



Csci 136 Computer Architecture II – MIPS Instruction Set Architecture

Xiuzhen Cheng
cheng@gwu.edu



Outline

- Variables vs. Registers
- MIPS Instruction Set
- MIPS Addressing Modes
- MIPS Instruction Format
 - R-Type
 - I-Type
 - J-Type
- Encoding Assembly Code



Homework

- Homework Assignment #2: due Feb. 01, before class
- Readings for this week:
 - Sections 2.2-2.6, 2.8-2.9, 2.19
- Homework #2 questions:
 - Problems 2.30-2.31, 2.33-2.34, 2.36-2.37, 2.39



Assembly Language

- Assembly language vs. higher-level language
 - Few, simple types of data and control
 - Does not specify variable type
 - Control flow is implemented with **goto/jump**
 - Assembly language programming is more difficult and error-prone, it is machine-specific; it is longer
- Assembly language vs. machine language
 - Symbolic representation
- When assembly language is needed
 - Speed and size, (eg. Embedded computer)
 - Time-critical parts of a program
 - Specialized instructions



Instructions

- Basic job of a CPU: execute lots of *instructions*.
- Instructions are the primitive operations that the CPU may execute.
- Different CPUs implement different sets of instructions. The set of instructions a particular CPU implements is an *Instruction Set Architecture (ISA)*.
 - Examples: Intel 80x86 (Pentium 4), IBM/Motorola PowerPC (Macintosh), MIPS, Intel IA64, ...



Instruction Set Architectures

- Early trend was to add more and more instructions to new CPUs to do elaborate operations
 - VAX architecture had an instruction to multiply polynomials!
- RISC philosophy (Cocke IBM, Patterson, Hennessy, 1980s) –
Reduced Instruction Set Computing
 - Keep the instruction set small and simple, makes it easier to build faster hardware.
 - Let software do complicated operations by composing simpler ones.



MIPS Architecture

- MIPS – semiconductor company that built one of the first commercial RISC architectures
- We will study the MIPS architecture in some detail in this class
- Why MIPS instead of Intel 80x86?
 - MIPS is simple, elegant. Don't want to get bogged down in gritty details.
 - MIPS widely used in embedded apps, x86 little used in embedded, and more embedded computers than PCs



Most HP LaserJet workgroup printers are driven by MIPS-based 64-bit processors.



Assembly Variables: Registers (1/4)

- Unlike HLL like C or Java, assembly cannot use variables
 - Why not? Keep Hardware Simple
- Assembly Operands are registers
 - 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 (faster than 1 billionth of a second)



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 - \$23 → \$s0 - \$s7
(correspond to C variables)
 - \$8 - \$15 → \$t0 - \$t7
(correspond to temporary variables)
 - Later will explain other 16 register names
- In general, use names to make your code more readable



C, Java variables vs. registers

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


```
int fahr, celsius;
char a, b, c, d, e;
```
- Each variable can ONLY represent a value of the type it was declared as (cannot mix and match `int` and `char` variables).
- In Assembly Language, the registers have no type; operation determines how register contents are treated



Comments in Assembly

- ◆ Another way to make your code more readable: comments!
- ◆ Hash (#) is used for MIPS comments
 - anything from hash mark to end of line is a comment and will be ignored
- ◆ Note: Different from C.
 - C comments have format `/* comment */` so they can span many lines



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), each line of assembly code contains at most 1 instruction
- ◆ Instructions are related to operations (=, +, -, *, /) in C or Java
- ◆ Ok, enough already...gimme my MIPS!



MIPS Addition and Subtraction (1/4)

- ◆ 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 of Integers (2/4)

- ◆ Addition in Assembly
 - Example: `add $s0,$s1,$s2` (in MIPS)
 - Equivalent to: `a = b + c` (in C)
 - where MIPS registers \$s0, \$s1, \$s2 are associated with C variables a, b, c
- ◆ Subtraction in Assembly
 - Example: `sub $s3,$s4,$s5` (in MIPS)
 - Equivalent to: `d = e - f` (in C)
 - where MIPS registers \$s3, \$s4, \$s5 are associated with C variables d, e, f



Addition and Subtraction of Integers (3/4)

- ◆ How do the following C statement?


```
a = b + c + d - e;
```
- ◆ Break into multiple instructions


```
add $t0, $s1, $s2 # temp = b + c
add $t0, $t0, $s3 # temp = temp + d
sub $s0, $t0, $s4 # a = temp - 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 (comments)



Addition and Subtraction of Integers (4/4)

- ◆ How do we do this?


```
f = (g + h) - (i + j);
```
- ◆ Use intermediate temporary register


```
add $t0,$s1,$s2 # temp = g + h
add $t1,$s3,$s4 # temp = i + j
sub $s0,$t0,$t1 # f=(g+h)-(i+j)
```



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; eg


```
add $s0,$s1,$zero (in MIPS)
f = g (in C)
```

 where MIPS registers \$s0, \$s1 are associated with C variables f, g
- defined in hardware, so an instruction


```
add $zero,$zero,$s0
```

 will not do anything!



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



Immediates

- There is no Subtract Immediate in MIPS: Why?
- Limit types of operations that can be done to absolute minimum
 - if an operation can be decomposed into a simpler operation, don't include it
 - addi ..., -X = subi ..., X => so no subi
- addi \$s0,\$s1,-10 (in MIPS)


```
f = g - 10 (in C)
```

 where MIPS registers \$s0, \$s1 are associated with C variables f, g



"And in Conclusion..."

- In MIPS Assembly Language:
 - Registers replace C variables
 - One Instruction (simple operation) per line
 - Simpler is Better
 - Smaller is Faster
- "There are no types in MIPS"
 - "There are no types associated with variables – the types are associated with the instructions". Said another way:
 - "In Assembly Language, the registers have no type; the operation determines how register contents are treated"
- New Instructions:


```
add, addi, sub
```
- New Registers:


```
C Variables: $s0 - $s7
Temporary Variables: $t0 - $t9
Zero: $zero
```



Assembly Operands: Memory

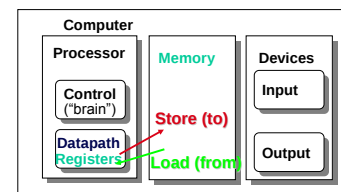
- C variables map onto registers; what about large data structures like arrays?
- 1 of 5 components of a computer: memory contains such data structures
- But MIPS arithmetic instructions only operate on registers, never directly on memory.
- Data transfer instructions transfer data between registers and memory:
 - Memory to register
 - Register to memory



Anatomy: 5 components of any Computer



Registers are in the datapath of the processor; if operands are in memory, we must transfer them to the processor to operate on them, and then transfer back to memory when done.



These are "data transfer" instructions...



Data Transfer: Memory to Reg (1/4)

- To transfer a word of data, we need to specify two things:
 - **Register:** specify this by # (\$0 - \$31) or symbolic name (\$s0, ..., \$t0, ...)
 - **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.
- **Remember:** “Load FROM memory”



Data Transfer: Memory to Reg (2/4)

- To specify a memory address to copy from, specify two things:
 - A register containing 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 name
 - 2) register that will receive value
 - 3) numerical offset in bytes
 - 4) register containing pointer to memory
- **MIPS 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 reg points to beginning of array or structure



Data Transfer: Reg to Memory

- **Also want to store from register into memory**
 - Store instruction syntax is identical to Load's
 - **MIPS 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 that memory address
- **Remember:** “Store INTO memory”



Pointers v. 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), and so on
- If you write `add $t2, $t1, $t0` then \$t0 and \$t1 better contain values
- If you write `lw $t2, 0($t0)` then \$t0 better contain a pointer
- Don't mix these up!



Addressing: Byte vs. word

- Every word in memory has an **address**, 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 "address" of a word
- Computers needed to access 8-bit **bytes** as well as words (4 bytes/word)
- Today machines address memory as bytes, (i.e., "Byte Addressed") hence 32-bit (4 byte) word addresses differ by 4
 - Memory[0], Memory[4], Memory[8], ...



Compilation with Memory

- What offset in `lw` to select `A[5]` in C?
- $4 \times 5 = 20$ to select `A[5]`: byte v. word
- Compile by hand using registers:
 - $g = h + A[5];$
 - $g: \$s1, h: \$s2, \$s3: \text{base address of } A$
- 1st transfer from memory to register:
 - `lw $t0, 20($s3) # $t0 gets A[5]`
 - Add 20 to $\$s3$ to select `A[5]`, put into $\$t0$
- Next add it to h and place in g
 - `add $s1, $s2, $t0 # $s1 = h+A[5]`



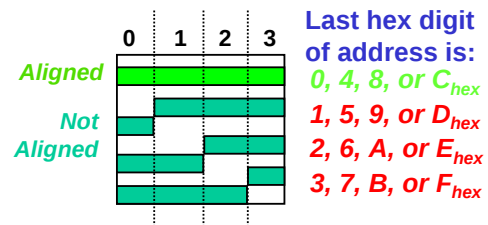
Notes about Memory

- Pitfall: Forgetting that sequential word addresses in machines with byte addressing do not differ by 1.
 - Many assembly language programmers have 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 byte addresses that are multiples of 4 bytes



- Called **Alignment**: 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
 - Less common in 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



So Far...

- All instructions so far only manipulate data...we've built a calculator.
- In order to build a computer, we need ability to make decisions...
- C (and MIPS) provide **labels** to support "goto" jumps to places in code.
 - C: Horrible style; MIPS: Necessary!
- Heads up: pull out some papers and pens, you'll do an in-class exercise!



C Decisions: if Statements

2 kinds of if statements in C

- if (condition) clause
- if (condition) clause1 else clause2

Rearrange 2nd if into following:

```
if (condition) goto L1;
    clause2;
    goto L2;
L1: clause1;
L2:
```

- Not as elegant as if-else, but same meaning



MIPS Decision Instructions

Decision instruction in MIPS:

- beq register1, register2, L1
- beq is "Branch if (registers are) equal"
- Same meaning as (using C):
if (register1==register2) goto L1

Complementary MIPS decision instruction

- bne register1, register2, L1
- bne is "Branch if (registers are) not equal"
- Same meaning as (using C):
if (register1!=register2) goto L1

- Called **conditional branches**



MIPS Goto Instruction

- In addition to conditional branches, MIPS has an **unconditional branch**:

```
j label
```

- Called a Jump Instruction: jump (or branch) directly to the given label without needing to satisfy any condition

- Same meaning as (using C):
goto label

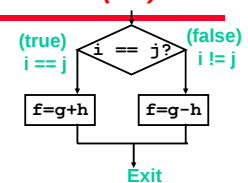
- Technically, it's the same as:
beq \$0, \$0, label
since it always satisfies the condition.



Compiling C if into MIPS (1/2)

Compile by hand

```
if (i == j) f=g+h;
else f=g-h;
```



Use this mapping:

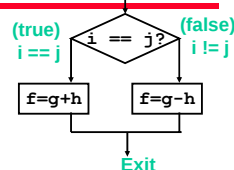
```
f: $s0
g: $s1
h: $s2
i: $s3
j: $s4
```



Compiling C if into MIPS (2/2)

Compile by hand

```
if (i == j) f=g+h;
else f=g-h;
```



Final compiled MIPS code:

```
beq $s3,$s4,True # branch i==j
sub $s0,$s1,$s2 # f=g-h(false)
j Fin # goto Fin
True: add $s0,$s1,$s2 # f=g+h (true)
Fin:
```

Note: Compiler automatically creates labels to handle decisions (branches).
Generally not found in HLL code.



"And in Conclusion..."

- 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).
- A Decision allows us to decide what to execute at run-time rather than compile-time.
- C Decisions are made using **conditional statements** within **if**, **while**, **do while**, **for**.
- MIPS Decision making instructions are the **conditional branches**: **beq** and **bne**.
- New Instructions:
lw, sw, beq, bne, j



Loading, Storing bytes 1/2

- In addition to word data transfers (`lw`, `sw`), MIPS has byte data transfers:
- load byte: `lb`
- store byte: `sb`
- same format as `lw`, `sw`



Loading, Storing bytes 2/2

- What do with other 24 bits in the 32 bit register?
 - `lb`: sign extends to fill upper 24 bits



- Normally don't want to sign extend chars
- MIPS instruction that doesn't sign extend when loading bytes:

load byte unsigned: `lbu`



Overflow in Arithmetic (1/2)

- Reminder: Overflow occurs when there is a mistake in arithmetic due to the limited precision in computers.

- Example (4-bit unsigned numbers):

```

+15      1111
+3        0011
-----
+18      10010
  
```

- But we don't have room for 5-bit solution, so the solution would be 0010, which is +2, and wrong.



Overflow in Arithmetic (2/2)

- Some languages detect overflow (Ada), some don't (C)
- MIPS solution is 2 kinds of arithmetic instructions to recognize 2 choices:
 - add (`add`), add immediate (`addi`) and subtract (`sub`) *cause overflow to be detected*
 - add unsigned (`addu`), add immediate unsigned (`addiu`) and subtract unsigned (`subu`) do *not* cause overflow detection
- Compiler selects appropriate arithmetic
 - MIPS C compilers produce `addu`, `addiu`, `subu`



Loops in C/Assembly (1/3)

- Simple loop in C; `A[]` is an array of ints

```

do {
    g = g + A[i];
    i = i + j;
} while (i != h);
  
```

- Rewrite this as:

```

Loop:  g = g + A[i];
       i = i + j;
       if (i != h) goto Loop;
  
```

- Use this mapping:

```

g,    h,    i,    j,    base of A
$s1,  $s2,  $s3,  $s4,  $s5
  
```



Loops in C/Assembly (2/3)

- Final compiled MIPS code:

```

Loop:  sll $t1,$s3,2    #$t1= 4*I
       add $t1,$t1,$s5  #$t1=addr A
       lw  $t1,0($t1)   #$t1=A[i]
       add $s1,$s1,$t1  #g=g+A[i]
       add $s3,$s3,$s4  #i=i+j
       bne $s3,$s2,Loop# goto Loop
                          # if i!=h
  
```

- Original code:

```

Loop:  g = g + A[i];
       i = i + j;
       if (i != h) goto Loop;
  
```



Loops in C/Assembly (3/3)

- There are three types of loops in C:
 - while
 - do... while
 - for
- Each can be rewritten as either of the other two, so the method used in the previous example can be applied to while and for loops as well.
- Key Concept:** Though there are multiple ways of writing a loop in MIPS, the key to decision making is conditional branch



Inequalities in MIPS (1/3)

- Until now, we've only tested equalities (== and != in C). General programs need to test < and > as well.
- Create a MIPS Inequality Instruction:
 - "Set on Less Than"
 - Syntax: `slt reg1, reg2, reg3`
 - Meaning:


```
if (reg2 < reg3)
    reg1 = 1;  reg1 = (reg2 < reg3);
else reg1 = 0;
```
 - In computerese, "set" means "set to 1", "reset" means "set to 0".



Inequalities in MIPS (2/3)

- How do we use this? Compile by hand:


```
if (g < h) goto Less; #g:$s0, h:$s1
```
- Answer: compiled MIPS code...


```
slt $t0,$s0,$s1 # $t0 = 1 if g<h
bne $t0,$0,Less # goto Less
                # if $t0!=0
                # (if (g<h)) Less:
```
- Branch if \$t0 != 0 → (g < h)
- Register \$0 always contains the value 0, so bne and beq often use it for comparison after an slt instruction.
- A slt → bne pair means if (... < ...) goto...



Inequalities in MIPS (3/3)

- Now, we can implement <, but how do we implement >, ≤ and ≥?
- We could add 3 more instructions, but:
 - MIPS goal: **Simpler is Better**
- Can we implement ≤ in one or more instructions using just slt and the branches?
- What about >?
- What about ≥?



Immediates in Inequalities

- There is also an immediate version of slt to test against constants: `slti`
 - Helpful in for loops


```
if (g >= 1) goto Loop
```

C Loop: . . .

MIPS

```
slti $t0,$s0,1    # $t0 = 1 if
                  # $s0<1 (g<1)
beq  $t0,$0,Loop  # goto Loop
                  # if $t0==0
                  # (if (g>=1))
```

A slt → beq pair means if (... ≥ ...) goto...



What about unsigned numbers?

- Also **unsigned** inequality instructions:


```
sltu, sltiu
```
- ...which sets result to 1 or 0 depending on unsigned comparisons
- What is value of \$t0, \$t1?


```
($s0 = FFFF FFFAhex, $s1 = 0000 FFFAhex)
    slt $t0, $s0, $s1
    sltu $t1, $s0, $s1
```

MIPS Signed vs. Unsigned – diff meanings!

- ◆ MIPS Signed v. Unsigned is an “overloaded” term

- Do/Don't sign extend (lb, lbu)
- Don't overflow (addu, addiu, subu, multu, divu)
- Do signed/unsigned compare (slt, slti/sltu, sltiu)

Example: The C Switch Statement (1/3)

- ◆ Choose among four alternatives depending on whether *k* has the value 0, 1, 2 or 3. Compile this C code:

```
switch (k) {  
  case 0: f=i+j; break; /* k=0 */  
  case 1: f=g+h; break; /* k=1 */  
  case 2: f=g-h; break; /* k=2 */  
  case 3: f=i-j; break; /* k=3 */  
}
```

Example: The C Switch Statement (2/3)

- ◆ This is complicated, so **simplify**.
- ◆ Rewrite it as a chain of if-else statements, which we already know how to compile:

```
if(k==0) f=i+j;  
else if(k==1) f=g+h;  
else if(k==2) f=g-h;  
else if(k==3) f=i-j;
```

- ◆ Use this mapping:

```
f:$s0, g:$s1, h:$s2,  
i:$s3, j:$s4, k:$s5
```

Example: The C Switch Statement (3/3)

- ◆ Final compiled MIPS code:

```
bne $s5,$0,L1 # branch k!=0  
add $s0,$s3,$s4 #k==0 so f=i+j  
j Exit # end of case so Exit  
L1: addi $t0,$s5,-1 # $t0=k-1  
bne $t0,$0,L2 # branch k!=1  
add $s0,$s1,$s2 #k==1 so f=g+h  
j Exit # end of case so Exit  
L2: addi $t0,$s5,-2 # $t0=k-2  
bne $t0,$0,L3 # branch k!=2  
sub $s0,$s1,$s2 #k==2 so f=g-h  
j Exit # end of case so Exit  
L3: addi $t0,$s5,-3 # $t0=k-3  
bne $t0,$0,Exit # branch k!=3  
sub $s0,$s3,$s4 #k==3 so f=i-j  
Exit:
```

“And in conclusion...”

- ◆ In order to help the **conditional branches** make decisions concerning inequalities, we introduce a single instruction: “Set on Less Than” called **slt**, **slti**, **sltu**, **sltiu**
- ◆ One can store and load (signed and unsigned) **bytes** as well as words
- ◆ Unsigned add/sub don't cause overflow
- ◆ New MIPS Instructions:
 sll, srl
 slt, slti, sltu, sltiu
 addu, addiu, subu

Bitwise Operations

- ◆ Up until now, we've done arithmetic (add, sub, addi), memory access (lw and sw), and branches and jumps.
- ◆ All of these instructions view contents of register as a single quantity (such as a signed or unsigned integer)
- ◆ **New Perspective:** View register as 32 raw bits rather than as a single 32-bit number
- ◆ Since registers are composed of 32 bits, we may want to access individual bits (or groups of bits) rather than the whole.
- ◆ Introduce two new classes of instructions:
 - Logical & Shift Ops



Logical Operators (1/3)

♦ **Two basic logical operators:**

- AND: outputs 1 only if **both** inputs are 1
- OR: outputs 1 if **at least one** input is 1

♦ **Truth Table: standard table listing all possible combinations of inputs and resultant output for each. E.g.,**

A	B	A AND B	A OR B
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1



Logical Operators (2/3)

♦ **Logical Instruction Syntax:**

- 1 2,3,4
- where
 - 1) operation name
 - 2) register that will receive value
 - 3) first operand (register)
 - 4) second operand (register) or immediate (numerical constant)

♦ **In general, can define them to accept > 2 inputs, but in the case of MIPS assembly, these accept exactly 2 inputs and produce 1 output**

- Again, rigid syntax, simpler hardware



Logical Operators (3/3)

♦ **Instruction Names:**

- and, or: Both of these expect the third argument to be a register
- andi, ori: Both of these expect the third argument to be an immediate

♦ **MIPS Logical Operators are all bitwise, meaning that bit 0 of the output is produced by the respective bit 0's of the inputs, bit 1 by the bit 1's, etc.**

- C: Bitwise AND is & (e.g., $z = x \& y;$)
- C: Bitwise OR is | (e.g., $z = x | y;$)



Uses for Logical Operators (1/3)

♦ **Note that anding a bit with 0 produces a 0 at the output while anding a bit with 1 produces the original bit.**

♦ **This can be used to create a mask.**

▪ Example:

```
1011 0110 1010 0100 0011 1101 1001 1010
0000 0000 0000 0000 0000 1111 1111 1111
```

▪ The result of anding these:

```
0000 0000 0000 0000 0000 1101 1001 1010
```

mask:

mask last 12 bits



Uses for Logical Operators (2/3)

♦ **The second bitstring in the example is called a mask. It is used to isolate the rightmost 12 bits of the first bitstring by masking out the rest of the string (e.g. setting it to all 0s).**

♦ **Thus, the and operator can be used to set certain portions of a bitstring to 0s, while leaving the rest alone.**

- In particular, if the first bitstring in the above example were in \$t0, then the following instruction would mask it:

```
andi    $t0,$t0,0xFFF
```



Uses for Logical Operators (3/3)

♦ **Similarly, note that oring a bit with 1 produces a 1 at the output while oring a bit with 0 produces the original bit.**

♦ **This can be used to force certain bits of a string to 1s.**

- For example, if \$t0 contains 0x12345678, then after this instruction:

```
ori     $t0, $t0, 0xFFFF
```

- ... \$t0 contains 0x1234FFFF (e.g. the high-order 16 bits are untouched, while the low-order 16 bits are forced to 1s).



Shift Instructions (1/4)

- ♦ Move (shift) all the bits in a word to the left or right by a number of bits.

- Example: shift right by 8 bits

0001 0010 0011 0100 0101 0110 0111 1000

0000 0000 0001 0010 0011 0100 0101 0110

- Example: shift left by 8 bits

0001 0010 0011 0100 0101 0110 0111 1000

0011 0100 0101 0110 0111 1000 0000 0000



Shift Instructions (2/4)

- ♦ Shift Instruction Syntax:

1 2,3,4

- where

- 1) operation name
- 2) register that will receive value
- 3) first operand (register)
- 4) shift amount (constant < 32)

- ♦ MIPS shift instructions:

1. `sll` (shift left logical): shifts left and fills emptied bits with 0s
2. `srl` (shift right logical): shifts right and fills emptied bits with 0s
3. `sra` (shift right arithmetic): shifts right and fills emptied bits by sign extending



Shift Instructions (3/4)

- ♦ Example: shift right arith by 8 bits

0001 0010 0011 0100 0101 0110 0111 1000

0000 0000 0001 0010 0011 0100 0101 0110

- ♦ Example: shift right arith by 8 bits

1001 0010 0011 0100 0101 0110 0111 1000

1111 1111 1001 0010 0011 0100 0101 0110



Shift Instructions (4/4)

- ♦ Since shifting may be faster than multiplication, a good compiler usually notices when C code multiplies by a power of 2 and compiles it to a shift instruction:

a *= 8; (in C)

would compile to:

`sll $s0,$s0,3` (in MIPS)

- ♦ Likewise, shift right to divide by powers of 2

- remember to use `sra`



"And in Conclusion..."

- ♦ Logical and Shift Instructions

- Operate on bits individually, unlike arithmetic, which operate on entire word.
- Use to isolate fields, either by masking or by shifting back and forth.
- Use shift left logical, `sll`, for multiplication by powers of 2
- Use shift right arithmetic, `sra`, for division by powers of 2.

- ♦ New Instructions:

`and,andi,or,ori,sll,srl,sra`

- ♦ Question:

- `sll`: Does it signal overflow?
- Answer: Nope, the bits are "lost" over the left side!



What's Next? – Instruction Representation

- ♦ Big idea: stored program

- consequences of stored program

- ♦ Instructions as numbers

- ♦ Instruction encoding

- ♦ MIPS instruction format for Add instructions

- ♦ MIPS instruction format for Immediate, Data transfer instructions



Big Idea: Stored-Program Concept

- ◆ **Computers built on 2 key principles:**
 - 1) Instructions are represented as numbers.
 - 2) Therefore, entire programs can be stored in memory to be read or written just like numbers (data).
- ◆ **Simplifies SW/HW of computer systems:**
 - Memory technology for data also used for programs



Consequence #1: Everything Addressed

- ◆ **Since all instructions and data are stored in memory as numbers, everything has a memory address: instructions, data words**
 - both branches and jumps use these
- ◆ **C pointers are just memory addresses: they can point to anything in memory**
 - Unconstrained use of addresses can lead to nasty bugs; up to you in C; limits in Java
- ◆ **One register keeps address of instruction being executed: "Program Counter" (PC)**
 - Basically a pointer to memory: Intel calls it Instruction Address Pointer, a better name



Consequence #2: Binary Compatibility

- ◆ **Programs are distributed in binary form**
 - Programs bound to specific instruction set
 - Different version for Macintoshes and PCs
- ◆ **New machines want to run old programs ("binaries") as well as programs compiled to new instructions**
- ◆ **Leads to instruction set evolving over time**
- ◆ **Selection of Intel 8086 in 1981 for 1st IBM PC is major reason latest PCs still use 80x86 instruction set (Pentium 4); could still run program from 1981 PC today**
- ◆ **By treating the instructions in the same way as the data, a stored-program machine can easily change the instructions. In other words the machine is reprogrammable. One important motivation for such a facility was the need for a program to increment or otherwise modify the address portion of instructions.**



Instructions as Numbers (1/2)

- ◆ **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.
- ◆ **So how do we represent instructions?**
 - Remember: Computer only understands 1s and 0s, so "add \$t0, \$0, \$0" is meaningless.
 - MIPS wants simplicity: since data is in words, make instructions be words too



Instructions as Numbers (2/2)

- ◆ **One word is 32 bits, so divide instruction word into "fields".**
- ◆ **Each field tells computer something about instruction.**
- ◆ **We could define different fields for each instruction, but MIPS is based on simplicity, so define 3 basic types of instruction formats:**
 - R-format
 - I-format
 - J-format



Instruction Formats

- ◆ **I-format: used for instructions with immediates, `lw` and `sw` (since the offset counts as an immediate), and the branches (`beq` and `bne`),**
 - (but not the shift instructions; later)
- ◆ **J-format: used for `j` and `jal`**
- ◆ **R-format: used for all other instructions**
- ◆ **It will soon become clear why the instructions have been partitioned in this way.**



R-Format Instructions (1/5)

- Define "fields" of the following number of bits each:
 $6 + 5 + 5 + 5 + 5 + 6 = 32$

6	5	5	5	5	6
---	---	---	---	---	---

- For simplicity, each field has a name:

opcode	rs	rt	rd	shamt	funct
--------	----	----	----	-------	-------

- Important: On these slides and in book, each field is viewed as a 5- or 6-bit unsigned integer, not as part of a 32-bit integer.

- Consequence: 5-bit fields can represent any number 0-31, while 6-bit fields can represent any number 0-63.



R-Format Instructions (2/5)

- What do these field integer values tell us?

- opcode: partially specifies what instruction it is
 - Note: This number is equal to 0 for all R-Format instructions.
- funct: combined with opcode, this number exactly specifies the instruction
- Question: Why aren't opcode and funct a single 12-bit field?
 - Answer: We'll answer this later.



R-Format Instructions (3/5)

- More fields:

- rs (Source Register): *generally* used to specify register containing first operand
- rt (Target Register): *generally* used to specify register containing second operand (note that name is misleading)
- rd (Destination Register): *generally* used to specify register which will receive result of computation



R-Format Instructions (4/5)

- Notes about register fields:

- 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.
- The word "generally" was used because there are exceptions that we'll see later. E.g.,
 - mult and div have nothing important in the rd field since the dest registers are hi and lo
 - mflr and mflr have nothing important in the rs and rt fields since the source is determined by the instruction (p. 264 P&H)



R-Format Instructions (5/5)

- Final field:

- 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.

- For a detailed description of field usage for each instruction, see green insert in COD 3/e



R-Format Example (1/2)

- MIPS Instruction:

add \$8, \$9, \$10

opcode = 0 (look up in table in book)

funct = 32 (look up in table in book)

rd = 8 (destination)

rs = 9 (first operand)

rt = 10 (second operand)

shamt = 0 (not a shift)



R-Format Example (2/2)

♦ MIPS Instruction:

add \$8, \$9, \$10

Decimal number per field representation:

0	9	10	8	0	32
---	---	----	---	---	----

Binary number per field representation:

000000	01001	01010	01000	00000	100000
--------	-------	-------	-------	-------	--------

hex representation: 012A 4020_{hex}

decimal representation: 19,546,144_{ten}

- Called a [Machine Language Instruction](#)



I-Format Instructions (1/4)

♦ What about instructions with immediates?

- 5-bit field only represents numbers up to the value 31: immediates may be much larger than this
- Ideally, MIPS would have only one instruction format (for simplicity): unfortunately, we need to compromise

♦ Define new instruction format that is partially consistent with R-format:

- First notice that, if instruction has immediate, then it uses at most 2 registers.



I-Format Instructions (2/4)

♦ Define "fields" of the following number of bits each: 6 + 5 + 5 + 16 = 32 bits

6	5	5	16
---	---	---	----

♦ Again, each field has a name:

opcode	rs	rt	immediate
--------	----	----	-----------

- Key Concept:** Only one field is inconsistent with R-format. Most importantly, `opcode` is still in same location.



I-Format Instructions (3/4)

♦ What do these fields mean?

- `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 (this is why it's called the *target* register "rt")



I-Format Instructions (4/4)

♦ The Immediate Field:

- `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 2^{16} 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.
- We'll see what to do when the number is too big latter...



I-Format Example (1/2)

♦ MIPS Instruction:

`addi $21, $22, -50`

`opcode` = 8 (look up in table in book)

`rs` = 22 (register containing operand)

`rt` = 21 (target register)

`immediate` = -50 (by default, this is decimal)



I-Format Example (2/2)

♦MIPS Instruction:

addi \$21,\$22,-50

Decimal/field representation:

8	22	21	-50
---	----	----	-----

Binary/field representation:

001000	10110	10101	111111111001110
--------	-------	-------	-----------------

hexadecimal representation: 22D5 FFC_{hex}

decimal representation: 584,449,998_{ten}



In conclusion...

- ♦Simplifying MIPS: Define instructions to be same size as data word (one word) so that they can use the same memory (compiler can use lw and sw).
- ♦Computer actually stores programs as a series of these 32-bit numbers.
- ♦MIPS Machine Language Instruction: 32 bits representing a single instruction

R	opcode	rs	rt	rd	shamt	funct
I	opcode	rs	rt	immediate		



I-Format Problems (0/3)

♦Problem 0: Unsigned # sign-extended?

- addiu, sltiu, sign-extends immediates to 32 bits. Thus, # is a "signed" integer.

♦Rationale

- addiu so that can add w/out overflow
- sltiu suffers so that we can have ez HW
 - Does this mean we'll get wrong answers?
 - Nope, it means assembler has to handle any unsigned immediate $2^{15} \leq n < 2^{16}$ (i.e., with a 1 in the 15th bit and 0s in the upper 2 bytes) as it does for numbers that are too large.



I-Format Problems (1/3)

♦Problem 1:

- Chances are that addi, lw, sw and slti will use immediates small enough to fit in the immediate field.
- ...but what if it's too big?
- We need a way to deal with a 32-bit immediate in any I-format instruction.



I-Format Problems (2/3)

♦Solution to Problem 1:

- Handle it in software + new instruction
- Don't change the current instructions: instead, add a new instruction to help out

♦New instruction:

lui register, immediate

- stands for Load Upper Immediate
- takes 16-bit immediate and puts these bits in the upper half (high order half) of the specified register
- sets lower half to 0s



I-Format Problems (3/3)

♦Solution to Problem 1 (continued):

- So how does lui help us?

▪ Example:

addi \$t0,\$t0, 0xABABCD

becomes:

```
lui    $at, 0xABAB
ori    $at, $at, 0xCD
add    $t0,$t0,$at
```

- Now each I-format instruction has only a 16-bit immediate.
- Wouldn't it be nice if the assembler would do this for us automatically? (later)



Branches: PC-Relative Addressing (1/5)

Use I-Format

opcode	rs	rt	immediate
--------	----	----	-----------

- opcode specifies beq v. bne
- rs and rt specify registers to compare
- What can immediate specify?
 - Immediate is only 16 bits
 - PC (Program Counter) has byte address of current instruction being executed; 32-bit pointer to memory
 - So immediate cannot specify entire address to branch to.



Branches: PC-Relative Addressing (2/5)

- How do we usually use branches?
 - Answer: if-else, while, for
 - Loops are generally small: typically up to 50 instructions
 - Function calls and unconditional jumps are done using jump instructions (j and jal), not the branches.
- Conclusion: may want to branch to anywhere in memory, but a branch often changes PC by a small amount



Branches: PC-Relative Addressing (3/5)

- Solution to branches in a 32-bit instruction: PC-Relative Addressing
- 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 $\pm 2^{15}$ bytes from the PC, which should be enough to cover almost any loop.
- Any ideas to further optimize this?



Branches: PC-Relative Addressing (4/5)

- Note: 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 $\pm 2^{15}$ words from the PC (or $\pm 2^{17}$ bytes), so we can handle loops 4 times as large.



Branches: PC-Relative Addressing (5/5)

- Branch Calculation:
 - If we don't take the branch:

$$PC = PC + 4$$

$$PC + 4 = \text{byte address of next instruction}$$
 - If we do take the branch:

$$PC = (PC + 4) + (\text{immediate} * 4)$$
 - Observations
 - Immediate field specifies the number of words to jump, which is simply the number of instructions to jump.
 - Immediate field can be positive or negative.
 - Due to hardware, add immediate to (PC+4), not to PC; will be clearer why later in course



Branch Example (1/3)

MIPS Code:

```

Loop:  beq  $9,$0,End
        add  $8,$8,$10
        addi $9,$9,-1
        j    Loop

```

End:

beq branch is I-Format:

opcode = 4 (look up in table)
rs = 9 (first operand)
rt = 0 (second operand)
immediate = ???



Branch Example (2/3)

♦MIPS Code:

```
Loop:  beq  $9,$0, End
       addi $8,$8,$10
       addi $9,$9,-1
       j   Loop
```

End:

♦Immediate Field:

- Number of instructions to add to (or subtract from) the PC, starting at the instruction *following* the branch.
- In beq case, immediate = 3



Branch Example (3/3)

♦MIPS Code:

```
Loop:  beq  $9,$0, End
       addi $8,$8,$10
       addi $9,$9,-1
       j   Loop
```

End:

decimal representation:

4	9	0	3
---	---	---	---

binary representation:

000100	01001	00000	0000000000000011
--------	-------	-------	------------------



Questions on PC-addressing

- ♦ Does the value in branch field change if we move the code?
- ♦ What do we do if destination is $> 2^{15}$ instructions away from branch?
- ♦ Since it's limited to $\pm 2^{15}$ instructions, doesn't this generate lots of extra MIPS instructions?



J-Format Instructions (1/5)

- ♦ For branches, we assumed that we won't want to branch too far, so we can specify *change* in PC.
- ♦ For general jumps (j and jal), we may jump to *anywhere* in memory.
- ♦ Ideally, we could specify a 32-bit memory address to jump to.
- ♦ Unfortunately, we can't fit both a 6-bit opcode and a 32-bit address into a single 32-bit word, so we compromise.



J-Format Instructions (2/5)

- ♦ Define "fields" of the following number of bits each:

6 bits	26 bits
--------	---------

- ♦ As usual, each field has a name:

opcode	target address
--------	----------------

♦Key Concepts

- Keep opcode field identical to R-format and I-format for consistency.
- Combine all other fields to make room for large target address.



J-Format Instructions (3/5)

- ♦ For now, we can specify 26 bits of the 32-bit bit address.
- ♦ Optimization:
 - Note that, just like with branches, jumps will only jump to word aligned addresses, so last two bits are always 00 (in binary).
 - So let's just take this for granted and not even specify them.



J-Format Instructions (4/5)

- Now specify 28 bits of a 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
 - only if straddle a 256 MB boundary
 - If we absolutely need to specify a 32-bit address, we can always put it in a register and use the *j r* instruction.



J-Format Instructions (5/5)

- Summary:
 - New PC = { PC[31..28], target address, 00 }
- Understand where each part came from!
- Note: { , , } means concatenation
 - { 4 bits , 26 bits , 2 bits } = 32 bit address
 - { 1010, 1111111111111111111111111111, 00 } = 10101111111111111111111111111100



MIPS Instruction Formats Summary

- Minimum number of instructions required
 - Information flow: load/store
 - Logic operations: logic and/or/not, shift
 - Arithmetic operations: addition, subtraction, etc.
 - Branch operations:
- Instructions have different number of operands: 1, 2, 3
- 32 bits representing a single instruction
- Disassembly is simple and starts by decoding opcode field.

Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format



MIPS Addressing Modes

- Register addressing
 - Operand is stored in a register. R-Type
- Base or displacement addressing
 - Operand at the memory location specified by a register value plus a displacement given in the instruction. I-Type
 - Eg: lw, \$t0, 25(\$s0)
- Immediate addressing
 - Operand is a constant within the instruction itself. I-Type
- PC-relative addressing
 - The address is the sum of the PC and a constant in the instruction. I-Type
 - Eg: beq \$t2, \$t3, 25 # if (\$t2==\$t3), goto PC+4+100
- Pseudodirect addressing
 - The 26-bit constant is logically shifted left 2 positions to get 28 bits. Then the upper 4 bits of PC+4 is concatenated with this 28 bits to get the new PC address. J-type, e. g., j 2500



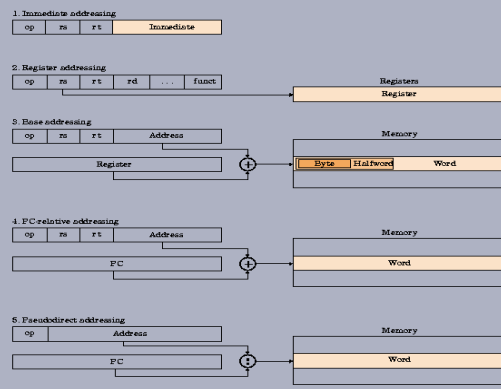
In-Class Question

- In PC-relative addressing,
 - How far can a program jump with respect to the current instruction location?

$$PC+4+4*(-2^{15}) \text{ ----- } PC+4+4*(2^{15}-1)$$
 - What if a conditional branch must jump further?
 - beq \$6, \$7, L1 #L1 >> (2¹⁵-1)
- bne \$6, \$7, L2
j L1
L2: ...
- Does this way work for all far away jumps?



MIPS Addressing Modes



Decoding Machine Language

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

Machine language $\Rightarrow$ Assembly language $\Rightarrow$ 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?

Decoding Example (1/7)

- Here are six machine language instructions in hexadecimal:

```
00001025_hex
0005402A_hex
11000003_hex
00441020_hex
20A5FFFF_hex
08100001_hex
```

- Let the first instruction be at address 4,194,304_{ten} (0x00400000_{hex}).
- Next step: convert hex to binary

Decoding Example (2/7)

- 🚗 The six machine language instructions in binary:

[illegible]

- 🔴 Next step: identify opcode and format

R	0	rs	rt	rd	shamt	funct
I	1,4-31	rs	rt	immediate		
J	2 or 3	target address				

Decoding Example (3/7)

- Select the opcode (first 6 bits) to determine the format:

Format:

R
R
R
I
J[illegible]

- Look at opcode:
 - 0 means R-Format,
 - 2 or 3 mean J-Format,
 - otherwise I-Format.
- Next step: separation of fields

Decoding Example (4/7)

- 🚨 Fields separated based on format/opcode:

Format:

R	0	0	0	2	0	37
R	0	0	5	8	0	42
I	4	8	0	+3		
I	0	2	4	2	0	32
I	8	5	5	-1		
J	2	1.048.577				

- **Next step: translate (“disassemble”) to MIPS assembly instructions**

Decoding Example (5/7)

- ## 🔴 MIPS Assembly (Part 1):

Address:	Assembly instructions:
0x00400000	or \$2,\$0,\$0
0x00400004	slt \$8,\$0,\$5
0x00400008	beq \$8,\$0,3
0x0040000c	add \$2,\$2,\$4
0x00400010	addi \$5,\$5,-1
0x00400014	j 0x100001

- **Better solution: translate to more meaningful MIPS instructions (fix the branch/jump and add labels, registers)**



Decoding Example (6/7)

♦ MIPS Assembly (Part 2):

```

or    $v0,$0,$0
Loop: slt    $t0,$0,$a1
      beq    $t0,$0,Exit
      add    $v0,$v0,$a0
      addi   $a1,$a1,-1
      j      Loop
Exit:

```

♦ Next step: translate to C code (be creative!)



Decoding Example (7/7)

Before Hex:

```

00001025hex
0005402Ahex
11000003hex
00441020hex
20A5FFFFhex
08100001hex

```

♦ After C code (Mapping below)

```

$v0: product  $a0: multiplicand
$a1: multiplier

product = 0;
while (multiplier > 0) {
    product += multiplicand;
    multiplier -= 1;
}

```

```

or    $v0,$0,$0
Loop: slt    $t0,$0,$a1
      beq    $t0,$0,Exit
      add    $v0,$v0,$a0
      addi   $a1,$a1,-1
      j      Loop
Exit:

```

Demonstrated Idea:
Instructions are just numbers, code is treated like data



Review from before: lui

♦ So how does lui help us?

Example:

```

addi   $t0,$t0, 0xABABCD

```

becomes:

```

lui     $at, 0xABAB
ori     $at,$at, 0xCD
add     $t0,$t0,$at

```

- Now each I-format instruction has only a 16-bit immediate.

♦ Wouldn't it be nice if the assembler would do this for us automatically?

- If number too big, then just automatically replace addi with lui, ori, add



True Assembly Language (1/3)

♦ Pseudoinstruction: A MIPS instruction that doesn't turn directly into a machine language instruction, but into other MIPS instructions

♦ What happens with pseudoinstructions?

- They're broken up by the assembler into several "real" MIPS instructions.
- But what is a "real" MIPS instruction? Answer in a few slides

♦ First some examples



Example Pseudoinstructions

♦ Register Move

```

move    reg2,reg1

```

Expands to:

```

add     reg2,$zero,reg1

```

♦ Load Immediate

```

li      reg,value

```

If value fits in 16 bits:

```

addi    reg,$zero,value

```

else:

```

lui     reg,upper 16 bits of value
ori     reg,$zero,lower 16 bits

```



True Assembly Language (2/3)

♦ Problem:

- When breaking up a pseudoinstruction, the assembler may need to use an extra reg.
- If it uses any regular register, it'll overwrite whatever the program has put into it.

♦ Solution:

- Reserve a register (\$1, called \$at for "assembler temporary") that assembler will use to break up pseudo-instructions.
- Since the assembler may use this at any time, it's not safe to code with it.



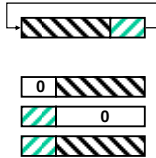
Example Pseudoinstructions

◆ Rotate Right Instruction

```
ror    reg, value
```

Expands to:

```
srl    $at, reg, value
sll     reg, reg, 32-value
or      reg, reg, $at
```



◆ “No Operation” instruction

```
nop
```

Expands to instruction = 0_{ten}

```
sll    $0, $0, 0
```



Example Pseudoinstructions

◆ Wrong operation for operand

```
addu   reg, reg, value # should be addiu
```

If value fits in 16 bits, addu is changed to:

```
addiu  reg, reg, value
```

else:

```
lui    $at, upper 16 bits of value
```

```
ori    $at, $at, lower 16 bits
```

```
addu   reg, reg, $at
```



In conclusion

◆ Disassembly is simple and starts by decoding opcode field.

- Be creative, efficient when authoring C

◆ Assembler expands real instruction set with pseudoinstructions

- Only hardware implemented instructions can be converted to raw binary
- Assembler's job to do conversion
- Assembler uses reserved register \$at
- pseudoinstructions make it much easier to write MIPS



Questions?