CS 61C: Great Ideas in Computer Architecture (Machine Structures)

More MIPS Machine Language

Instructors:
Michael Greenbaum

http://inst.eecs.Berkeley.edu/~cs61c/su11

Levels of Representation/Interpretation

High Level Language Program (e.g., C)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

Compiler

Assembly Language Program (e.g., MIPS)

```
lw   $t0, 0($2)
lw   $t1, 4($2)
sw   $t1, 0($2)
sw   $t0, 4($2)
```

Anything can be represented as a *number*, i.e., data or instructions

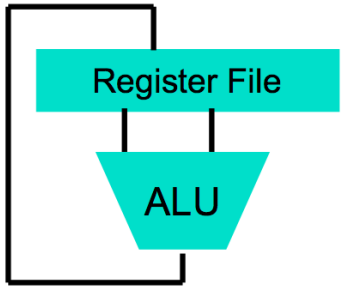
Assembler

Machine Language Program (MIPS)

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

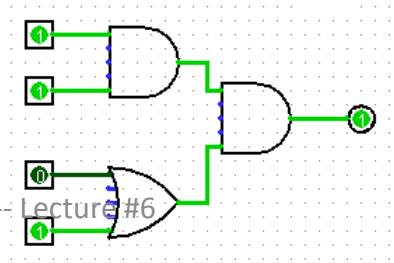
Machine Interpretation

Hardware Architecture Description (e.g., block diagrams)



Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)



Agenda

- Instructions For Data Transfer
- Instructions For Decisions
- **Administrivia**
- Misc: Immediates, Overflow, Inequalities
- **Break**
- Support for Strings
- C to MIPS Example
- Summary

Data Structures vs. Simple Variables

- C variables map onto registers; what about large data structures like arrays?
- Remember *memory*, our big array indexed by addresses.
- But MIPS instructions only operate on registers!
- Data transfer instructions transfer data between registers and memory:
  - Memory to register
  - Register to memory

Transfer from Memory to Register

- MIPS instruction: *Load Word*, abbreviated `lw`
- Load Word Syntax:

  lw 1,3(2)

  – where

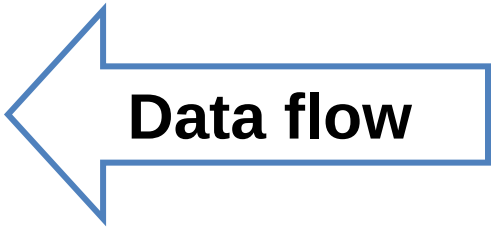
    1) register that will receive value

    2) register containing pointer to memory

    3) numerical offset from 2) in bytes

- Adds 3) to address stored in 2), loads **FROM** this address in memory and puts result in 1).

Transfer from Memory to Register



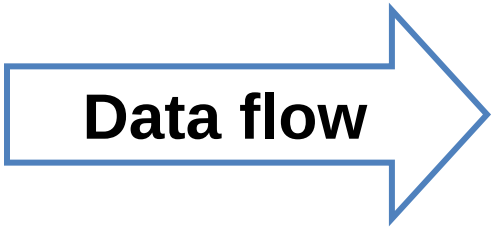
Example: `lw $t0,12($s0)`

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 (note offset must be a constant known at assembly time)

Transfer from *Register* to *Memory*

- MIPS instruction: *Store Word*, abbreviated `sw`
- Syntax similar to `lw`.



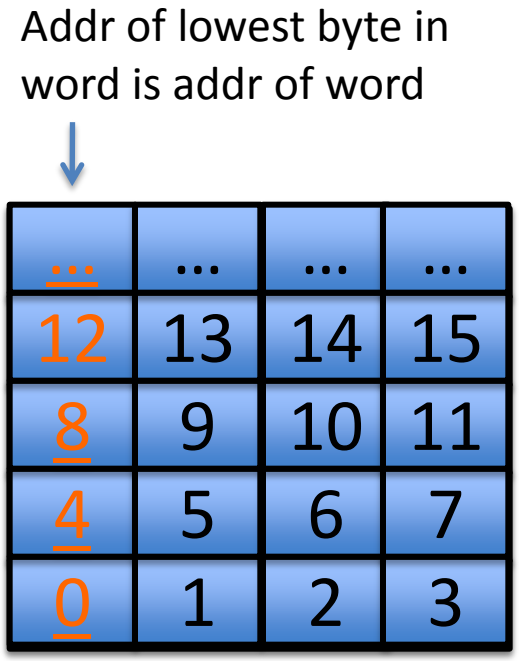
- Example: `sw $t0,12($s0)`

  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"

Memory Addresses are in Bytes

- Lots of data is smaller than 32 bits, but rarely smaller than 8 bits – works fine if everything is a multiple of 8 bits
- 8 bit item is called a *byte*

  (1 word = 4 bytes)
- Memory addresses are really in *bytes*, not words
- Word addresses are 4 bytes apart
  - Word address is same as leftmost byte



lw/sw Example

- Assume A is an array of 100 words, variables `g` and `h` map to registers `$s1` and `$s2`, the starting address, or base address, of the array A is in `$s3`

```
A[10] = h + A[3];
```

- Turns into

```
lw  $t0,12($s3)  # Temp reg $t0 gets A[3]
add $t0,$s2,$t0  # t0 = h + A[3]
sw  $t0,40($s3)  # A[10] = h + A[3]
```

Remember, Data can be Anything

- Key Concept: A register holds 32 bits of raw data. That data might be a (signed) int, an unsigned int, a pointer (memory addr), and so on
  - E.g., If you write: add $t2,$t1,$t0
    then $t0 and $t1 better contain values that can be added
  - E.g., If you write: lw $t2,0($t0)
    then $t0 better contain a pointer

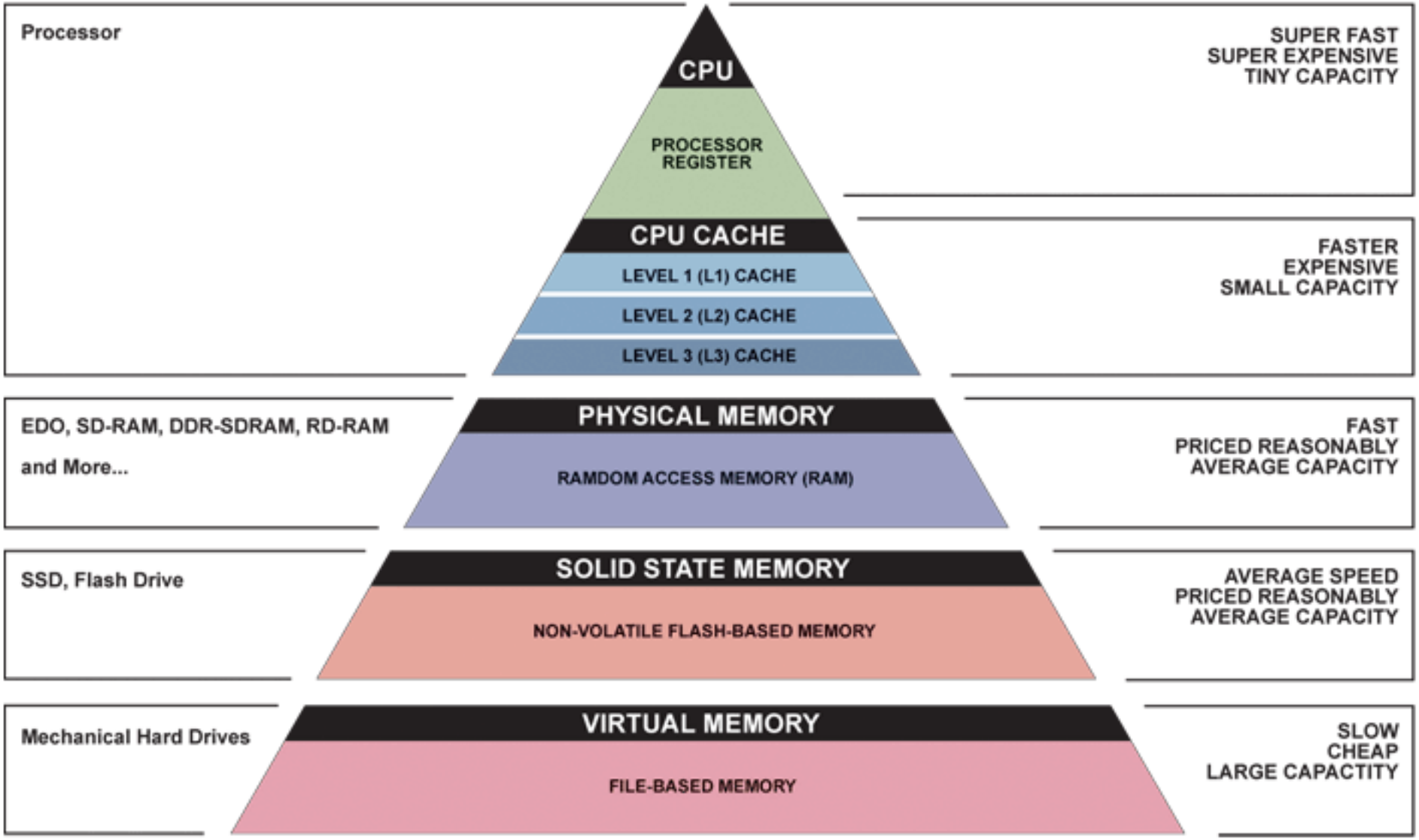
Speed of Registers vs. Memory

- Given that
  - Registers: 32 words (128 Bytes)
  - Memory: Billions of bytes (2 GB to 8 GB on laptop)
- and the RISC principle is
  - Smaller is faster
- How much faster are registers than memory??
- About 100-500 times faster!

Role of Registers vs. Memory

- What if more variables than registers?
  - Compiler tries to keep most frequently used variable in registers
  - Less common variables in memory: spilling
- Why not keep all variables in memory? Why not implement instructions that operate directly on memory?
  - This would be really slow! Interfacing with memory takes a long time.

Great Idea #3: Principle of Locality/ Memory Hierarchy



Peer Instruction

We want to translate `*x = *y` into MIPS
(`x`, `y` ptrs stored in: `$s0 $s1`)

```
1: add $s0, $s1, zero
2: add $s1, $s0, zero
3: lw  $s0, 0($s1)
4: lw  $s1, 0($s0)
5: lw  $t0, 0($s1)
6: sw  $t0, 0($s0)
7: lw  $s0, 0($t0)
8: sw  $s1, 0($t0)
```

green) 1 or 2
purple) 3 or 4
yellow) 5→6
red) 6→5
blue) 7→8

Agenda

- Instructions For Data Transfer
- Instructions For Decisions
- **Administrivia**
- Misc: Immediates, Overflow, Inequalities
- **Break**
- Support for Strings
- C to MIPS Example
- Summary

Side Note: Register Zero

- One particular value, 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`
- value fixed in hardware, so an instruction

  `add $zero,$zero,$s0`

  will not do anything!

Computer Decision Making

- Based on computation, do something different
  - Otherwise program does the same thing every time it's run!
- In programming languages: if-statement
  - Sometimes combined with gotos and labels
- MIPS: conditional instruction is

```
beq reg1,reg2,L1
```

  means go to statement labeled L1
  if value in reg1 == value in reg2
- `beq` stands for *branch if equal*
- Other instruction: `bne` for *branch if not equal*

Making Decisions in MIPS

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

- f => $s0, g => $s1, h => $s2, i => $s3, j => $s4
- If false, skip “then” part to “else” part
- Otherwise, (its true) do “then” part and skip over “else” part

```
        bne $s3,$s4,Else   # go to Else part if i ≠ j
        add $s0,$s1,$s2    # f = g + h (Then part)
        j Exit             # go to Exit
Else:   sub $s0,$s1,$s     # f = g – h (Else part)
Exit:
```

Unconditional Jump

- The jump instruction `j` moves control to the specified label:

```
j Exit
```

- Why not use `beq $0 $0 Label`? Doesn't MIPS favor a simple instruction set?
  - There is a very specific reason that `j` is used instead. Wait until Thursday to find out.

Implementing Control Flow

- 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 these loops as well.
- Key Concept: Though there are multiple ways of writing a loop in MIPS, the key to decision-making is conditional branch

What We Know So Far

- Arithmetic Instructions
  - add, sub, and, or, sll, srl
- Data Transfer instructions
  - lw, sw
- Branch and Jump instructions
  - beq, bne, j

Agenda

- Instructions For Data Transfer
- Instructions For Decisions
- **Administrivia**
- Misc: Immediates, Overflow, Inequalities
- **Break**
- Support for Strings
- C to MIPS Example
- Summary

Administrivia

- HW2 Posted.
  - Remember, it's big!
- Project 1 posted by Thursday, due 7/10.
  - No homework that week.
- Will be sending out a survey to determine times for Midterm and Final. If you have a conflict with any of the times listed, let us know in the comments.

Agenda

- Instructions For Data Transfer
- Instructions For Decisions
- **Administrivia**
- Misc: Immediates, Overflow, Inequalities
- **Break**
- Support for Strings
- C to MIPS Example
- Summary

Immediates

- Immediates are numerical constants.
- They appear often in code, so there are special instructions for them.
- *Add Immediate* (`addi`):

  `addi $s0,$s1,10` (in MIPS)

- Syntax similar to `add` instruction, except that last argument is a (signed) 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` same as `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`

Instructions with Immediates

- `andi`
  - `andi reg1,reg2,Imm`
  - Bitwise `and` of `reg2` and `Imm`, store result in `reg1`.
- `ori`
  - `ori reg1,reg2,Imm`
  - Bitwise `or` of `reg2` and `Imm`, store result in `reg1`.
- Is the immediate here signed or unsigned?
  - It makes the most sense to interpret an argument to a bitwise operation as unsigned.
  - Green Sheet contains information like this.

Overflow in Arithmetic

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

Inequalities in MIPS (1/4)

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

```
if (reg2 < reg3)
   reg1 = 1;
else reg1 = 0;
```

"set" means "change to 1", otherwise, reset to 0.

Inequalities in MIPS (2/4)

- How do we use this? What is the MIPS for this:

```
if (g < h) goto Less; #g:$s0,h:$s1
```

- Answer: compiled MIPS code…

```
slt $t0,$s0,$s1 # $t0 = 1 if g<h
                # $t0 = 0 if g>=h
bne $t0,$0,Less # goto Less
                # if $t0!=0
```

- 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 (3/4)

- Now we can implement <, but how do we implement >, ≤ and ≥ ?
- We could add 3 more instructions, but:
  - MIPS goal: Simpler is Better
- Can implement all four of <, ≤, >, ≥ by:
  - Switching the 2nd and 3rd arguments to `slt` (so it behaves like "set on greater than")
  - Using `bne` instead of `beq`
  - Four total variations, one for each of <, ≤, >, ≥ .

Inequalities in MIPS (4/4)

- Lets compile this by hand:

```
if (g >= h) goto Label; #g:$s0, h:$s1
```

- Answer: compiled MIPS code…

```
slt $t0,$s0,$s1 # $t0 = 1 if g<h
                # $t0 = 0 if g>=h
beq $t0,$0,Label # goto Label if g>=h
```

Immediates in Inequalities

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

C

```
Loop: ...
    if (g >= 1) goto Loop;
```

MIPS

```
Loop:   . . .

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

An 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`**?

(**`$s0 = FFFF FFFA`**$_{hex}$, **`$s1 = 0000 FFFA`**$_{hex}$)

```
slt $t0, $s0, $s1
sltu $t1, $s0, $s1
```

Peer Instruction

```
Loop:addi $s0,$s0,-1    # i = i - 1
     slti $t0,$s1,2     # $t0 = (j < 2)
     beq  $t0,$0 ,Loop  # goto Loop if $t0 == 0
     slt  $t0,$s1,$s0   # $t0 = (j < i)
     bne  $t0,$0 ,Loop  # goto Loop if $t0 != 0
```

($s0=i, $s1=j)

What C code properly fills in the blank in loop below?

```
do {i--;} while(____);
```

g)	j	<	2	&&	j	< i
p)	j	<=	2	&&	j	<= i
r)	j	>	2	&&	j	> i
b)	j	>=	2	&&	j	>= i
g)	j	<	2	||	j	< i
p)	j	<=	2	||	j	<= i
r)	j	>	2	||	j	> i
b)	j	>=	2	||	j	>= i

Peer Instruction Answer

```
Loop:addi $s0,$s0,-1    # i = i - 1
     slti $t0,$s1,2     # $t0 = (j < 2)
     beq  $t0,$0 ,Loop  # goto Loop if $t0 == 0
     slt  $t0,$s1,$s0   # $t0 = (j < i)
     bne  $t0,$0 ,Loop  # goto Loop if $t0 != 0
```

($s0=i, $s1=j)

What C code properly fills in the blank in loop below?

```
do {i--;} while(____);
```

g) j < 2 && j < i
p) j >= 2 && j < i
y) j < 2 && j >= i
r) j >= 2 && j >= i
b) j > 2 && j < i
g) j < 2 || j < i
p) j >= 2 || j < i
y) j < 2 || j >= i
r) j >= 2 || j >= i
b) j > 2 || j < i

Agenda

- Instructions For Data Transfer
- Instructions For Decisions
- **Administrivia**
- Misc: Immediates, Overflow, Inequalities
- **Break**
- Support for Strings
- C to MIPS Example
- Summary

Strings: C vs. Java

- C: each char is 8-bit ASCII. Deal with 1 byte at a time.
- Java: 16-bit UNICODE (a much larger character vocabulary). Deal with 2 bytes at a time.

Support for Characters and Strings

- Load a word, use `andi` to isolate byte

```
lw   $s0,0($s1)
andi $s0,$s0,255 # Zero everything but last 8 bits
```

- RISC Design Principle: "Make the Common Case Fast"—Many programs use text: MIPS has *load byte* instruction (*lb*)

```
lb $s0,0($s1)
```

- Also *store byte* instruction (*sb*)

Loading, Storing bytes

- 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`)

Support for Characters and Strings

- MIPS also provides fast support for Unicode:

- *load halfword* instruction (`lh`)

```
lh $s0,0($s1)
```

  - There's also *load halfword unsigned* (`lhu`)
- *store halfword* instruction (`sh`)

```
sh $s0,0($s1)
```

MIPS Signed vs. Unsigned – Three Different Meanings!

- MIPS terms Signed/Unsigned "overloaded":
  - Do/Don't sign extend
    - `(lb, lbu)`
  - Do/Don't overflow
    - `(add, addi, sub, mult, div)`
    - `(addu, addiu, subu, multu, divu)`
  - Do signed/unsigned compare
    - `(slt, slti/sltu, sltiu)`

What Do We Know Now?

- Arithmetic
  - add, addi, addu, addiu, sub, subu, and, andi, or, ori, sll, slr, slt, slti, sltiu!
- Data Transfer
  - lw, sw, lb, lbu, sb, lh, lhu, sh!
- Branches and Jumps
  - bne, beq, j

Agenda

- Instructions For Data Transfer
- Instructions For Decisions
- **Administrivia**
- Misc: Immediates, Overflow, Inequalities
- **Break**
- Support for Strings
- C to MIPS Example
- Summary

Example: Fast String Copy Code in C

- `p`, `q` each pointers to strings. Copy string from p to q.

```
while((*q++ = *p++) != '\0') ;
```

Fast String Copy in MIPS Assembly

p and q assigned to $s0 and $s1 respectively

```
Loop:                                   # $t0 = *p
                                        # *q = $t0
                                        # p = p + 1
                                        # q = q + 1
                                        # if *p == 0, go to Exit
        j  Loop                         # go to Loop
Exit:   #  N characters => N*6 instructions
```

Fast String Copy in MIPS Assembly

p and q assigned to $s0 and $s1 respectively

```
Loop: lb $t0,0($s0)          # $t0 = *p
                             # *q = $t0
                             # p = p + 1
                             # q = q + 1
                             # if *p == 0, go to Exit
      j Loop                 # go to Loop
Exit: #  N characters => N*6 instructions
```

Fast String Copy in MIPS Assembly

p and q assigned to $s0 and $s1 respectively

```
Loop: lb $t0,0($s0)        # $t0 = *p
      sb $t0,0($s1)        # *q = $t0
                           # p = p + 1
                           # q = q + 1
                           # if *p == 0, go to Exit
      j Loop               # go to Loop
Exit: #  N characters => N*6 instructions
```

Fast String Copy in MIPS Assembly

p and q assigned to $s0 and $s1 respectively

```
Loop: lb $t0,0($s0)          # $t0 = *p
      sb $t0,0($s1)          # *q = $t0
      addi $s0,$s0,1         # p = p + 1
      addi $s1,$s1,1         # q = q + 1
                             # if *p == 0, go to Exit
      j Loop                 # go to Loop
Exit: #  N characters => N*6 instructions
```

Fast String Copy in MIPS Assembly

p and q assigned to $s0 and $s1 respectively

```
Loop: lb $t0,0($s0)          # $t0 = *p
      sb $t0,0($s1)          # *q = $t0
      addi $s0,$s0,1         # p = p + 1
      addi $s1,$s1,1         # q = q + 1
      beq $t0,$0,Exit        # if *p == 0, go to Exit
      j Loop                 # go to Loop
Exit: #  N characters => N*6 instructions
```

And in Conclusion, …

- Conditional instructions for making decisions, implementing control flow
  - `beq, bne`
  - combine with `slt` for inequalities
- Immediates let instructions work with constants.
- Special instructions (`lb,sb,lh,sh`) for handling strings.
- Three different notions of “unsigned” in MIPS