**Instruction Representation**

Computers built on 2 key principles:

* Instructions are represented as numbers.
* Therefore, entire programs can be stored in memory to be read or written just like numbers (data).

🡪 Consequence 1: Everything has a memory address: instructions, data words

One register keeps address of instruction being executed: *“Program Counter” (PC)*

🡪 Consequence 2: Programs are distributed in binary form

Programs bound to specific instruction set

Different version for Macintosh and IBM PC

Currently all data we work with is in words (32-bit blocks):

Each register is a word.

lw and sw both access memory one word at a time.

Since data is in words, make instructions be words too (simplicity)

One word is 32 bits, so divide instruction word into *“fields”. Each field is viewed as a 5- or 6- bit unsigned integer, not as part of a 32-bit integer; i.e. 5-bit fields can represent any number 0-31, while 6-bit fields can represent any number 0-63.*

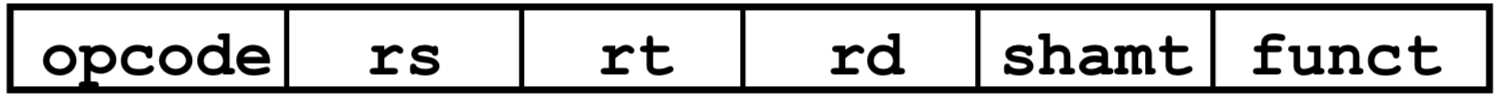
We could define different fields for each instruction, but MIPS is based on **simplicity**, so define 3 basic types of instruction formats:

* **I-format** [ Instructions with immediates, lw and sw (since the offset counts as an immediate), and the branches (beq and bne) but NOT the shift instructions ]
* **J-format** [ j and jal ]
* **R-format** [All other instructions(stands for register format) ]

**R-format:**

**All other instructions (stands for register format)**





* opcode: **partially** specifies what instruction it is. It is equal to 0 for all R-Format instructions.
* funct: combined with opcode, this number exactly specifies the instruction
* rs(Source Register): **Generally** for register containing first operand
* rt (Target Register): **Generally** for register containing second operand (name is misleading)
* rd (Destination Register): **Generally** for register which will receive result of computation
* shamt: This field contains the amount a shift instruction will shift by. Shifting a 32-bit word by more than 31 is useless, so this field is only 5 bits (so it can represent the numbers 0-31).

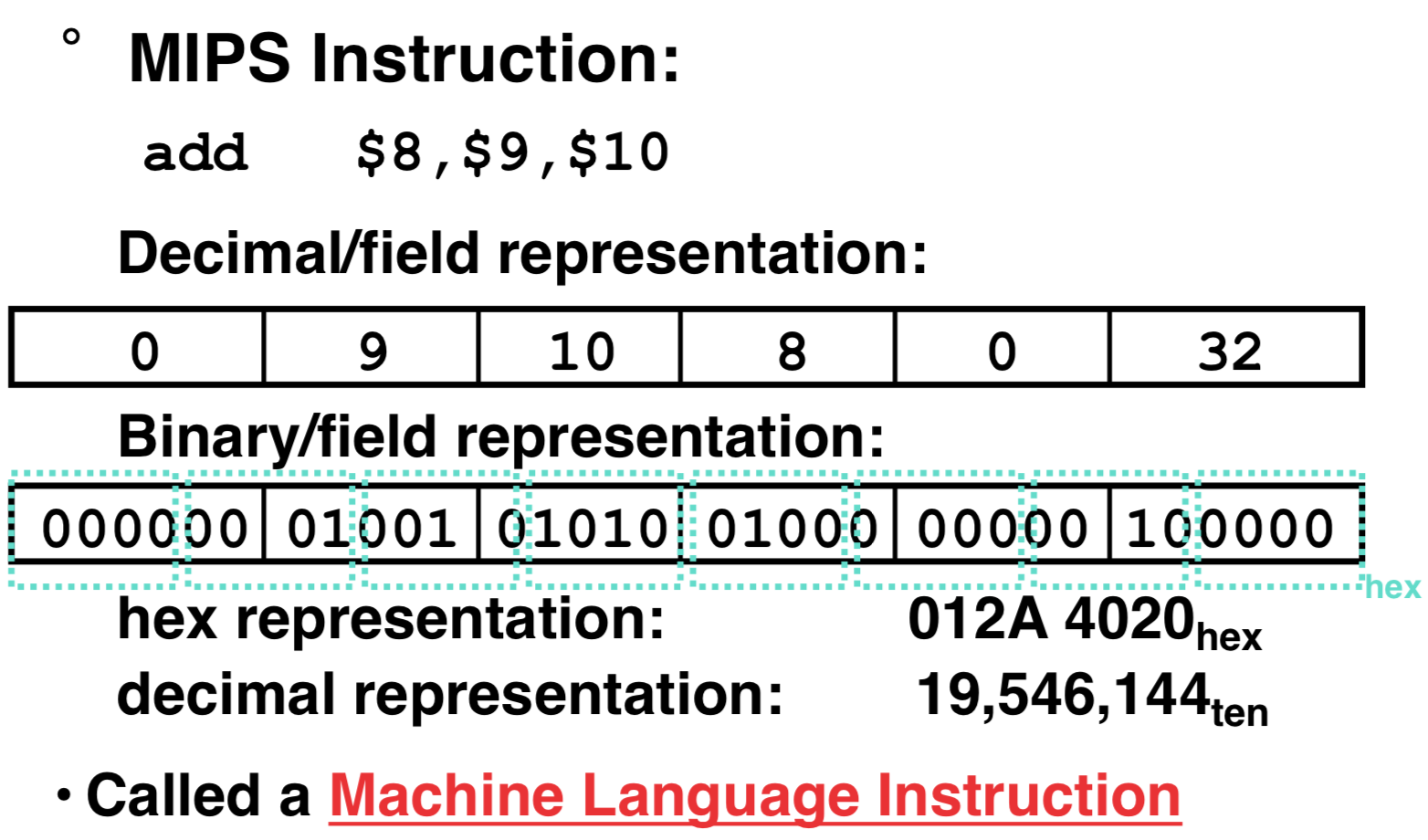
This field is set to 0 in all but the shift instructions.

Note:

1. Each register field is exactly 5 bits, which means that it can specify any unsigned integer in the range 0-31. Each of these fields specifies one of the 32 registers by number.

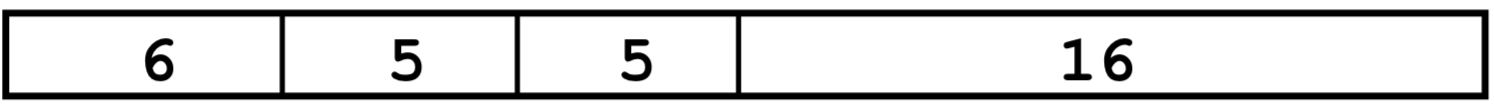
2. There are some exceptions. e.g

* mult and div have nothing important in the rd field since the dest registers are hi and lo
* mfhi and mflo have nothing important in the rs and rt fields since the source is determined by the instruction



**I-format :**

**Instructions with immediates, lw and sw (since the offset counts as an immediate), and the branches (beq and bne) (but NOT the shift instructions)**

****

****

Only one field is inconsistent with R-format. Most importantly, opcode is still in same location.

**Instructions with immediate**

* opcode: same as before except that, since there’s no funct field, opcode **uniquely** specifies an instruction in I-format. This also answers question of why R-format has two 6-bit fields to identify instruction instead of a single 12-bit field: in order to be consistent with other formats.
* rs: specifies the only register operand (if there is one)
* rt: specifies register which will receive result of computation (why it’s called the target register)
* immediate:

*addi, slti, sltiu*, the immediate is sign-extended to 32 bits. Thus, it’s treated as a signed integer.

16 bits can be used to represent immediate up to 216 different values.

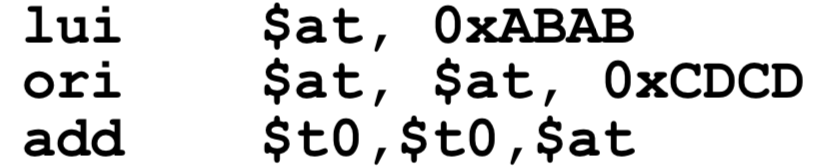
This is large enough to handle the offset in a typical *lw* or *sw*, plus a vast majority of values that will be used in the *slti* instruction.

What if immediates are too big to fit in the immediate field?

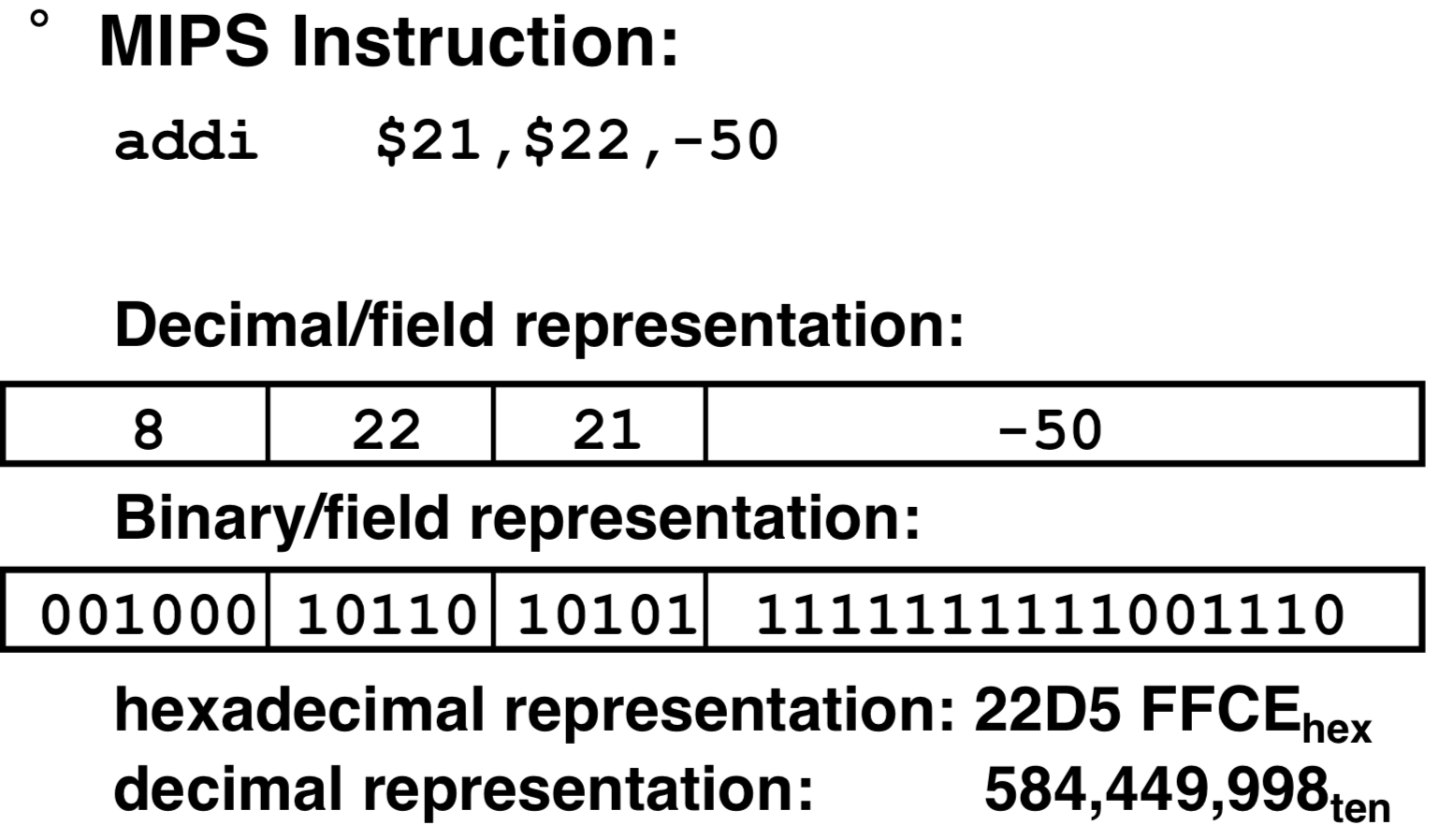
Don’t change the current instructions: instead, add a new instruction to help out



*Load Upper Immediate:* Takes 16-bit immediate and puts these bits in the upper half (high order half) of the specified register. Set lower half to 0s







Branches: PC-Relative Addressing

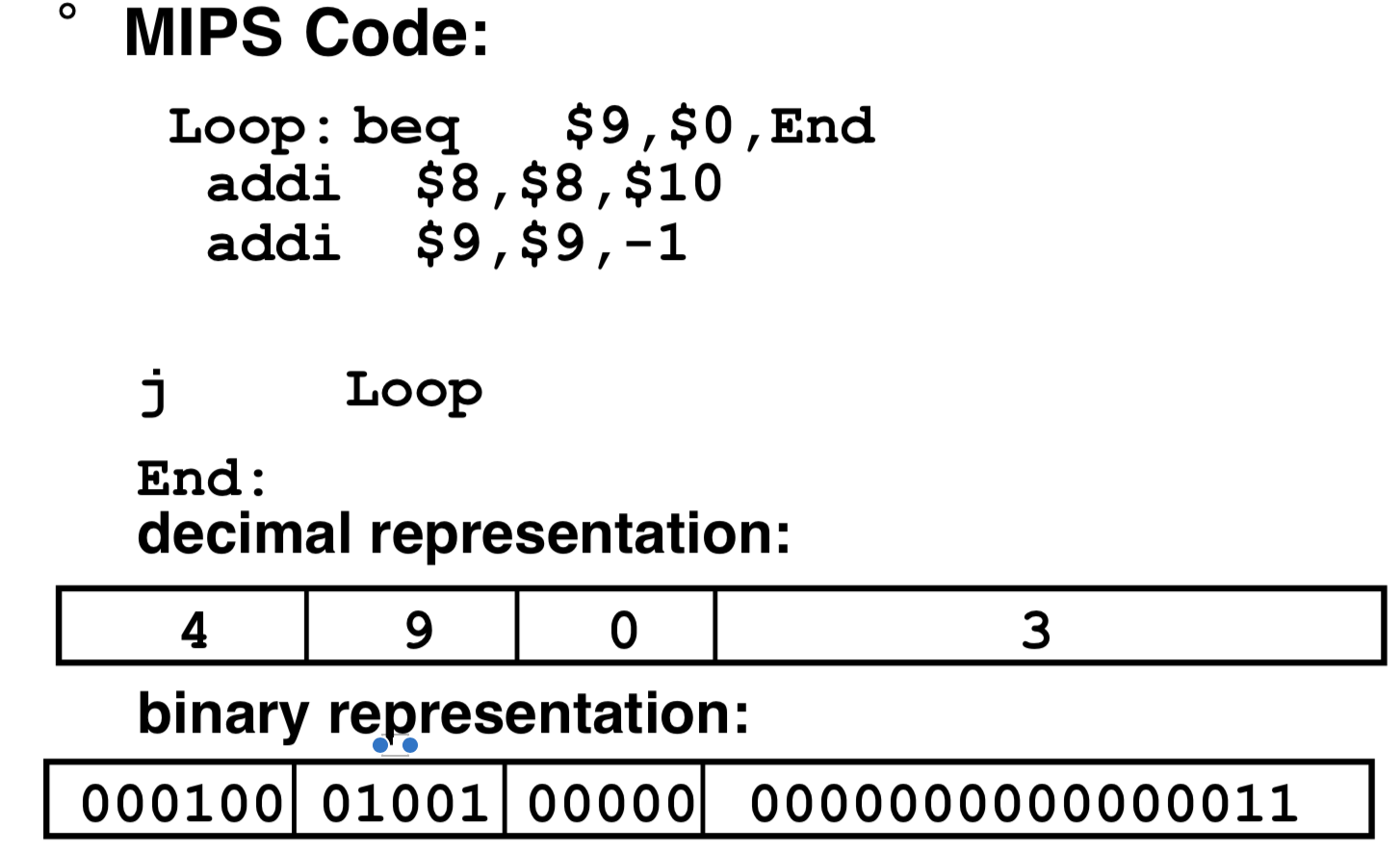


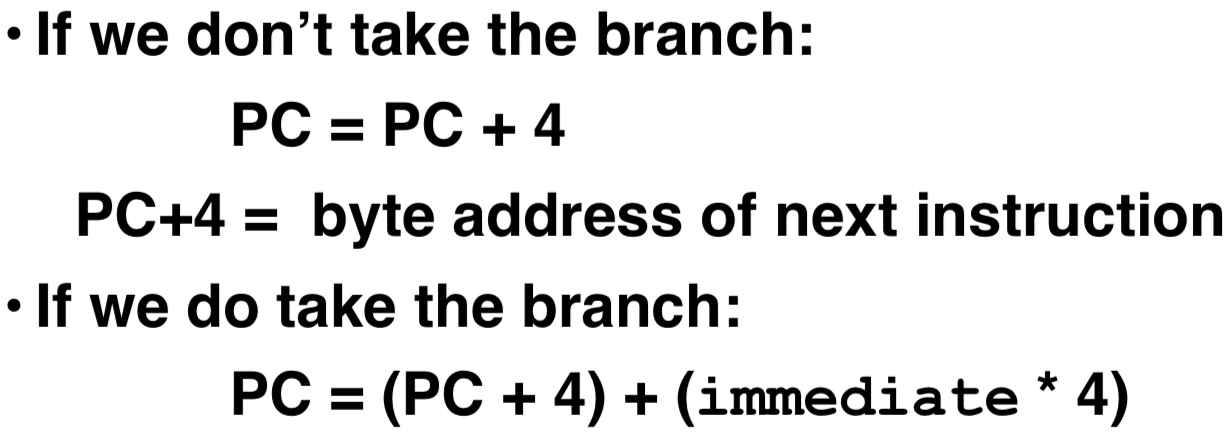
* opcode: specifies beq v. bne
* Rs and Rt: specify registers to compare
* immediate: Immediate is only 16 bits, but PC is 32-bit pointer to memory. So immediate cannot specify entire address to branch to. BUT

We usually use branches in *if-else, while, for* 🡪 Loops are generally small: typically up to 50 instructions. Though we may want to branch to anywhere in memory, a single branch will generally change the PC by a very small amount. [Note that function calls and unconditional jumps are done using jump instructions (j and jal), not the branches ]

Let the 16-bit immediate field be a signed two’s complement integer to be added to the PC if we take the branch. Now we can branch +/- 215 bytes from the PC, which should be enough to cover any loop.

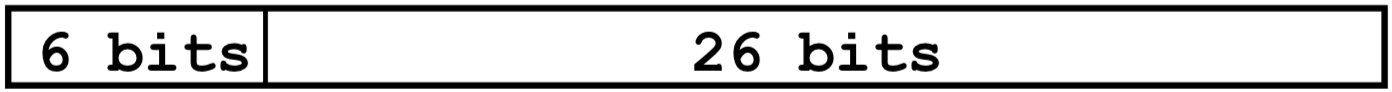
🡪 Instructions are words, so they’re word aligned (byte address is always a multiple of 4, which means it ends with 00 in binary). So the number of bytes to add to the PC will always be a multiple of 4. So specify the immediate in words. Now, we can branch **+/- 215 words** from the PC (or +/- 217 bytes), so we can handle loops 4 times as large.





**J-format:**

**j and jal**

****

****

Keep opcode field identical to R-format and I-format for consistency.

Combine all other fields to make room for large target address.

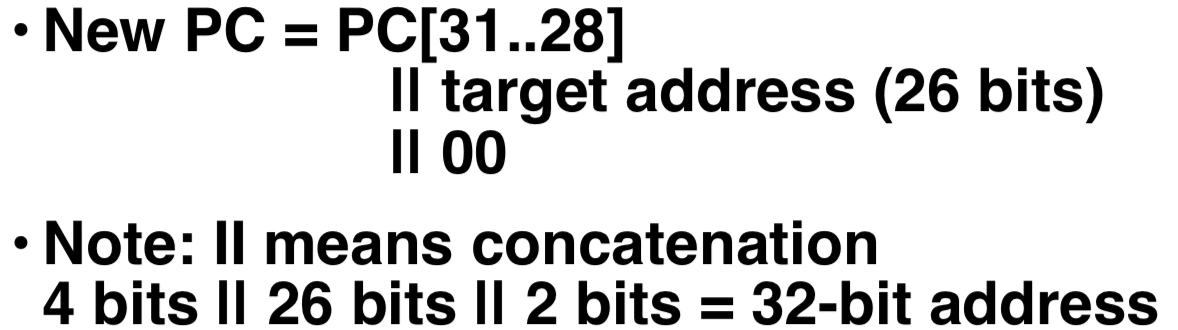
rt: Note: just like with branches, jumps will only jump to word aligned addresses, so last two bits are always 00 (in binary). So, we can specify **28 bits** of the 32-bit address.

Where do we get the other 4 bits?

• By definition, **take the 4 highest order bits from the PC.**

• Technically, this means that we cannot jump to anywhere in memory, but it’s adequate 99.9999...% of the time, since programs aren’t that long.

• If we absolutely need to specify a 32-bit address, **we can always put it in a register and use the** **jr instruction.**



**Decoding Machine Language**

How do we convert 1s and 0s to C code?

Machine language => C

For each 32 bits:

**• Look at opcode: 0 means R-Format, 2 or 3 mean J-Format, otherwise I-Format.**

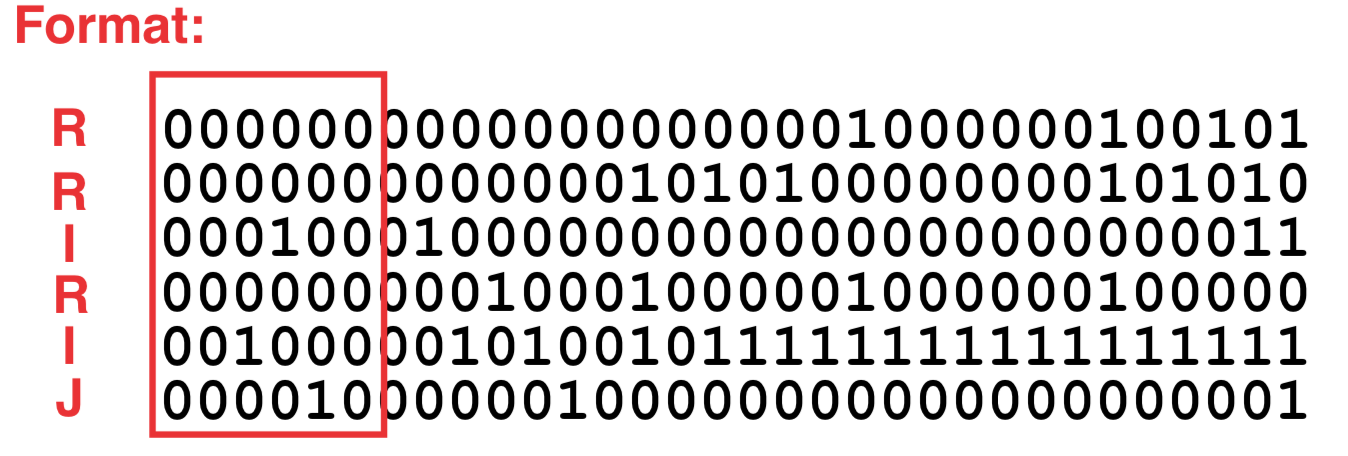
**• Use instruction type to determine which fields exist.**

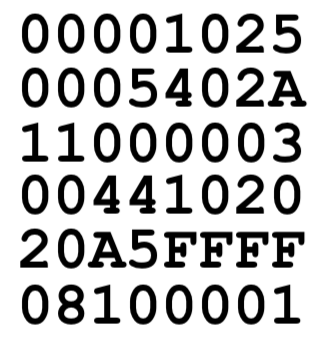
**• Write out MIPS assembly code, converting each field to name, register number/name, or decimal/hex number.**

**• Logically convert this MIPS code into valid C code. Always possible? Unique?**

Here are six machine language instructions in hex:

Let the first instruction be at address 4,194,30410 (0x00400000).

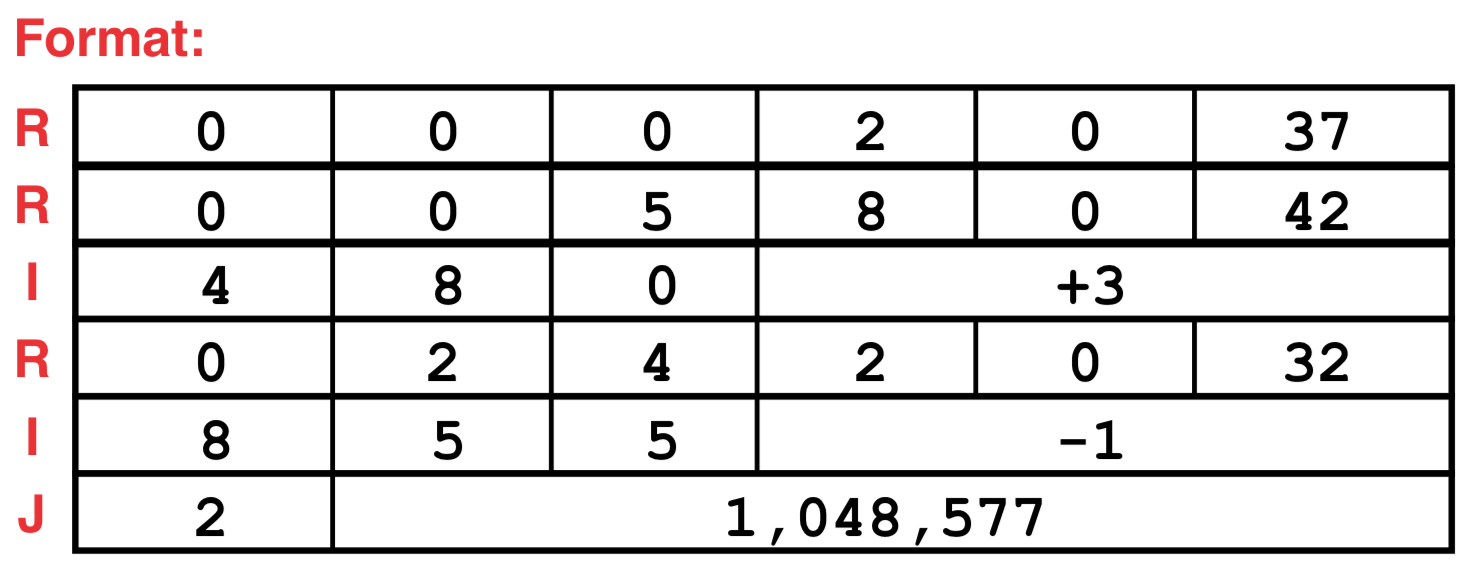




1. Convert to binary

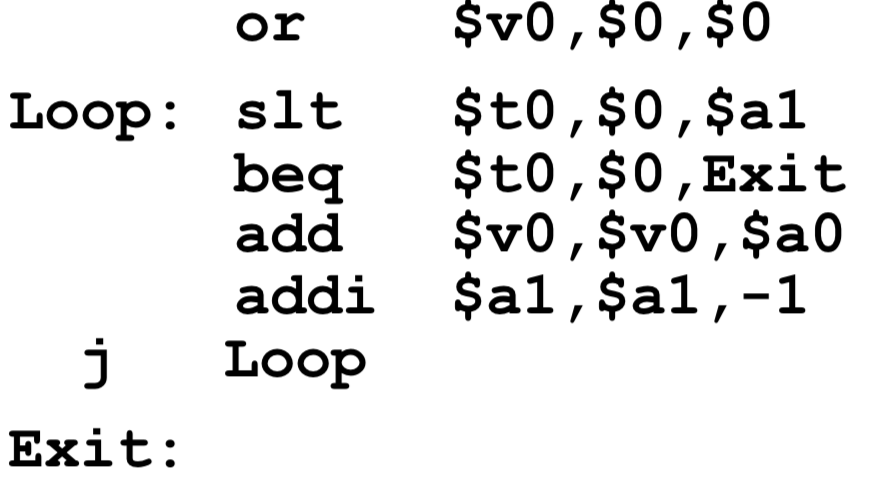
2. Identify opcode and format

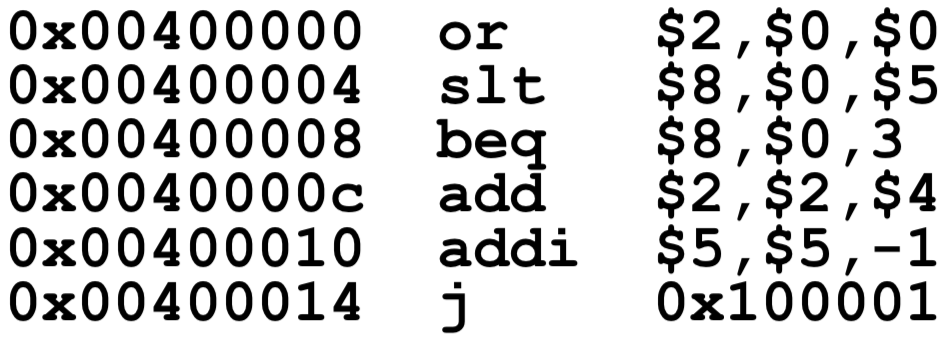
3. Separation of fields



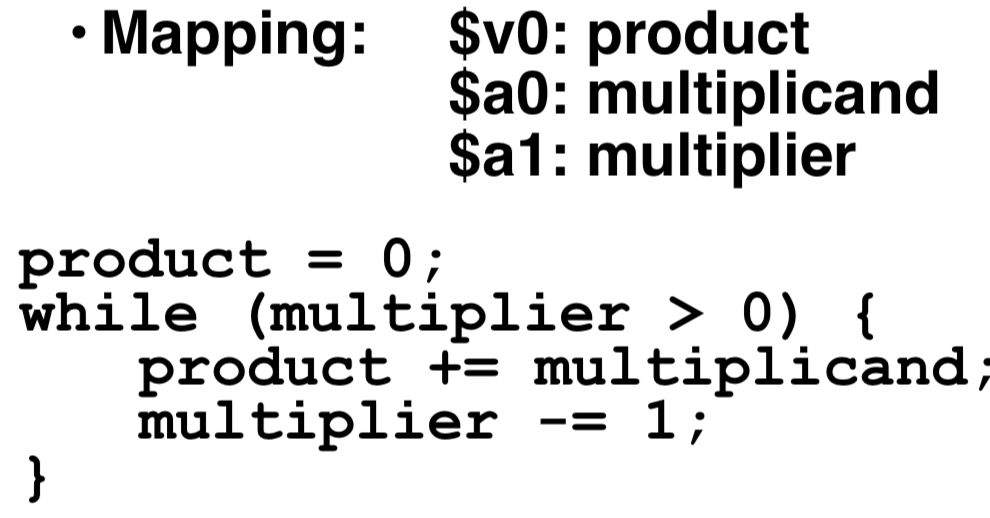
4. Translate to MIPS assembly instructions

5. Fix the branch/jump and add labels





6. Translate to C

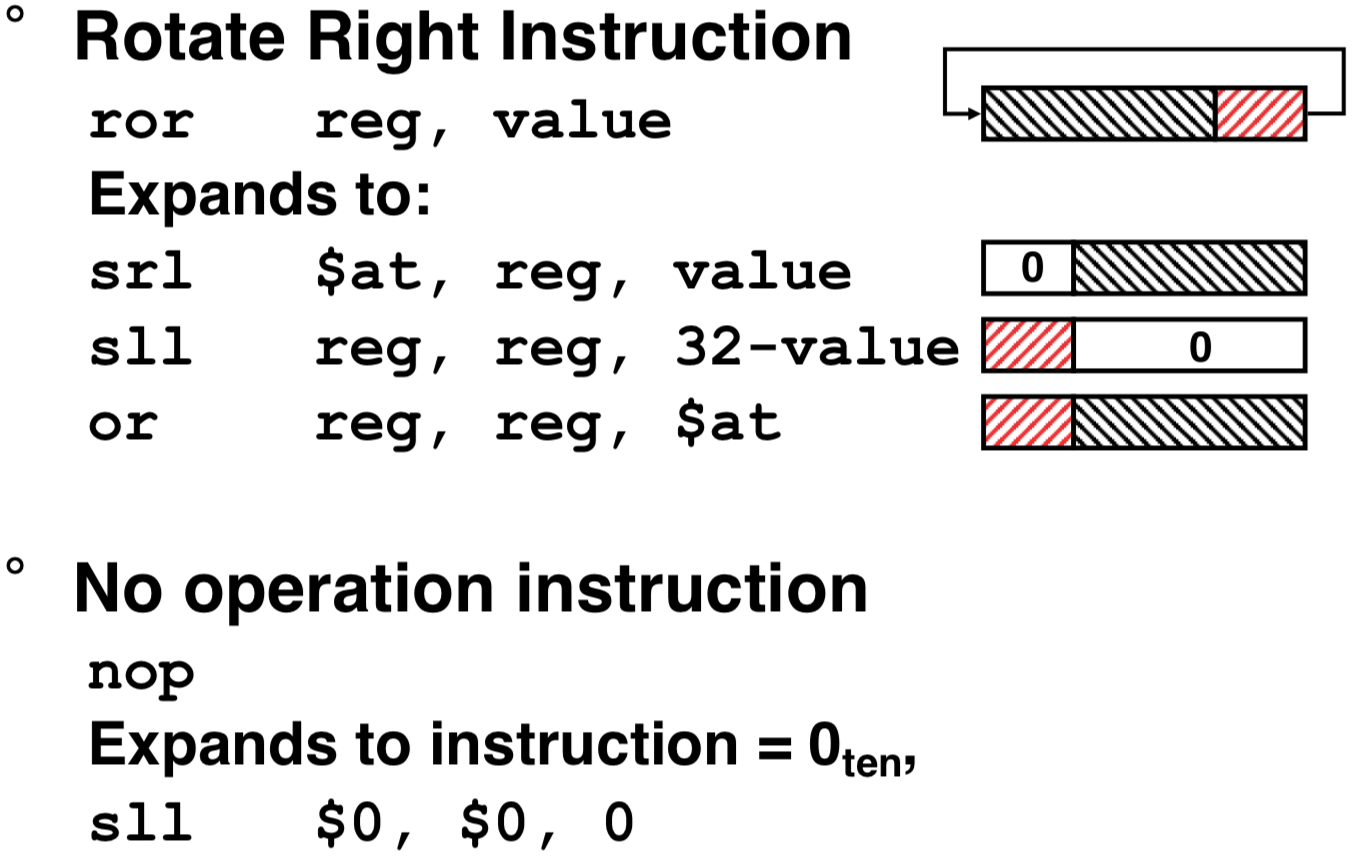
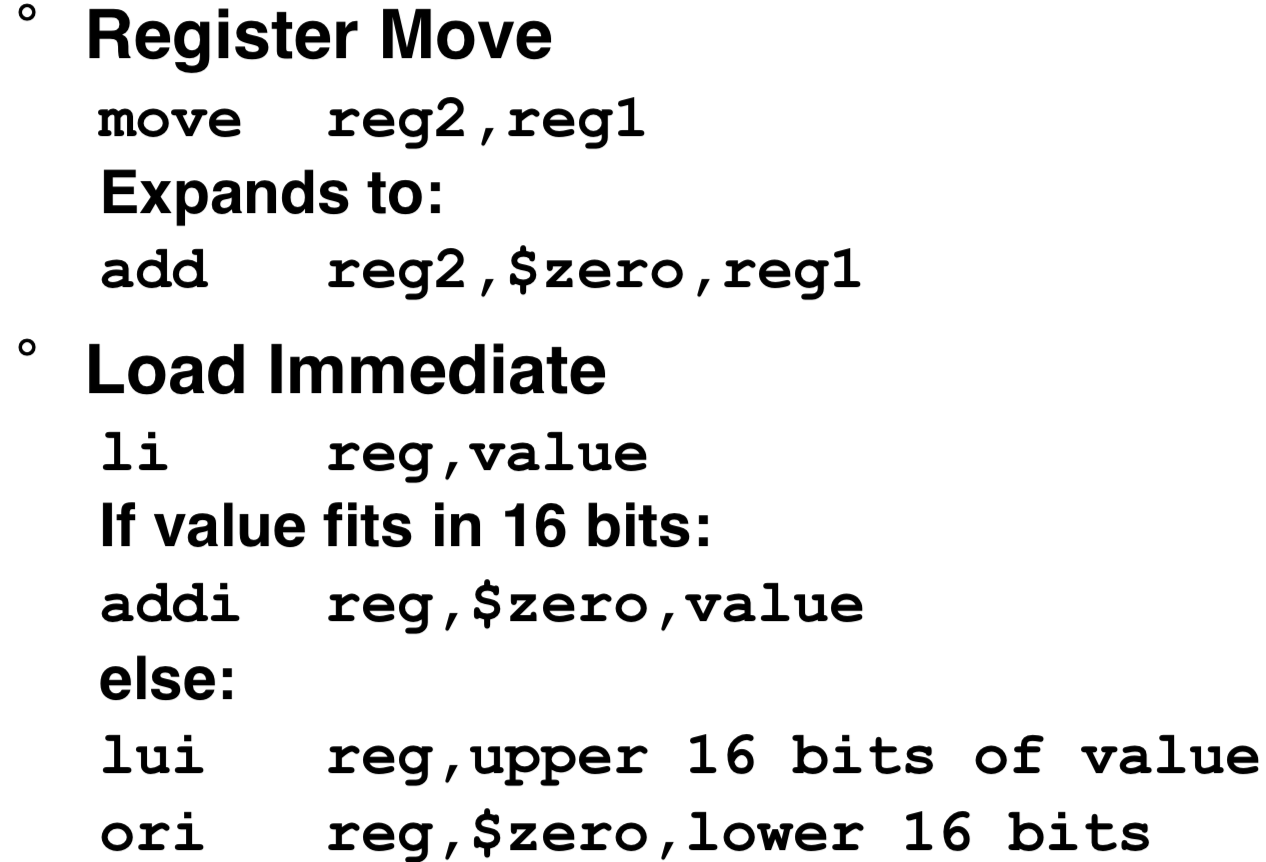


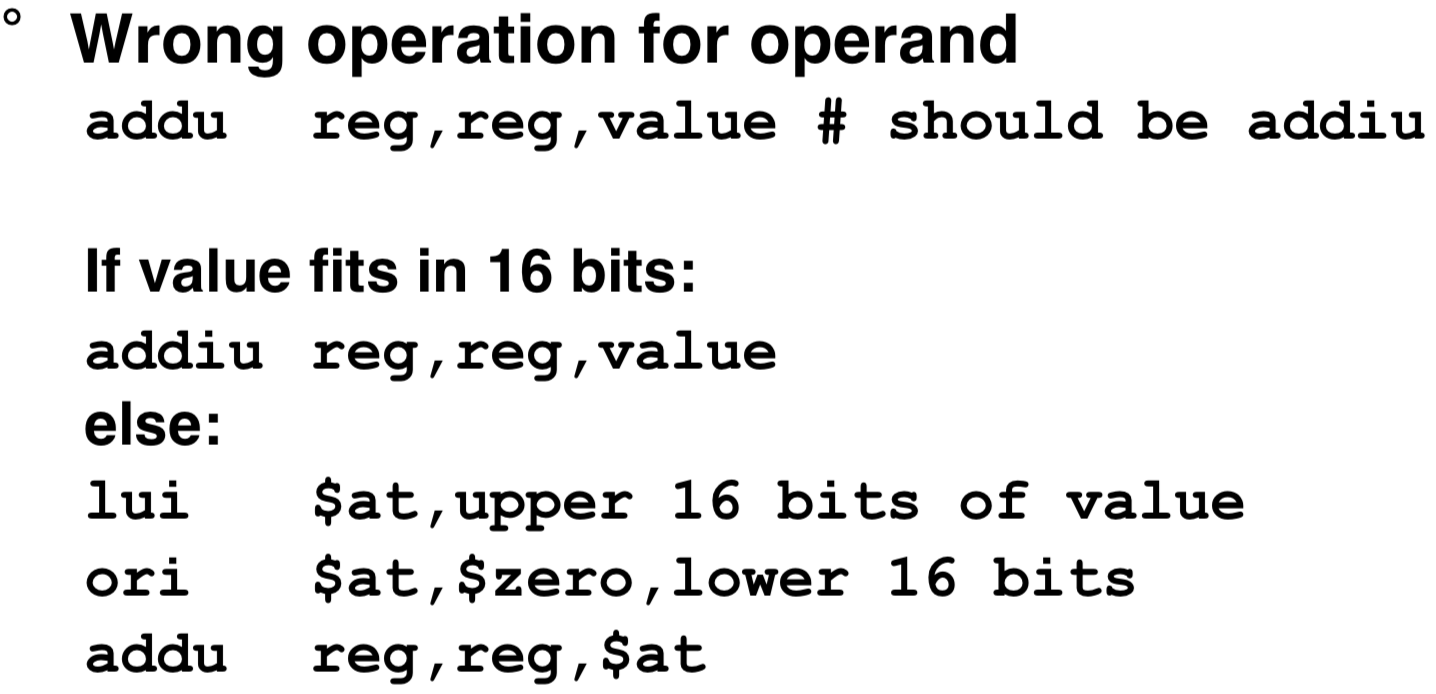
**“True” Assembly Language (TAL) v. “MIPS” Assembly Language (MAL)**

***Pseudoinstructions*: A MIPS instruction that doesn’t turn directly into a machine language instruction. They’re broken up by the assembler into several “real” MIPS instructions.**

When breaking up a pseudo instruction, the assembler may need to use an extra register. If it uses any regular register, it’ll overwrite whatever the program has put into it.

🡪 Reserve a register **($1, called $at for “assembler temporary”**) that the assembler will use when breaking up pseudoinstructions. Since the assembler may use this at any time, it’s not safe to code with it.





***MAL (MIPS Assembly Language):* the set of instructions that a programmer may use to code in MIPS; this includes pseudoinstructions**

***TAL (True Assembly Language):* the set of instructions that can actually get translated into a single machine language instruction (32-bit binary string)**

A program must be converted from MAL into TAL before it can be translated into 1s and 0s.

