More MIPS instructions

- Last time we introduced the MIPS instruction set architecture, with its three-address instructions and dedicated loads and stores.
- Today we'll go into more detail about the ISA.
  - Constant values can be embedded in instructions.
  - Pseudo-instructions make programming easier for people.
  - Branches and jumps alter a program's control flow.
- We also discuss the binary representation of MIPS instructions.
  - This will reveal some restrictions on certain instructions.
  - We'll see workarounds for all of the limitations.

Immediate operands

- The ALU instructions we've seen so far expect register operands. How do you get data into registers in the first place?
  - $0, or $zero, is a dedicated register which always contains 0 and cannot be modified. (Go ahead and try.)

```
add   $t0, $0, $0        # $t0 = 0
```

  - Some MIPS instructions allow you to specify a signed constant, or immediate value, for the second source instead of a register. For example, here is the immediate add instruction, addi:

```
addi  $t0, $t1, 4        # $t0 = $t1 + 4
```

- MIPS is still considered a load/store architecture, because arithmetic operands cannot be from arbitrary memory locations. They must either be registers or constants that are embedded in the instruction.

Immediate examples

- We can use constant operands and $0 to initialize registers or to copy data between registers.

```
addi $a0, $0, 2000      # Initialize $a0 to 2000
add  $a1, $t0, $0       # Copy $t0 into $a1
```

- Here is a short example which sets the first two words of an array to 0 and 23, where the array starts at address 2000.

```
addi $a0, $0, 2000      # Initialize $a0 to 2000
sw   $0,  0($a0)        # Store 0 in address 2000
addi $t0, $0, 23        # Set $t0 = 23
sw   $t0, 4($a0)        # Store 23 in address 2004
```

Pseudo-instructions

- MIPS assemblers support pseudo-instructions that give the illusion of a more expressive instruction set, but are actually translated into one or more simpler, "real" instructions.
- For example, you can use the li and move pseudo-instructions:

```
li    $a0, 2000        # Load immediate 2000 into $a0
move  $a1, $t0         # Copy $t0 into $a1
```

- They are probably clearer than the MIPS instructions we just saw:

```
addi  $a0, $0, 2000    # Initialize $a0 to 2000
add   $a1, $t0, $0     # Copy $t0 into $a1
```

- We'll see lots more pseudo-instructions this semester.
  - A complete list of instructions is given in Appendix A of the text.
  - Unless otherwise stated, you can always use pseudo-instructions in your assignments and on exams.

Control flow in high-level languages

- The instructions in a program usually execute one after another, but it's often necessary to alter the normal control flow.
- Conditional statements execute only if some test expression is true.

```
// Find the absolute value of *a0
v0 = *a0;
if (v0 < 0)
    v0 = -v0;           // This might not be executed
v1 = v0 + v0;
```

- Loops cause some statements to be executed many times.

```
// Sum the elements of a five-element array a0
v0 = 0;
t0 = 0;
while (t0 < 5) {        // These statements
    v0 = v0 + a0[t0];   // will be executed
    t0++;               // five times
}
```

Jumps

- Many processors only provide relatively simple control flow instructions that just set the program counter.
- The MIPS jump instruction j always changes the value of the PC.

```
          lw   $t1, 0($a0)
          lw   $t2, 0($a1)
          add  $t0, $t1, $t2
          j    PartC           // Skip two instructions
PartB:    add  $t0, $t0, $t0
          addi $t0, $t0, 1
PartC:    sw   $t0, 8($a1)
```

- Here "PartB" and "PartC" are labels which represent the addresses of their respective instructions.
- The two skipped instructions in this example could still be executed, if a jump (or branch) to "PartB" exists elsewhere in the program.

Branches

- A branch instruction *may* change the PC, depending on whether or not some condition is true.
- For example, a bne branches if its two register operands are *not* equal. Otherwise, execution continues as usual with the next instruction.

```
        bne   $t0, $t1, PartC   // If $t0 != $t1 then
PartB:  lw    $t1, 0($a0)       //   these instructions
        addi  $t1, $t1, 5       //   will be skipped
PartC:  sw    $t1, 0($a0)
```

- Here are some other useful MIPS branches.

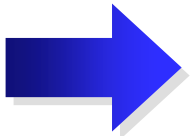
```
beq  $t0, $t1, L0    // Branch if $t0 == $t1
blt  $t0, $t1, L1    // Branch if $t0 < $t1
ble  $t0, $t1, L2    // Branch if $t0 <= $t1
bgt  $t0, $t1, L3    // Branch if $t0 > $t1
bge  $t0, $t1, L4    // Branch if $t0 >= $t1
```

- There are also immediate versions of these branches, where the second source is a constant instead of a register.

Translating an if-then statement

- We can use branch instructions to translate if-then statements into MIPS assembly code.

```
v0 = *a0;
if (v0 < 0)
    v0 = -v0;
v1 = v0 + v0;
```



```
      lw   $v0, 0($a0)
      bge  $v0, $0, L
      sub  $v0, $0, $v0
L:    add  $v1, $v0, $v0
```

- Sometimes it's easier to *invert* the original condition.
  - In this case, we changed "continue if v0 < 0" to "skip if v0 >= 0".
  - This saves a few instructions in the resulting assembly code.

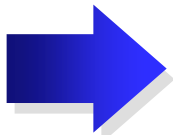
Translating a while statement

- Here is a translation of the while loop, using both branches and jumps.

```
v0 = 0;
t0 = 0;
while (t0 < 5) {
    v0 = v0 + a0[t0];



    t0++;
}
...
```



```
        move $v0, $0
        move $t0, $0
Loop:   bge  $t0, 5, Exit
        mul  $s0, $t0, 4
        add  $s0, $a0, $s0
        lw   $s1, 0($s0)
        add  $v0, $v0, $s1
        addi $t0, $t0, 1
        j    Loop
Exit:   ...
```

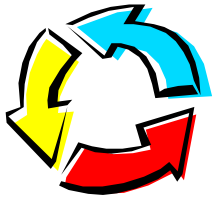
- Note how complex the loop body in magenta is!
  - The loop index $t0 is multiplied by 4 to arrive at a byte offset.
  - That is added to the array's base address, to compute the address of array element a0[t0].
  - Finally, a0[t0] is loaded into register $s1 for the add instruction.

Translating a for loop

- Every for loop can be re-written as a while loop.

```
v0 = 0;

for (t0=0; t0 < 5; t0++)
    v0 = v0 + a0[t0];
```



```
v0 = 0;
t0 = 0;
while (t0 < 5) {
    v0 = v0 + a0[t0];
    t0++;
}
```

- So if you know how a while loop works, you'll also understand for loops.

Pseudo-branches

- The MIPS processor only supports two branch instructions, beq and bne. The other branches are all pseudo-instructions!
- The (real) set-if-less-than instruction slt compares two registers.
  - If the first is less than the second, the destination is set to 1.
  - Otherwise, the destination register is set to 0.

```
addi $t1, $t0, 1        // $t1 = $t0 + 1
slt  $v0, $t0, $t1      // $v0 = 1 since $t0 < $t1
slt  $v1, $t1, $t0      // $v1 = 0 since $t1 >= $t0
```

- Many pseudo-branches can be implemented using slt. For example, a branch-if-less-than instruction `blt $a0, $a1, Label` translates into the following.

```
slt  $at, $a0, $a1      // $at = 1 if $a0 < $a1
bne  $at, $0, Label     // Branch if $at != 0
```

Immediate pseudo-branches

- There is also an immediate version of slt, slti, which compares registers against constants.

```
slti $at, $a0, 5        // $at = 1 if $a0 < 5
```

- This supports immediate branches, which are also pseudo-instructions. For example, `blti $a0, 5, Label` is translated into two instructions.

```
slt  $at, $a0, 5
bne  $at, $0, Label    // Branch if $a0 < 5
```

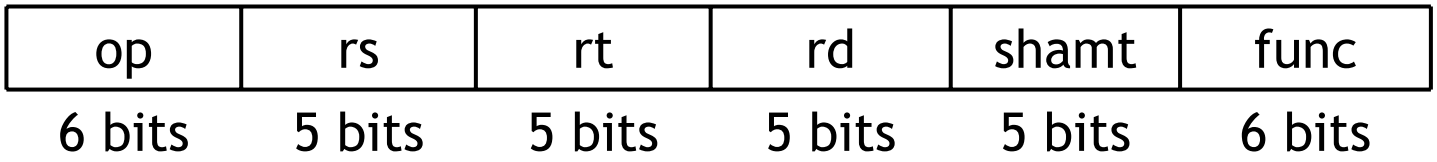
- All of the pseudo-branches need a register to save the result of slt, even though it's not needed afterwards.
  - MIPS assemblers use register $1, or $at, for temporary storage.
  - You should be careful in using $at in your own programs, as it may be overwritten by assembler-generated code.
- Later this semester we'll see how supporting just beq and bne simplifies the processor design.

Assembly vs. machine language

- So far we've been using assembly language.
  - We assign names to operations (e.g., add) and operands (e.g., $t0).
  - Branches and jumps use labels instead of actual addresses.
  - Assemblers support many pseudo-instructions.
- Programs must eventually be translated into machine language, a binary format that can be stored in memory and decoded by the CPU.
- MIPS machine language is designed to be easy to decode, much like the basic processor studied in CS231.
  - Each MIPS instruction is the same length, 32 bits.
  - There are only three different instruction formats, which are very similar to each other.
- Studying MIPS machine language will also reveal some restrictions in the instruction set architecture, and how they can be overcome.

R-type format

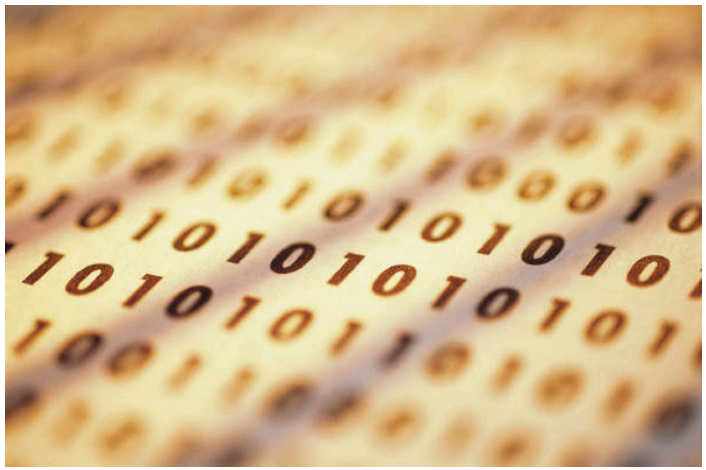
- Register-to-register arithmetic instructions use the R-type format.



- This format includes six different fields.
  - op is an operation code or opcode that selects a specific operation.
  - rs and rt are the first and second source registers.
  - rd is the destination register.
  - shamt is used for shift instructions, which we haven't seen yet.
  - func is used together with op to select an arithmetic instruction.
- The inside back cover of the textbook lists opcodes and function codes for all of the MIPS instructions.

About the registers

- We have to encode register names as 5-bit numbers from 00000 to 11111.
  - For example, $t8 is register $24, which is represented as 11000.
  - The complete mapping is given on page A-23 in the book.
- The number of registers available affects the instruction length.
  - Each R-type instruction references 3 registers, which requires a total of 15 bits in the instruction word.
  - We can't add more registers without either making instructions longer than 32 bits, or shortening other fields like op and possibly reducing the number of available operations.



I-type format

- Load, store, branch and immediate instructions all use the I-type format.

- For uniformity, op, rs and rt are in the same positions as in the R-format.
- The meaning of the register fields depends on the exact instruction.
  - rs is a source register—an address for loads and stores, or an operand for branch and immediate arithmetic instructions.
  - rt is a source register for branches, but a destination register for the other I-type instructions.
- The address is a 16-bit signed two's-complement value.
  - It can range from -32,768 to +32,767.
  - But that's not always enough!

Larger constants

- Larger constants can be loaded into a register 16 bits at a time.
  - The load upper immediate instruction lui loads the highest 16 bits of a register with a constant, and clears the lowest 16 bits to 0s.
  - An immediate logical OR, ori, then sets the lower 16 bits.
- To load the 32-bit value 0000 0000 0011 1101 0000 1001 0000 0000:

```
lui $s0, 61             # $s0 = 003D 0000 (in hex)
ori $s0, $s0, 2304      # $s0 = 003D 0900
```

- This illustrates the principle of making the common case fast.
  - Most of the time, 16-bit constants are enough.
  - It's still possible to load 32-bit constants, but at the cost of two extra instructions and one temporary register.
- Pseudo-instructions may contain large constants. Assemblers including SPIM will translate such instructions correctly.

Loads and stores

```
lw $t0, constant($a0)
```

- The limited 16-bit constant can present problems for array accesses.
  - If an array's base address is higher than 32,767, the constant won't be able to represent it, like we showed last time.
  - If the array contains more than 32,767 *bytes* of data, the constant won't be able to represent the index.
- In these situations, we have to compute the exact address of the desired array element and load it into the register manually.
- For instance, the following code fragment can load the one millionth byte of an array that starts at decimal address 3,000,000.

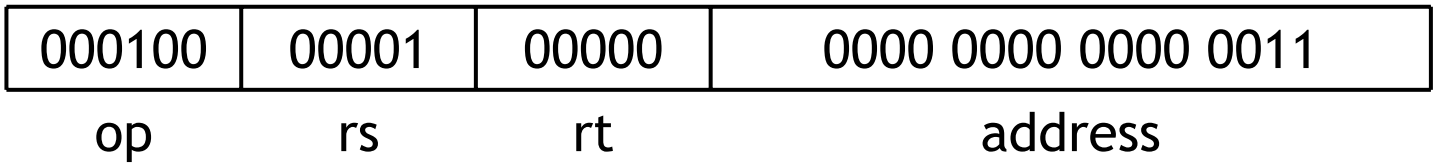
```
    lui  $s0, 61
    ori  $s0, $s0, 2304        # $s0 = 4000000 (decimal)
    lb   $t1, 0($s0)           # Read from Mem[4000000]
```

Branches

- For branch instructions, the constant field is not an address, but an *offset* from the current program counter (PC) to the target address.

```
        beq  $at, $0, L
        add  $v1, $v0, $0
        add  $v1, $v1, $v1
        j    Somewhere
L:      add  $v1, $v0, $v0
```

- Since the branch target L is three *instructions* past the beq, the address field would contain 3. The whole beq instruction would be stored as:



- For some reason SPIM is off by one, so the code it produces would contain an address of 4. (But SPIM branches still execute correctly.)

Larger branch constants

- Empirical studies of real programs show that most branches go to targets less than 32,767 instructions away—branches are mostly used in loops and conditionals, and programmers are taught to make code bodies short.
- If you do need to branch further, you can use a jump with a branch. For example, if "Far" is very far away, then the effect of:

```
        beq  $s0, $s1, Far
        ...
```

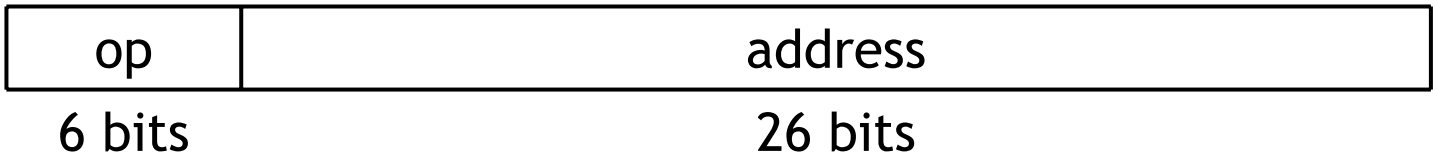
  can be simulated with the following actual code.

```
        bne  $s0, $s1, Next
        j    Far
Next:   ...
```

- Again, the MIPS designers have taken care of the common case first.

J-type format

- Finally, the jump instruction uses the J-type instruction format.



- The jump instruction contains a *word* address.
  - Remember that each MIPS instruction is one word long, and word addresses must be divisible by four.
  - So instead of saying "jump to address 4000," it's enough to just say "jump to instruction 1000."
  - A 26-bit address field lets you jump to any address from 0 to 2^{28}.
- For even longer jumps MIPS has a jump register, or jr, instruction that can jump to any 32-bit address stored in a register.

```
jr  $ra      # Jump to 32-bit address in register $ra
```

Summary

- Today we saw several additional MIPS features.
  - Immediate instructions permit the use of constant values.
  - Assemblers can translate more powerful pseudo-instructions into the simpler instructions actually supported in hardware.
  - Branches and jumps help to implement various high-level control flow structures, like conditional statements and loops.
- We also studied MIPS machine language.
  - All instructions are the same length, 32 bits.
  - The three instruction formats are I-type, R-type and J-type.
- The 16-bit constant field in I-type instructions is enough for most common situations. In other cases, we can always resort to longer code fragments.
- Sections begin this week, and will provide an introduction to the SPIM assembler that we use to run MIPS programs.