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.
- In addition to the li (load immediate) we saw on last lecture, you can use the move and la pseudo-instructions:

```
move  $a1, $t0          # Copy $t0 into $a1
la    $a0, data         # Load address of memory labeled
                        # 'data' into $a0
```

- Simpler, clearer than equivalent MIPS instructions.

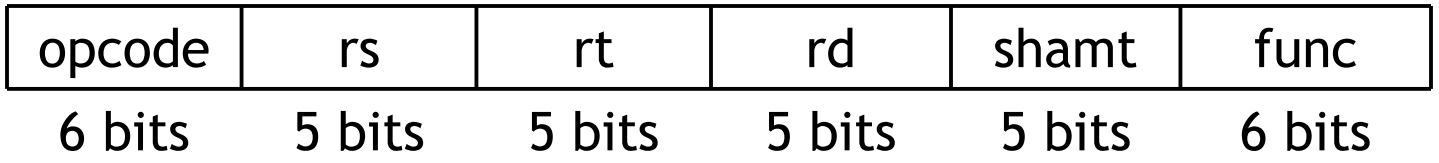
- We'll see lots more pseudo-instructions this semester.
  - A complete list of instructions is given in Appendix A of the text.
- Unless otherwise stated, you can always use pseudo-instructions in your assignments and on exams.
- For now, we'll focus on real instructions…

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.
  - Each MIPS instruction is the same length, 32 bits.
  - There are only three different instruction formats, which are very similar to each other.
  - The format of an instruction is determined by its first 6 bits, known as the operation code, or *opcode*.
- Studying MIPS machine language will also reveal some restrictions in the instruction set architecture, and how they can be overcome.

R-type format

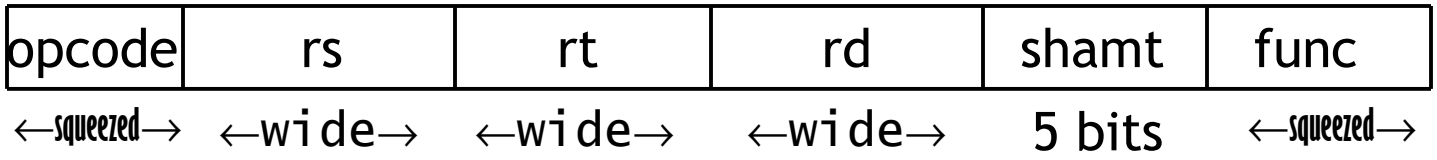
- Register-to-register arithmetic instructions use the R-type format.



- This format includes six different fields.
  - opcode is an operation code that selects a specific operation.
  - rs and rt are the first and second source registers.
  - rd is the destination register.
  - shamt is only used for shift instructions.
  - func is used together with opcode to select an arithmetic instruction.
- The inside back cover of the textbook lists opcodes and function codes for all of the MIPS instructions.

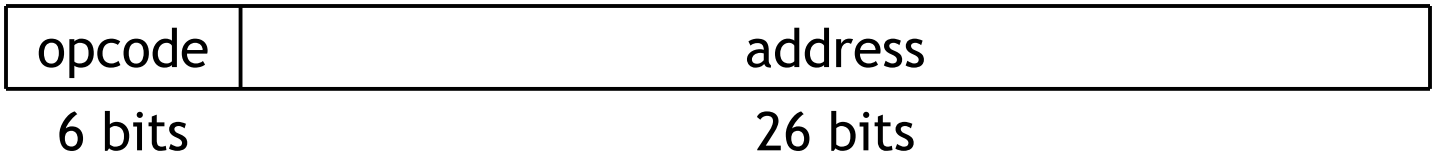
MIPS registers

- 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 opcode and possibly reducing the number of available operations.



J-type format

- The j and jal instructions use the J-type instruction format.



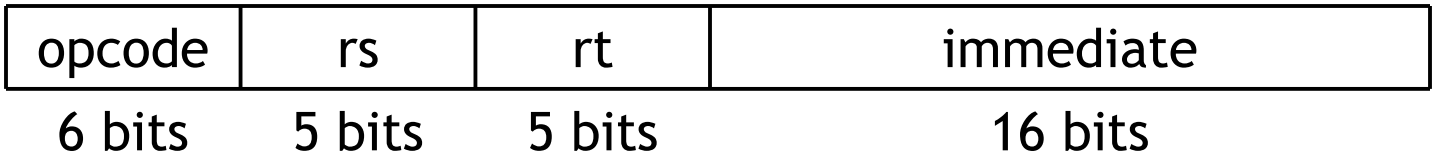
- Last 26 bits: *word* address of the target instruction
  - Remember that each MIPS instruction is one word long, and word addresses must be divisible by four.
  - So instead of saying "jump to address 4036," it's enough to just say "jump to instruction 1009".
  - A 26-bit address field lets you jump to any address from 0 to 2^{28}.
    - your MP solutions had better be smaller than 256MB

- For even longer jumps, the jump register (jr) instruction can be used.

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

I-type format

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



- For uniformity, opcode, rs and rt are located as in the R-format.
- The meaning of the register fields depends on the exact instruction.
  - rs is *always* 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 and stores, but a *destination* register for the other I-type instructions.
- The immediate is a 16-bit signed two's-complement value.
  - It can range from -32,768 to +32,767.
  - But that's not always enough! E.g. how do you load a 32-bit constant into a register?

Loading 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 (sets 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, 0x003D          # $s0 = 003D 0000 (in hex)
ori $s0, $s0, 0x0900     # $s0 = 003D 0900
```

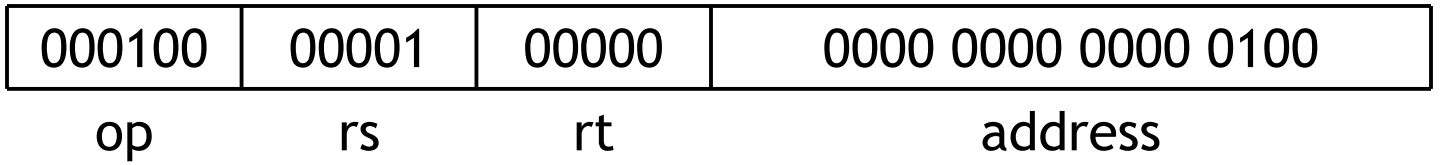
- This illustrates the principle of making the common case fast.
  - Most of the time, 16-bit constants are enough.
  - It's still possible to load 32-bit constants, but at the cost of two instructions and one temporary register.
- Pseudo-instructions may contain large constants. Assemblers including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

Branches

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

```
L0:   beq  $at, $0, L4
L1:   add  $v1, $v0, $0
L2:   add  $v1, $v1, $v1
L3:   j    Somewhere
L4:   add  $v1, $v0, $v0
```

- Since the branch target L4 is *four* instructions past the beq, the address field contains 4. The whole beq instruction would be stored as:



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.

Pseudo-branches

- The MIPS processor only supports two branch instructions, beq and bne, but to simplify your life the assembler provides the following other branches:

```
        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.

- Later this semester we'll see how supporting just beq and bne simplifies the processor design.

Implementing pseudo-branches

- Most pseudo-branches are implemented using slt. For example, a branch-if-less-than instruction `blt $a0, $a1, Label` is translated into the following.

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

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

```
slti $at, $a0, 5       // $at = 1if $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.
  - Be careful in using $at in your own programs – if you use it, the assembler warns that you may overwrite assembler-generated code.

Translating an if-then statement

- We saw something very similar to an if-then statement in section:

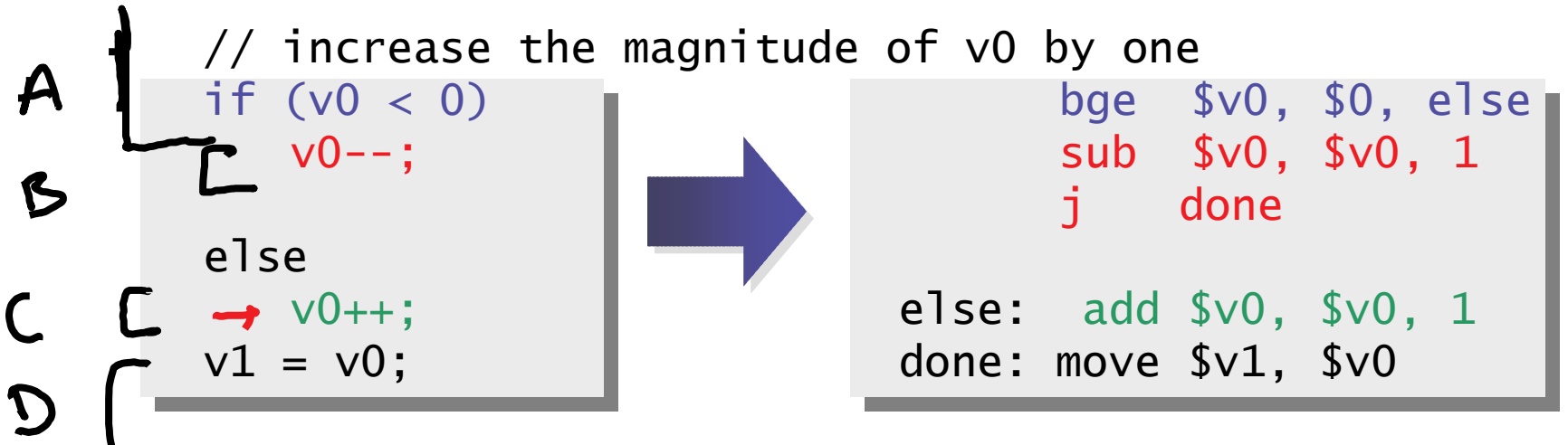
```
while(t0 < t1){
      ...
}

// other stuff
```

```
loop: if(t0 < t1) {
        ...
        goto loop;
      }
      // other stuff
```

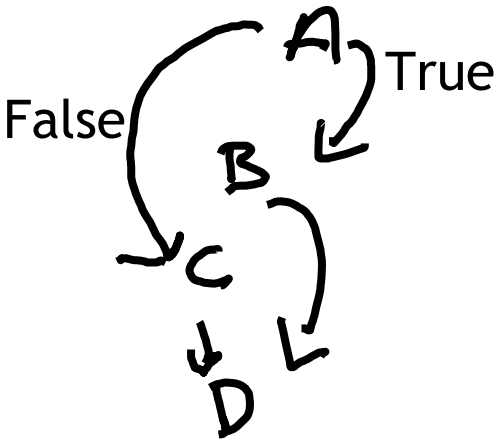
- Sometimes it's easier to *invert* the original condition.
  - We can save a few instructions in the resulting assembly code.

Translating an if-then-else statements

- If there is an else clause, it is the target of the conditional branch
  - And the then clause needs a jump over the else clause



- Dealing with else-if code is similar, but the target of the first branch will be another if statement.
  - Drawing the control-flow graph can help you out.

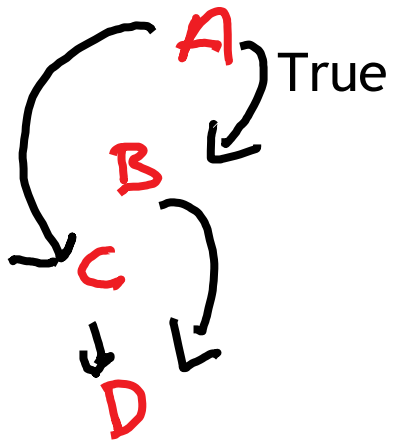


Control-flow Example

- Translate the following into MIPS:

```
A  if(t0 < t1 || t0 < t2)
B      t0 = getRandom();
   else
C      t1--;
D  t2++;
```

Control-flow graph



False, i.e. t0≥t1 && t0≥t2

```
    blt  $t0, $t1, then
                          # 1st test failed
    bge  $t0, $t2, else   # 2nd test also failed
  then:
    jal  getRandom
    move $t0, $t1
    j    done
  else:
    addi $t1, $t1, -1
  done:
    addi $t2, $t2, 1
```

Pointers & Pointer Arithmetic

- Many programmers have a vague understanding of pointers
  - Looking at assembly code is useful for their comprehension.

Normal C code

```
void reverse(int a[], int len) {
   int i, t1, t2;
   for (i = 0; i < len/2; ++i) {
      t1 = a[i]; t2 = a[len-1-i];

      t1 = t1 ^ t2;  // ^ is XOR
      t2 = t1 ^ t2;
      t1 = t1 ^ t2;

      a[i] = t1; a[len-1-i] = t2;
   }
}
```

Pointer version (more efficient)

```
void reverse(int *a, int len) {
   int t1, t2;
   int *p1 = a;
   int *p2 = a + len - 1;
   while(p1 < p2) {
      t1 = *p1; t2 = *p2;

      t1 = t1 ^ t2;  // ^ is XOR
      t2 = t1 ^ t2;
      t1 = t1 ^ t2;

      *p1 = t1; *p2 = t2;
      ++p1; --p2;
   }
}
```

What is a Pointer?

- A pointer is an address.
- Two pointers that point to the same thing hold the same address
- Dereferencing a pointer means loading from the pointer's address
- A pointer has a type; the type tells us what kind of load to do
  - Use load byte (lb) for char *
  - Use load half (lh) for short *
  - Use load word (lw) for int *
  - Use load single precision floating point (l.s) for float *
- Pointer arithmetic is often used with pointers to arrays
  - Incrementing a pointer (i.e., ++) makes it point to the next element
  - The amount added to the point depends on the type of pointer
    - pointer = pointer + sizeof(*pointer's type)*
      - 1 for char *, 4 for int *, 4 for float *, 8 for double *

What is really going on here…

```
int *p1 = a;
int *p2 = a + len - 1;

while(p1 < p2) {
  t1 = *p1; t2 = *p2;

  // swap t1, t2

  *p1 = t1; *p2 = t2;
  ++p1; --p2;
}
```

```
  move  p1, a
  sub   t0, len, 1
  sll   t0, t0, 2
  add   p2, a, t0
loop:
  bge   p1, p2, done

  lw    t1, 0(p1)
  lw    t2, 0(p2)

  # swap

  sw    t1, 0(p1)
  sw    t2, 0(p2)

  addi  p1, p1, 4
  addi  p2, p2, -4

  j     loop
done:
```

Summary

- Machine language is the binary representation of instructions:
  - The format in which the machine actually executes them
- MIPS machine language is designed to simplify processor implementation
  - Fixed length instructions
  - 3 instruction encodings: R-type, I-type, and J-type
  - Common operations fit in 1 instruction
    - Uncommon (e.g. long immediates) require more than one
- Pointers are just addresses!!
  - "Pointees" are locations in memory
- Pointer arithmetic updates the address held by the pointer
  - "int_ptr++" points to the next element in an int-array
  - Pointers are typed so address is incremented by sizeof(pointee)