Instructions: Part 2
What have we seen?

- Machine instructions vs. Assembly
- The operation of various tools
  - Linkers/Loaders
  - Compilers/Assemblers
- Memory Representations
- A bit about instructions
Step back and look at the big picture...

• Focus on the what and the why
  – What a computer should do
    • Operations and functions
  – Why instructions are designed the way they are
    • Focus on MIPS
Principles of Computer Design

• Fundamental ideas behind RISC
  – Simplicity favors regularity
  – Smaller is faster
  – Good design demands good compromises
  – Make common case fast
Fundamental operations

• Arithmetic (obviously)
  – Addition, subtraction, multiplication, division
  – Logic: AND, OR, XOR

• Simplicity
  – Every arithmetic operation has 3 operands
    • Two operated on, One result
  – Make every instruction use 3 operands
And that works how?

- Consider $a = b + c$
- If we were to write this in assembly, it would be
  
  $\text{add a, b, c}$

- Three operands keeps the instruction logically simple
  
  $\text{mul b, x, y} \quad b = x \cdot y$
  $\text{addi a, b, 42} \quad a = b + 42$
More complex example

• Consider: $f = (g+h) - (i+j)$
  
  add $t_0$, $g$, $h$
  add $t_1$, $i$, $j$
  sub $f$, $t_0$, $t_1$

• Notes
  
  – Consider the operator precedence

  • ( ) before -

  – Pseudo code

  • Cannot use $g$ and $h$
What about the operands?

- Unlike C or Pascal, operands cannot be variables
  - Instead we keep them in registers
- Follows 1\textsuperscript{st} and 2\textsuperscript{nd} design principles
  - Simplicity follows regularity
  - Smaller is faster
- This forces the machine to only deal with two possible cases for arithmetic
  - Data in registers
  - Operand is encoded in the instruction
The Why of Registers

• As we saw...
  – Registers are efficient
  – Registers are *FAST*
    • Much faster than data over the bus from memory

• How do select the number of registers?
  – Must balance too many registers against the length of the clock cycle
How many operands?

• Real programs have many more operands than 32
  – Also there may be arrays or complex data structs
  – Can't fit everything into registers
• But so far, operations only work with registers
  – Need the concept of memory operations
Memory Representation

• Recall from last lecture…
  – Memory a big byte array
  – 0 is the first byte
  – Can access upto $2^{32} - 1$ bytes

• Byte vs Word Addressing
  – Access just a byte or a word in one operation
  – MIPS is byte-addressable
Memory Operations

• Load
  – Copies data *from* memory to a register
    lw $t0, 8($s0)
  – lw stands for Load Word
• Store
  – Stores data *to* memory from a register
    sw $t0, 8($s0)
Consider an example

- Example: \( g = h + A[8] \)
  
  - Assume
    - the address of \( A \) is located in \( s3 \)
    - \( h \) is associated with \( s1 \)
    - \( g \) is associated with \( s2 \)

\[
lw \ t0, 8(s3) \\
add \ s2, \ s1, \ t0
\]
Real Example

  – Even though we can access any byte, the array stores words
  – So $A[8]$ actually retrieves the SECOND element

```
lw $t0, 32($s3)
add $t0, $s1, $t0
sw $t0, 48($s3)
```
Now we must represent

• Instructions are data
  – A series of bits that can be interpreted
  – Can consider each part of an instruction a number

• Instructions are broken up into a series of fields
  – Each field is a different part of the instruction
Registers

- Heavily used, must have some mnemonic
  - Used in (just about) every instruction
  - To make it easier for us, we give each register a name
    - $s0..$s7
    - $t0..$t7
    - $zero
    - $gp
Load/Store

• How do we represent load/store operations?
  – Want to keep simplicity
  – R-Type would only allow for a 5 bit address
    • 32 different values
  – We need new instruction type
    • I-Type

\[
\text{l}w \; rt, \text{address}(rs)
\]

- Can access any address +/- \(2^{15}\) of rs
  • rt receives the data from memory
I-Type

- But this violates the simplicity of a single format
  - Principle 3: Good design demands good compromises

- Compromise
  - All instructions same length
  - The first 3 fields are all the same
    - Use op field to determine instruction format
Real world example

  
  \[
  \begin{align*}
  \text{lw} & \quad \text{\$t0, 1200($t1)} \\
  \text{add} & \quad \text{\$t0, \$s1, \$t0} \\
  \text{sw} & \quad \text{\$t0, 1200($t1)}
  \end{align*}
  \]

- Becomes the machine language instructions

  \[
  \begin{align*}
  100011 & \quad 01001 & \quad 01000 & \quad 0000 & \quad 0100 & \quad 1011 & \quad 0000 \\
  000000 & \quad 10010 & \quad 01000 & \quad 01000 & \quad 00000 & \quad 100000 \\
  101011 & \quad 01001 & \quad 01000 & \quad 0000 & \quad 0100 & \quad 1011 & \quad 0000
  \end{align*}
  \]
Reflect

• Have the tools to write a program that does a sequence of instructions
  – Calculate just about any expression
  – Store the result into memory

• Next step
  – Making decisions (branching)
Branches

• A computer is more than a very fast calculator
  – Often we want to change the flow of execution
    • At times we don't want to execute the “next” instruction

• This happens with a branch instruction
  
  \[
  \text{beq } \$t0, \$t1, \text{address} \\
  \text{bne } \$t0, \$t1, \text{address}
  \]
  – Checks $t0 against $t1 and then moves the execution to the given address
    • Address is the number of instructions to jump ahead from the current location
If...then

• If an expression is true, then do something

  \[ \text{if ( } i == j \text{ ) goto L1} \]
  \[ f = g + h \]
  \[ \text{L1: } f = f - i \]

• Becomes

  \[ \text{beq } s3, s4, \text{ L1} \]
  \[ \text{add } s0, s1, s2 \]
  \[ \text{L1: sub } s0, s0, s3 \]
What about if...then...else

- The Else is executed only if the condition is false

\[
\text{if ( } i == j \text{ ) } f = g + h \\
\text{else } f = g - h
\]

- Becomes

```
bne $s3, $s4, Else 
add $s0, $s1, $s2 
j Exit 
Else: sub $s0, $s1, $s2 
Exit: (some instructions)
```
Unconditional branch

• Doesn't check for a condition
  – Just jumps directly to the address
  – Format
    
    j address
  – Notice, it is (nearly) the same as
    
    beq $zero, $zero, address
Loops

• Sometimes we want to execute the same instructions a number of times
  – While loop
    
    ```
    while ( i != k )
      i = i + j
    ```
    
    Loop: beq $s1, $s3, Exit
    add $s1, $s1, $s2
    j Loop
    Exit: (some instructions)
Variable Indexing

• Loops are often used when initializing arrays
  – Often arrays accessed with a variable
    \[ A[i] \]
    • Where \( i \) is the index into the array.
    • Identifies the element to work with

• But that would be byte addressing, not word addressing
  – Must multiply \( i \) by 4
  – OR, add \( i \) to itself twice \( (i + i + i + i = i \times 4) \)
Accessing a variable index

• Consider A[i]
  – $s0 contains the address of A
  – $s1 contains i
    add $t0, $s1, $s1
    add $t0, $t0, $t0
    lw $t1, 0($t0)
Lets look at a loop with arrays

• Consider

\[
\text{while ( A[i] != k )}
\]
\[
i = i + j
\]

• Becomes

Loop: add $t0, $s1, $s1
add $t0, $t0, $t0
lw $t1, 0($t0)
beq $s1, $s3, Exit
add $s1, $s1, $s2
j Loop
Exit: (some instructions)
Non-equality

• Often in loops and if statements, we compare equality
  – beq, bne

• Situations require us to know if a variable is bigger than another
  – Is A > B
  – Use the slt instruction

    slt $t0, $s1, $s2
    slt dest, op1, op2

  – Checks if op1 < op2 and stores the result in dest
    • 1 if op1 < op2
    • 0 otherwise
slt, seq, sgt, gne

- Sets a given register with the results of a comparison
  - Can implement branching on less than, greater to
    - Less complexity
    - More instructions

```
slt $t0, $s1, $s2
bne $t0, $zero, address
```
Alternative Addressing Modes

• What about constants?
  – On average, 50% of a program is a constant
  – The instructions we have seen don't handle constants

• We introduce a new mode called *immediate*
  – It has the format of I-Type instructions
  – Each instruction has an alternative that uses intermediates
    • Intermediate is encoded as part of the instruction
Some examples

- Add a constant
  ```
  addi $t0, $s1, 42
  ```

- Multiply by a constant
  ```
  muli $t0, $s5, 4
  ```

- Compare against a constant
  ```
  slti $t0, $s1, 145
  ```
Encoding Constants

• By making this design tradeoff we see another design principle
  – Principle 4: Make the common case fast
• Constants in use ~50% of instructions
• The instructions reserve 16 bits
  – Can represent constants up to +/- $2^{15}$ in size
Branching modes

- With the branch instructions, we have 16 bits to specify the address
  - This is too small for today's programs
- Solution: offset based
  - We define a register that will always be added to the offset so that
    \[ \text{Program Counter} = \text{register} + \text{offset} \]
  - This works because most while and if..then branch less than 40 instructions away
    - Which register?
Branching distance

• Branching often occurs in a small distance
  – Program Counter contains the address of the current instruction
  – Using the PC would allows to branch to within $+/- 2^{15}-1$ addresses
    • Usually, more than enough
  – Called PC-Relative branching
But what about direct jumps

- No conditions to worry about
  - Discard two register fields from branch instructions
  - Give us 26 bits to use for jumps
  - Define a J-type instruction
    - Absolute address, someplace within $2^{26}$ bits

- Used to jump to procedures
  - Often, procedures have no reason to be close to the current address
Some final details

• Branching
  – All instructions are 4 bytes long
  – Branch address uses number of *words* to jump
    • Address of 4 says jump ahead 16 bytes

• Jumps
  – Also specify the word to jump to
    • 26 bits is really 28 bits
Branch out

- Branching a long way
  - What if we need to branch farther than 16 bits allows?
    - Wrap a jump instruction with a branch
      ```
      bne $t0, $s1, L2
      j L1
      ```
      ```
      L2: (some instructions)
      ```
Speaking of jumps...

- Jumps specify 28 bits
  - (because of word addressing)
  - Memory addresses are 32 bits
  - Where are the extra 4 bits?

- MIPS uses the top 4 bits of the PC
  - This means that the linker must be very careful
  - Cannot load the program across a 256 MB boundary