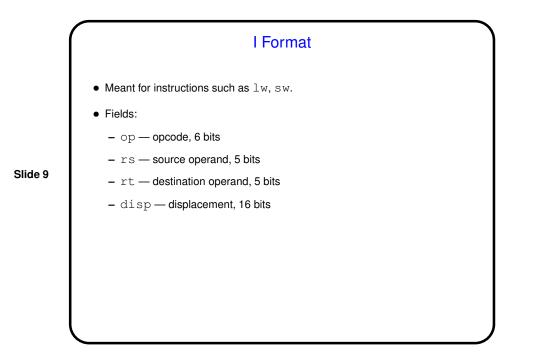


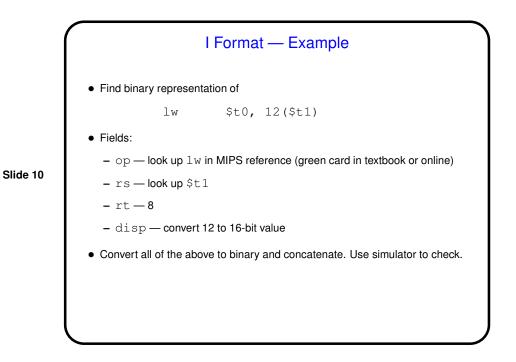
- xspim starts graphical version; most-often used buttons are probably "load" and "step".
- spim starts command-line version; commands include load, p to print, s to step. Also can run program directly (not in debug mode); useful for testing programs you think are complete and working.

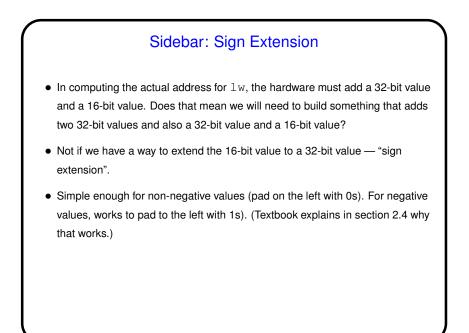
• Note that there's no machine code for la. Why? ...

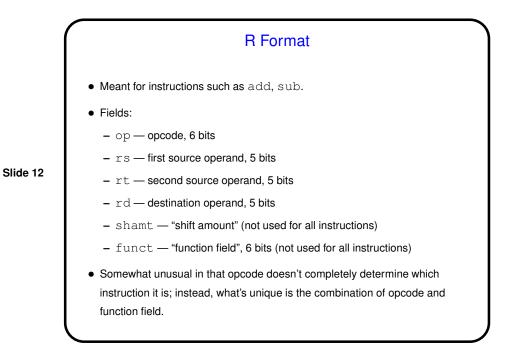
Sidebar: Pseudoinstructions

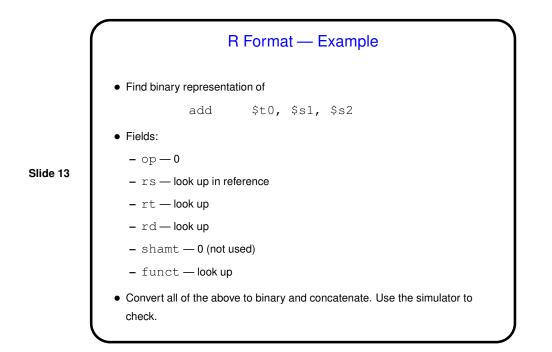
- Assemblers can also allow so-called pseudoinstructions: No matching machine instruction, but easily expanded at assembly time into one or a few "real" instructions.
- I say avoid when you can, but some (such as la) are just too useful.
- Slide 8
- (Why do I say avoid? because our goal here is not to be ace assembly-language programmers able to write compact programs, but to understand the instructions that the hardware must provide directly.)

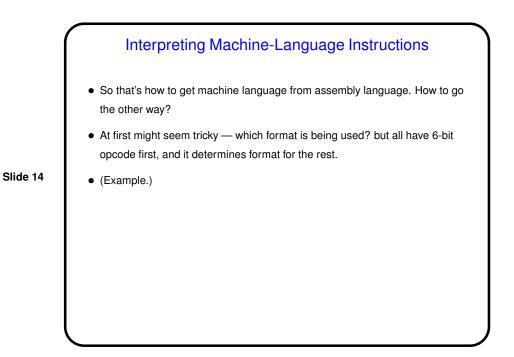














- Does one of the two instruction formats (I and R) seem like it would work for addi? If so, which one, and can you say anything about what the values of the various fields might be? If not, what fields would you need in a new format?
- Slide 15
- Anything particularly unclear?

Minute Essay Answer

• I format works — the operands of addi are two register numbers and a 16-bit constant value, same as lw and sw. Like those two instructions, it has "source" and "destination" registers, which can go in those two fields, and a 16-bit immediate value that can go in the field used for displacement in the load/store instructions.