## Instruction encoding

## - The ISA defines

- The format of an instruction (syntax)
- The meaning of the instruction (semantics)
- Format = Encoding
- Each instruction format has various fields
- Opcode field gives the semantics (Add, Load etc ...)
- Operand fields (rs,rt,rd,immed) say where to find inputs (registers, constants) and where to store the output
/3/2004
CSE378 Instr. encoding.
1


## MIPS Instruction encoding

- MIPS $=$ RISC hence
- Few (3+) instruction formats
- R in RISC also stands for "Regular"
- All instructions of the same length (32-bits = 4 bytes)
- Formats are consistent with each other
- Opcode always at the same place (6 most significant bits)
- rd and rs always at the same place
- immed always at the same place etc.

4/3/2004
CSE378 Instr. encoding.
2

## I-type (immediate) format

- An instruction with an immediate constant has the SPIM form:

Addi

## Operands <br> \$4,\$7,78

- Encoding of the 32 bits:
- Opcode is 6 bits
- Since we have 32 registers, each register "name" is 5 bits
- This leaves 16 bits for the immediate constant

4/3/2004
CSE378 Instr. encoding.
3

## Sign extension

- Internally the ALU (adder) deals with 32-bit numbers
- What happens to the 16 -bit constant?
- Extended to 32 bits
- If the Opcode says "unsigned" (e.g., Addiu)
- Fill upper 16 bits with 0's
- If the Opcode says "signed" (e.g., Addi)
- Fill upper 16 bits with the msb of the 16 bit constant
- i.e. fill with 0 's if the number is positive
- i.e. fill with 1's if the number is negative

4/3/2004
CSE378 Instr. encoding.
5

I-type format example


In hex: 21840021

4/3/2004
CSE378 Instr. encoding.

## R-type (register) format

- Arithmetic, Logical, and Compare instructions require encoding 3 registers.
- Opcode ( 6 bits) +3 registers ( $5 \times 3=15$ bits) $\Rightarrow 32-21=11$ "free" bits
- Use 6 of these bits to expand the Opcode
- Use 5 for the "shift" amount in shift instructions


4/3/2000
CSE378 Instr. encoding.


## Load and Store instructions

- MIPS $=$ RISC $=$ Load-Store architecture
- Load: brings data from memory to a register
- Store: brings data back to memory from a register
- Each load-store instruction must specify
- The unit of info to be transferred (byte, word etc. ) through the Opcode
- The address in memory
- A memory address is a 32-bit byte address
- An instruction has only 32 bits so ....

CSE378 Instr. encoding.

## Addressing in Load/Store instructions

- The address will be the sum
- of a base register (register rs)
- a 16-bit offset (or displacement) which will be in the immed field and is added (as a signed number) to the contents of the base register
- Thus, one can address any byte within $\pm 32 \mathrm{~KB}$ of the address pointed to by the contents of the base register.

4/3/2004
CSE378 Instr. encoding.
9


## Examples of load-store instructions

- Load word from memory:

$$
\text { Lw rt,rs,offset } \quad \text { \#rt }=\text { Memory[rs }+ \text { offset }]
$$

- Store word to memory:
Sw rt,rs,offset \#Memory[rs+offset]=rt
- For bytes (or half-words) only the lower byte (or halfword) of a register is addressable
- For load you need to specify if it is sign-extended or not Lb rt,rs,offset \#rt =sign-ext( Memory[rs+offset]) Lbu rt,rs,offset \#rt =zero-ext( Memory[rs+offset]) Sb rt,rs,offset \#Memory[rs+offset]= least signif. \#byte of rt

4/3/2004
CSE378 Instr. encoding.
10

## Loading small constants in a register

- If the constant is small (i.e., can be encoded in 16 bits) use the immediate format with Li (Load immediate)
Li
\$14,8
$\# \$ 14=8$
- But, there is no opcode for Li !
- Li is a pseudoinstruction
- It's there to help you
- SPIM will recognize it and transform it into Addi (with signextension) or Ori (zero extended)
Addi $\quad \$ 14, \$ 0,8 \quad \# \$ 14=\$ 0+8$

4/3/2004
CSE378 Instr. encoding.
12

## Loading large constants in a register

- If the constant does not fit in 16 bits (e.g., an address)
- Use a two-step process
- Lui (load upper immediate) to load the upper 16 bits; it will zero out automatically the lower 16 bits
- Use Ori for the lower 16 bits (but not Li, why?)
- Example: Load the constant 0x1B234567 in register \$t0 Lui $\$ 0,0 \times 1$ B23 $\quad$ \#note the use of hex constants
Ori Ori $\$ \mathrm{t} 0, \$ \mathrm{t} 0,0 \times 4567$

4/3/2004
CSE378 Instr. encoding.
13

How to address memory in assembly language

- Problem: how do I put the base address in the right register and how do I compute the offset
- Method 1 (most recommended). Let the assembler do it!

|  | .data | \#define data section |
| :--- | :--- | :--- |
| xyz: | .word 1 | \#reserve room for 1 word at address xyz |


| ........ |  |
| :--- | :--- |
| \#more data |  |
| text | \#define program section |

..... \#dexine program section
Iw \$5, xyz \# load contents of word at add. xyz in \$5

- In fact the assembler generates:

Lw $\quad \$ 5$, offset $(\$ \mathrm{gp}) \quad \# \$ \mathrm{gp}$ is register 28
4/3/2004 CSE378 Instr. encoding.

14

## Generating addresses

- Method 2. Use the pseudo-instruction La (Load address)

La \$6,xyz \#\$6 contains address of xyz
Lw \$5,0(\$6) \#\$5 contains the contents of xyz

- La is in fact Lui followed by Ori
- This method can be useful to traverse an array after loading the base address in a register
- Method 3
- If you know the address (i.e. a constant) use Li or Lui + Ori

[^0]CSE378 Instr. encoding.
15


[^0]:    4/3/2004

