MIPS Introduction


The appendix is available as http://pages.cs.wisc.edu/~larus/HP_AppA.pdf from spim website.

Note: The on-line version is named Appendix A. It is the same as the 4th edition’s Appendix B.

- **Introduction**: Slide 2
- **Basic CPU Organization**: Slide 3
- **Memory Organization**: Slide 4
- **Registers vs. Memory**: Slide 6
- **MIPS Arithmetic**: Slide 9
- **Load and Store Instructions**: Slide 12
- **QtSpim and spim**: Slide 21
- **Printing in QtSpim and spim**: Slide 26
- **Program Flow Control**: Slide 33
- **Unconditional Branch**: Slide 33
- **Conditional Branch**: Slide 34
- **For Loops**: Slide 41
- **Other conditions for branching**: Slide 55
- **Constant or Immediate Operands**: Slide 56
- **Machine Language**: Slide 59
- **Logical Operations**: Slide 71
MIPS Introduction

- Language of the Machine.
- More primitive than higher level languages.
  - E.g., no sophisticated control flow such as `for` and `while`.
  - Only simple branch, jump, and jump subroutine.
- Very restrictive.
  - E.g., MIPS Arithmetic Instructions have: two operands, one result.
    - Can do in one instruction: \( a = b + c \)
    - Cannot do in one instruction: \( a = b + c \times d - e \)
- We will be working with the MIPS-32 instruction set architecture.
  - Similar to other architectures developed since the 1980's.
  - Used by (at various times) NEC, Nintendo, SGI (formerly Silicon Graphics Inc.), Sony
- Design goals of MIPS:
  - Maximize performance and minimize cost.
  - Reduce design time.
**Basic CPU Organization:**

- Simplified picture of a computer:

  ![CPU Organization Diagram](image)

- Three components:
  
  - Processor (or **Central Processing Unit** or CPU or “core”); MIPS in our case. Intel, PowerPC, UltraSparc, …
  
  - Memory — contains the program instructions to execute and the data for the program.
  
  - I/O Devices — how the computer communicates to the outside world. Keyboard, mouse, monitor, printer, game controller, tablet, etc.

- CPU contains three components:
  
  - Registers — hold data values for the CPU to manipulate.
  
  - Arithmetic Logic Unit (ALU) — performs arithmetic and logic functions. Takes values from and returns values to the registers.
  
  - Control — Determines what operation to perform, directs data flow to/from memory, directs data flow between registers and ALU.

  - Actions are determined by the current Instruction.
Memory Organization:

- Viewed as a large, one-dimensional array, with an address for each element — byte — of the array.

- A memory address is an index into the array.

- “Byte addressing” — the index points to a byte, 8 bits in today’s computers, of memory.

- MIPS addresses (up to) 4 Gigabytes of memory:
  - Bytes are numbered from 0 to $2^{32} - 1$; or 0 to 4,294,967,295.
  - Bytes are nice, but most data items use larger “words”.

<table>
<thead>
<tr>
<th>Address</th>
<th>8 bits of data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8 bits of data</td>
</tr>
<tr>
<td>1</td>
<td>8 bits of data</td>
</tr>
<tr>
<td>2</td>
<td>8 bits of data</td>
</tr>
<tr>
<td>3</td>
<td>8 bits of data</td>
</tr>
<tr>
<td>4</td>
<td>8 bits of data</td>
</tr>
<tr>
<td>5</td>
<td>8 bits of data</td>
</tr>
<tr>
<td>6</td>
<td>8 bits of data</td>
</tr>
<tr>
<td>...</td>
<td>8 bits of data</td>
</tr>
<tr>
<td>4,294,967,293</td>
<td>8 bits of data</td>
</tr>
<tr>
<td>4,294,967,294</td>
<td>8 bits of data</td>
</tr>
<tr>
<td>4,294,967,295</td>
<td>8 bits of data</td>
</tr>
</tbody>
</table>
Memory Organization (continued):

- For MIPS, a “word” is 32 bits, or 4 bytes.
  - Each register in the CPU holds 32 bits.
  - Not just a coincidence!
- $2^{32}$ bytes with byte addresses from 0 to $2^{32} - 1$.
- $2^{30}$ words. The words are at addresses:
  - 0, 4, 8, 12, 16, 20, 24, 28, ..., $2^{32} - 4$ (in decimal)
  - 0, 4, 8, C, 10, 14, 18, 1C, 20, 24, 28, 2C, ... (in hexadecimal)
- Words are “aligned”.
  - Each word starts on an address that is divisible by 4.
  - What are the least 2 significant bits of a word address in binary?
- Notes: If you have not already memorized these:
  - $2^{10} = 1,024_{\text{ten}} = 1$ Kilo = 1K
  - $2^{20} = 1$ Mega = 1M
  - $2^{30} = 1$ Giga = 1G
  - $2^{40} = 1$ Tera = 1T
**Registers vs. Memory:**

- Registers can be thought of as a type of memory. They are inside the CPU; thus, they are the “closest” memory.
- Registers provide a place to hold values inside the CPU, and allow a large set of operations to be performed on their values. I.e., add, subtract, compare, etc.
- Principal advantages of registers vs. memory:
  - Fast access.
  - Fast access.
  - Fast access.
- Principal advantages of memory vs. registers:
  - Lower cost.
  - Lower cost.
  - Lower cost.
- An intermediate type of memory: Cache.
  - Different “flavors” depending on size and physical location.
  - Level 1 cache “closest” to the CPU.
    - Usually installed on the chip as part of the CPU.
    - Typically small: 32K, 64K
  - Level 2 cache between the CPU and the memory.
    - Not part of one processor core, but is present on the chip. Shared by all processor cores (or not).
    - Typically a few Megabytes.
  - We will come back to the topic of cache later in the semester.
Registers vs. Memory (continued):

- Register Organization. (See also the SPIM Appendix, page 24).

<table>
<thead>
<tr>
<th>Name</th>
<th>Register number</th>
<th>Usage</th>
<th>Preserved on call?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>the constant value 0</td>
<td>n.a.</td>
</tr>
<tr>
<td>$at</td>
<td>1</td>
<td>reserved for the assembler</td>
<td>n.a.</td>
</tr>
<tr>
<td>$v0-$v1</td>
<td>2-3</td>
<td>values for results &amp; expression evaluation</td>
<td>no</td>
</tr>
<tr>
<td>$a0-$a3</td>
<td>4-7</td>
<td>arguments</td>
<td>yes</td>
</tr>
<tr>
<td>$t0-$t7</td>
<td>8-15</td>
<td>“temporary”; used for almost anything</td>
<td>no</td>
</tr>
<tr>
<td>$s0-$s7</td>
<td>16-23</td>
<td>“saved”; used for almost anything</td>
<td>yes</td>
</tr>
<tr>
<td>$t8-$t9</td>
<td>24-25</td>
<td>“temporary”; used for almost anything</td>
<td>no</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>global pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>stack pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>frame pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>return address</td>
<td>yes</td>
</tr>
</tbody>
</table>

- These are the “General Registers”. MIPS also has:
  - PC (program counter) register, Status register.
  - Floating-point registers.
Registers vs. Memory (continued):

- For now (programs 1, 2, 3, and 4) we will use:
  - \$zero, \$s0–\$s7, \$t0–\$t9 for writing programs.
    - \$zero always has the value 0; you cannot change its contents.
    - \$s0–\$s7 and \$t0–\$t9 can be used interchangeably in programs.
      - This will change later when we start using functions.
  - \$a0 and \$v0 for printing.
    - Only for printing!
    - Both \$a0 and \$v0 have other uses that will show up in functions.
  - Do NOT use any of the other registers:
    - \$at
    - \$v1
    - \$a1–\$a3
    - \$gp, \$sp, \$fp, \$ra
  - Some of the above will change when we start using functions (programs 5, 6, and 7).
**MIPS Arithmetic:**

- All arithmetic instructions have at most 3 operands.
- All arithmetic is done in registers!
  - Can not, for example, add a number directly to a value stored in memory. In MIPS, this requires 3 steps:
    - Load the value from memory into a register.
    - Add the number to the register.
    - Store the result in memory.
  - Thus, MIPS is a “load-store” architecture. All work other than loading and storing is done only in registers.
- Operand order is fixed.
  - Destination operand is first.
- Example:
  - C or Java code:
    ```
    xray = yoke + zebra;   $t0 = $t1 + $t2
    ```
  - MIPS code:
    ```
    add    $t0, $t1, $t2
    ```
    - adds contents of $t1 and $t2, placing the result in $t0.
MIPS Arithmetic (continued):

- Longer expressions require more instructions:
  - C or Java code (assume all variables are of type `int`):
    ```c
    oscar = papa + sierra + tango;
    foxtrot = foxtrot - oscar;
    ```
  - MIPS version:
    ```
    add $t0, $s1, $s2  # $t0 = papa + sierra; put result "temporarily" in $t0
    add $s0, $t0, $s3  # oscar = $t0 + tango; use the "temporary" result from $t0
    sub $s5, $s5, $s0  # $s5 = foxtrot - oscar
    ```
    - Note: use of `$t0` to hold “temporary” result.
    - Note: `#` marks the beginning of a comment that runs until the end of the current line
    - Note: operands must be registers.
  - We did not use variable names in the MIPS version, just register names.
    - Programmer has to “remember” which registers hold which variables.
      - Useful to have comments that say this!
      - Essential if you want a good grade on the programs.

```
$s1$ has papa
$s2$ has sierra
$s3$ has tango
$s5$ has foxtrot
```

```
add $s3, $s3, $s3
```
MIPS Arithmetic (continued):

- Registers vs. Memory:
  - Arithmetic instructions — operands must be registers.
  - There are only 32 registers available.
  - Compiler for a language, such as C, will associate variables with registers automatically.
- What about programs with more variables than there are registers?
  - This covers just about every program!
  - Must move the values of the variables between memory and registers.
Load and Store Instructions:

- MIPS uses load instructions to copy a value from memory to a register. There are three versions, depending on the amount of data being copied:
  - lw will copy a word (32-bits).
  - lh will copy half a word (16-bits).
  - lb will copy one byte (8-bits).
- MIPS uses store instructions to copy a value from a register to a memory location.
  - sw will copy a word (32-bits).
  - sh will copy half a word (16-bits).
  - sb will copy one byte (8-bits).
Load and Store Instructions (continued):

- MIPS allows us to use symbolic names for memory locations.
  - Saves having to use binary or hexadecimal addresses.
    - bats is easier to remember and understand than the memory address \texttt{0x1000 0010}, for example.
  - In a MIPS assembly program, we can assign symbolic names to memory locations using \texttt{.word}, \texttt{.half}, and \texttt{.byte}, as appropriate:

```assembly
bats:   .word  17    # creates a 32-bit integer w/ the initial value 17, named bats
balls:  .word   3    # creates a 32-bit integer w/ the initial value  3, named balls
```
Load and Store Instructions (continued):

- We can create as many variables, with initial values, as we need for a program. I.e.,

  ```
  bats: .word 17
  balls: .word 3
  gloves: .word 10
  bases: .word 4
  ```

- Suppose we want to perform some arithmetic on these values? To take a specific example:

  - Want to find the total of the `bats + balls + gloves`.

- To do arithmetic, we have to get the values from memory into registers. Thus, we need to put the values for `bats`, `balls`, and `gloves` into three registers.

- Where in memory?

  - The names `bats`, `balls`, etc. above are symbolic names that represent locations in memory.
    - Note: `bats` is a location, not a value.

  - To load a value from a location, we use the `lw` instruction (load word). `lw` has the form:

    ```
    lw   destination-register, offset(register-with-address)
    ```

    - The destination-register can be any of `$s0-$s7`, `$t0-$t9`.
    - The register-with-address can also be any of these registers.
    - How does the register-with-address get its value?
Load and Store Instructions (continued):

- Getting an address into a register:
  - The `la` instruction (load address) will put the address associated with a symbolic name into a register. I.e.,
    
    ```
    la   $t0, bats   # puts the address of the bats memory location into register $t0
    ```
  - Once the address is in a register, we can use that register for the register-with-address part of an `lw` instruction.
    
    ```
    lw  $s0, 0($t0)     # gets value located at address stored in $t0 & puts it into $s0
    ```
  - These two instructions (`la` and `lw`) together will put the number stored at location `bats` into register `$s0`.
  - For now, we will use 0 for the offset every time.

- Now, we can write the code to find the sum: `bats + balls + gloves`:
  
  ```
  la   $t0, bats       # $t0 now contains the address of the bats memory location
  lw   $s0, 0($t0)     # $s0 now contains the number of bats
  la   $t0, balls      # $t0 now contains the address of the balls memory location
  lw   $s1, 0($t0)     # $s1 now contains the number of balls
  la   $t0, gloves     # $t0 now contains the address of the gloves memory location
  lw   $s2, 0($t0)     # $s2 now contains the number of gloves
  add  $s3, $s0, $s1   # $s3 now contains the number of bats + balls
  add  $s3, $s3, $s2   # $s3 now contains the number of bats + balls + gloves
  ```

- Our answer is now in register `$s3`.

- How do we get the answer from register `$s3` into a memory location?
Load and Store Instructions (continued):

- Copying a value from a register to a memory location is done with store (from slide #11):
  - `sw` will copy a word (32-bits).
  - `sh` will copy half a word (16-bits).
  - `sb` will copy one byte (8-bits).

- Concentrating on `sw` for now.

- `sw` has a similar format to the `lw` command:
  
  `sw  source-register, offset(register-with-address)`

- Again, we need an address (a location) in memory. We get this using the `la` command.

- Need to declare a location to put our result. Will add this to the memory declarations for `bats`, `balls`, etc.

- Two step process to save a value:
  - Get the address of the location (using `la`).
  - Save the value into that location (using `sw`).
Load and Store Instructions (continued):

- A short (not yet complete) MIPS program, named *bats1.s*, to find: \( \text{sum} = \text{bats} + \text{balls} + \text{gloves} \)

```
.data
bats: .word 17
balls: .word 3
gloves: .word 10
bases: .word 4
sum: .word 0  # Create a place to put our answer

.text
main:
la $t0, bats  # $t0 has the address of the bats memory location
lw $s0, 0($t0)  # $s0 now holds the number of bats
la $t0, balls  # $t0 has the address of the balls memory location
lw $s1, 0($t0)  # $s1 now holds the number of balls
la $t0, gloves  # $t0 has the address of the gloves memory location
lw $s2, 0($t0)  # $s2 now holds the number of gloves
add $s3, $s0, $s1  # $s3 now holds the number of bats+balls
add $s3, $s3, $s2  # $s3 now holds the number of bats+balls+gloves
la $t0, sum  # $t0 has the address of the sum memory location
sw $s3, 0($t0)  # sum now holds the number of bats+balls+gloves
```

- **.data** marks a part of the program that defines memory locations.
- **.text** marks a part of the program that contains assembly instructions (the program).

- **main:** is a required label to tell the *spim* simulator where to begin executing.
Load and Store Instructions (continued):

- *bats2.s*: A complete version of the bats program. `main` is now a function that includes the proper way to start and end a function. For now, just put these lines into every `main` that you write, with the body of `main` in-between. We will cover later what these lines are actually doing, and why they are needed.

```assembly
.data
bats:    .word 17
balls:   .word 3
gloves: .word 10
bases:   .word 4
sum:     .word 0    # Create a place to put our answer
.text
main:    # Function prologue -- even main has one
    addiu $sp, $sp, -24   # allocate stack space -- default of 24 here
    sw    $fp, 0($sp)     # save caller's frame pointer
    sw    $ra, 4($sp)     # save return address
    addiu $fp, $sp, 20    # setup main's frame pointer
    la    $t0, bats       # $t0 has address of the bats memory location
    lw    $s0, 0($t0)     # $s0 now holds the number of bats
    la    $t0, balls      # $t0 has address of the balls memory location
    lw    $s1, 0($t0)     # $s1 now holds the number of balls
    la    $t0, gloves     # $t0 has address of the gloves memory location
    lw    $s2, 0($t0)     # $s2 now holds the number of gloves
    add   $s3, $s0, $s1   # $s3 now holds the number of bats+balls
    add   $s3, $s3, $s2   # $s3 now holds the number of bats+balls+gloves
```
Load and Store Instructions (continued):

• bats2.s, continued:

```
la    $t0, sum        # $t0 has address of the sum memory location
sw    $s3, 0($t0)     # sum now holds the number of bats+balls+gloves

done:    # Epilogue for main -- restore stack & frame pointers and return
lw    $ra, 4($sp)     # get return address from stack
lw    $fp, 0($sp)     # restore the caller's frame pointer
addiu $sp, $sp, 24    # restore the caller's stack pointer
jr     $ra             # return to caller's code
```
Load and Store Instructions (continued):

- You can find the bats1.s and bats2.s examples from the previous 3 slides on D2L
  - We will be adding to this list of examples!
  - You can execute this program using:
    - QtSpim
      - Your own Windows/Linux/MacOSX machine.
      - lectura, Ubuntu machines in GS-930 and Mac’s in GS-228.
- You will find, when you execute either program, that it runs to completion.
  - From bats1.s, you get an error:
    - Neither version prints anything!
      - We put the answer in a register and in memory.
      - We did not print it!
QtSpim and spim

  - Appendix B in the 4th edition, and Appendix A from the spim web site are the same.
  - I refer to this as the “Spim Appendix”.
  - Especially read section 9, pages 40 to 45.
  - Appendix A can also be found on the web at: http://www.cs.wisc.edu/~larus/HP_AppA.pdf
    - There is a link to this on-line version on the 252 class web page and in D2L.
- There are “Getting Started” guides for the command-line version of spim:
  - You can also find these links on the 252 D2L page. Click on “Useful web links” under Content.
QtSpim and spim (continued):

- spim is a command-line tool usable on a UNIX system (Linux/OS X, among others).
- QtSpim is a GUI interface to spim that is usable on Windows/Linux/OSX.
- QtSpim displays:
  - Text, a scrollable window showing your code.
  - In Regs[16]: A scrollable window that shows:
    - Program Counter (PC) and status registers, among others.
    - General Registers: The 32 general-purpose registers.
  - Data: A scrollable window that shows:
    - User data segment (useful now).
    - User stack (will use this later).
  - FP Regs: A scrollable window that shows:
    - Floating Point Registers, both single- and double-precision registers
QtSpim and spim (continued):

- Register Display:
  - After \textit{bats2.s} program has finished.
  - All register contents are displayed in \textit{hexadecimal}.

\begin{itemize}
  \item \texttt{$t0$} has the hex address of sum
  \item \texttt{$s0$} has bats
  \item \texttt{$s1$} has balls
  \item \texttt{$s2$} has gloves
  \item \texttt{$s3$} has bats+balls+gloves
\end{itemize}

\begin{tabular}{ll}
PC & 400020 \\
EPC & 0 \\
Cause & 0 \\
BadVAddr & 0 \\
Status & 3000ff10 \\
HI & 0 \\
LO & 0 \\
R0 [r0] & 0 \\
R1 [at] & 10010000 \\
R2 [v0] & a \\
R3 [v1] & 0 \\
R4 [a0] & 1 \\
R5 [a1] & 7fffffff \\
R6 [a2] & 7fffffff \\
R7 [a3] & 0 \\
R8 [t0] & 10010010 \\
R9 [t1] & 0 \\
R10 [t2] & 0 \\
R11 [t3] & 0 \\
R12 [t4] & 0 \\
R13 [t5] & 0 \\
R14 [t6] & 0 \\
R15 [t7] & 0 \\
R16 [s0] & 11 \\
R17 [s1] & 3 \\
R18 [s2] & a \\
R19 [s3] & 1e \\
R20 [s4] & 0 \\
R21 [s5] & 0 \\
R22 [s6] & 0 \\
\end{tabular}
QtSpim and spim (continued):

- Displays the program code.

Address of instruction hexadecimal

MIPS machine language 32-bit, hexadecimal

Line number from your .s file

<table>
<thead>
<tr>
<th>Address</th>
<th>Hexadecimal</th>
<th>MIPS Machine Language</th>
<th>Line number from your .s file</th>
</tr>
</thead>
<tbody>
<tr>
<td>004000024</td>
<td>27bdff6</td>
<td>addiu $29, $29, -24</td>
<td>13: addiu $sp, $sp, -24 # allocate stack space -- default of 24 here</td>
</tr>
<tr>
<td>004000028</td>
<td>afebe000</td>
<td>sw $30, 0($29)</td>
<td>14: sw $fp, 0($sp) # save caller's frame pointer</td>
</tr>
<tr>
<td>004000030</td>
<td>27be0014</td>
<td>sw $31, 4($29)</td>
<td>15: sw $ra, 4($sp) # save return address</td>
</tr>
<tr>
<td>004000038</td>
<td>3c081001</td>
<td>addiu $30, $29, 20</td>
<td>16: addiu $fp, $sp, 20 # setup main's frame pointer</td>
</tr>
<tr>
<td>00400034</td>
<td>8d100000</td>
<td>lw $16, 0($8)</td>
<td>18: la $t0, bats # $t0 has address of the bats memory location</td>
</tr>
<tr>
<td>0040003c</td>
<td>3c011001</td>
<td>lui $1, 4097 [bats]</td>
<td>19: lw $s0, 0($t0) # $s0 now holds the number of bats</td>
</tr>
<tr>
<td>00400040</td>
<td>34280004</td>
<td>lui $1, 4097 [balls]</td>
<td>20: la $t0, balls # $t0 has address of the balls memory location</td>
</tr>
<tr>
<td>00400048</td>
<td>8d110000</td>
<td>ori $8, $1, 4 [balls]</td>
<td>21: lw $s1, 0($t0) # $s1 now holds the number of balls</td>
</tr>
<tr>
<td>0040004c</td>
<td>3c011001</td>
<td>ori $8, $1, 8 [gloves]</td>
<td>22: la $t0, gloves # $t0 has address of the gloves memory location</td>
</tr>
<tr>
<td>00400050</td>
<td>8d120000</td>
<td>lw $18, 0($8)</td>
<td>23: lw $s2, 0($t0) # $s2 now holds the number of gloves</td>
</tr>
<tr>
<td>00400054</td>
<td>02119820</td>
<td>add $19, $16, $17</td>
<td>24: add $s3, $s0, $s1 # $s3 now holds the number of balls+bats</td>
</tr>
<tr>
<td>00400058</td>
<td>02729820</td>
<td>add $19, $19, $18</td>
<td>25: add $s3, $s3, $s2 # $s3 now holds the number of balls+bats+gloves</td>
</tr>
<tr>
<td>0040005c</td>
<td>3c011001</td>
<td>lui $1, 4097 [sum]</td>
<td>27: la $t0, sum # $t0 has address of the sum memory location</td>
</tr>
<tr>
<td>00400060</td>
<td>34280010</td>
<td>ori $8, $1, 16 [sum]</td>
<td>28: sw $s3, 0($t0) # sum now holds the number of balls+bats+gloves</td>
</tr>
<tr>
<td>00400064</td>
<td>ad130000</td>
<td>sw $19, 0($8)</td>
<td>31: lw $ra, 4($sp) # get return address from stack</td>
</tr>
<tr>
<td>00400068</td>
<td>8fb0004</td>
<td>lw $31, 4($29)</td>
<td>32: lw $fp, 0($sp) # restore the caller's frame pointer</td>
</tr>
<tr>
<td>0040006c</td>
<td>8fbe0000</td>
<td>lw $30, 0($29)</td>
<td>33: addiu $sp, $sp, 24 # restore the caller's stack pointer</td>
</tr>
<tr>
<td>00400070</td>
<td>27bd0018</td>
<td>addiu $29, $29, 24</td>
<td>34: jr $ra # return to caller's code</td>
</tr>
<tr>
<td>00400074</td>
<td>03e00008</td>
<td>jr $31</td>
<td>24</td>
</tr>
</tbody>
</table>

Assembly language instruction, from your .s file, before assembly

Assembly language instruction, after assembly
QtSpim and spim (continued):

- Data Segment:
  - Shows the contents of memory. This snapshot was taken after bats2.s stopped executing.

<table>
<thead>
<tr>
<th>Addresses of bytes, hexadecimal</th>
<th>Contents of bytes, hexadecimal, grouped into words</th>
</tr>
</thead>
<tbody>
<tr>
<td>User data segment</td>
<td>bats</td>
</tr>
<tr>
<td>[10000000]..[10040000]</td>
<td>00000000</td>
</tr>
<tr>
<td>[10010010]</td>
<td>00000000</td>
</tr>
<tr>
<td>[10010020]..[10040000]</td>
<td>00000000</td>
</tr>
</tbody>
</table>

User Stack: ignore (for now)
Printing in QtSpim and spim:

- Need to connect to an outside device (“outside” the CPU).
- The simulator provides this for us via the System Call mechanism.
- There are 17 system calls. See Figure 9.1 on page 44 of the SPIM Appendix for the complete list.
- Here are some of the more useful ones (there are others you will need; look them up!):

<table>
<thead>
<tr>
<th>Service</th>
<th>System call code</th>
<th>Arguments</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>print_int</td>
<td>1</td>
<td>$a0 = integer</td>
<td></td>
</tr>
<tr>
<td>print_float</td>
<td>2</td>
<td>$f12 = float</td>
<td></td>
</tr>
<tr>
<td>print_double</td>
<td>3</td>
<td>$f12 = double</td>
<td></td>
</tr>
<tr>
<td>print_string</td>
<td>4</td>
<td>$a0 = string</td>
<td></td>
</tr>
<tr>
<td>read_int</td>
<td>5</td>
<td>integer (in $v0)</td>
<td></td>
</tr>
<tr>
<td>read_float</td>
<td>6</td>
<td>float (in $f0)</td>
<td></td>
</tr>
<tr>
<td>read_double</td>
<td>7</td>
<td>double (in $f0)</td>
<td></td>
</tr>
<tr>
<td>read_string</td>
<td>8</td>
<td>$a0 = buffer, $a1 = length</td>
<td></td>
</tr>
</tbody>
</table>
Printing in QtSpim and spim (continued):

Printing Strings:

- We can declare character **arrays** using `.ascii`.
  
  - This is done in the `.data` section.
  
  - For example,

    ```
    myChars:   .ascii  "abc7 d8-#$stuff8y wi"
    gibberish: .ascii  "982(*&$junk(*&stuff"
    words:     .ascii  "the of and or nor a an alphabet Zulu"
    ```

  - A **string** is a character array with an additional character on the end. This last character is always the **null** character (ascii value zero).

    - Declare a string using `.asciiz`. Examples:

      ```
      myName:   .asciiz   "Patrick T. Homer"
      morning:  .asciiz   "Good morning"
      week:     .asciiz   "Su Mo Tu We Th Fr Sa"
      ```
Printing in QtSpim and spim (continued):

Printing Strings (continued):

- Declare one (or more) strings in the `.data` segment using `.asciiz`:
  
  morning: .asciiz "Good morning"
  myName: .asciiz "Patrick T. Homer"
  week: .asciiz "Su Mo Tu We Th Fr Sa"

- Load the address of the string into $a0:
  
  la $a0, myName  # put the address of my name in register $a0

- Put the system call number for `print_string` into $v0.

  - Need a new instruction: `add immediate`:
    
    - Like `add`, but 3rd register is replaced with a positive or negative integer:
      
      addi $v0, $zero, 4  # put system call number 4 into register $v0

- Call system:
  
  syscall  # spim will now print the string that has my name
Printing in QtSpim and spim (continued):

- **HelloWorld1.s**

  # Print the Hello World phrase.
  # Here we load the base address of the string
  # into register $a0, and use the print_string
  # syscall to print the phrase.

  .data
  hello:  .asciiz "Hello World\n"

  .text
  main:    # Function prologue -- even main has one
           addiu $sp, $sp, -24   # allocate stack space -- default of 24 here
           sw    $fp, 0($sp)     # save caller's frame pointer
           sw    $ra, 4($sp)     # save return address
           addiu $fp, $sp, 20    # setup main's frame pointer

           # set up $a0 to hold address of the hello world string
           # then print the string
           la      $a0, hello    # Point to the string
           addi    $v0, $zero, 4 # syscall value for print_string
           syscall

  done:    # Epilogue for main -- restore stack & frame pointers and return
           lw    $ra, 4($sp)     # get return address from stack
           lw    $fp, 0($sp)     # restore the caller's frame pointer
           addiu $sp, $sp, 24    # restore the caller's stack pointer
           jr     $ra             # return to caller's code
Printing in QtSpim and spim (continued):

- **HelloWorld1.s**: Use the step command to advance to the `syscall` on line 25.

  ```assembly
  PC = 40003c
  EPC = 0
  Cause = 0
  BadVAddr = 0
  Status = 3000ff10
  HI = 0
  LO = 0
  R0 [r0] = 0
  R1 [at] = 0
  R2 [v0] = 4
  R3 [v1] = 0
  R4 [a0] = 10010000
  R5 [a1] = 7fffffff
  R6 [a2] = 7fffffff
  R7 [a3] = 0
  R8 [t0] = 0
  R9 [t1] = 0
  R10 [t2] = 0
  ```

  **print_str** system call

  Address of hello string

  User data segment [10000000]..[10040000]
  [10000000]..[10010000] 00000000[10010000] 6c6c6548 6f57206f 0a646c72 00000000 Hello
  World
  [10010010]..[10040000] 00000000

  User Stack [7fffffff]..[80000000]
  [7fffffff] 00000001 7fffffff
  [7fffffff] 00000000 7fffffff
  [7fffffff] 7fffffff4
  [7fffffff] 7fffffff94 00000000
  [7fffffff] 7fffffff50 7fffffff80 7fffffff73 7fffffff63 7fffffff40 $ s c @
Printing in QtSpim and spim (continued):

- **HelloWorld2.s**: Can break the string into multiple parts:

```assembly
.data
hello:   .asciiz "Hello"
space:   .asciiz " "
world:   .asciiz "World"
newline: .asciiz "\n"
.text
# set up and print the string "Hello"
la      $a0, hello      # Point to the string
addi    $v0, $zero, 4   # syscall value for print_string
syscall
# set up and print the string " 
"  
la      $a0, space      # Point to the string
addi    $v0, $zero, 4   # syscall value for print_string
syscall
# set up and print the string "World"
la      $a0, world      # Point to the string
addi    $v0, $zero, 4   # syscall value for print_string
syscall
# set up and print the string "\n"
la      $a0, newline    # Point to the string
addi    $v0, $zero, 4   # syscall value for print_string
syscall
```

**Note**: The prologue and epilogue of main are missing here. They are present in the `HelloWorld2.s` example that is on D2L.
Printing in QtSpim and spim (continued):

- **HelloWorld2.s**: Can break the string into multiple parts:

```
.data

hello: .asciiz "Hello"
space: .asciiz " 
world: .asciiz "World"
newline: .asciiz "\n"
```

```
User data segment [10000000].1[10040000]
[10000000].1[10010000] 00000000[10010000]
[10010010].1[10040000] 00000000

User Stack [7ffffffe38].[80000000]
[7ffffffe38] 00000001 7ffffff6
[7ffffffe40] 00000000 7ffffffd 7fffffff4 7fffffff94
[7ffffffe50] 7fffffff0 7fffffff3 7fffffff63 7fffffff40
```

```
1 1 e H \0 \0 o l r o W \0 \n \0 d
```

```
6c6548 002006f 6c726f57 000a064 Hello World
```
Program Flow Control

• Assembly language supports a much smaller set of flow control instructions compared to high-level programming languages:
  
  • What assembly language does not have:
    • `for, while, do...while, switch`
  
  • What assembly language does have:
    • Unconditional branch.
    • Conditional branch.
    • Function calls.

**Unconditional branch.**

The jump instruction:

```
  j    label
```

• Example:

```
  add  $t0, $s1, $s2
  j    toThere
  add  $t0, $t0, $s1  # This instruction is skipped
  add  $s1, $s2, $s2  # This instruction is skipped
  toThere:  add  $s7, $s1, $s3
```
Conditional branch.

• Chooses between two control flows. Either
  • Execute the next instruction, or
  • Jump to a different instruction.

• MIPS has two conditional branch instructions:
  • Branch if equal:
    • `beq` register, register, Label
  • Branch if not equal:
    • `bne` register, register, Label

• Example:

```assembly
if ( i == j )
    bne $s0, $s1, downThere  # if (i != j)
    h = i + j;
    add $s3, $s0, $s1        # h = i + j
    z = h + h;             downThere: add $s4, $s3, $s3         # z = h + h
```

• Note: reversal of the condition from equality to inequality!
  • This is a common technique.
Conditional branch (continued):

- Example, version 1:

  if ( i != j )
      h = i + j;
  j Lab2
Lab1:
      add $s3, $s4, $s5  # h = i + j
      Lab3:
else
    h = i - j;
Lab2:
    sub $s3, $s4, $s5  # h = i - j
Lab3:
    add $s6, $s3, $4   # k = h + i

- The statement labeled Lab1: is executed **only** when $s4$ is **not** equal to $s5$.
- The statement labeled Lab2: is executed only when $s4$ is equal to $s5$.
- The statement labeled Lab3: is executed **always**.
- Need to “skip the true part” if the bne is false; hence the need for j Lab2.
- Confusing structure, prone to errors.
Conditional branch (continued):

• Example, version 2:

```assembly
if ( i != j )
    h = i + j;
    add $s3, $s4, $s5 # h = i + j
    j Lab2 # skip false part

else
    h = i - j;
    sub $s3, $s4, $s5 # h = i - j

k = h + i; # skip true part
```

• The reversed condition makes this easier to read, and less prone to errors.
  • Allows the assembly code to more closely follow the pattern from the higher-level language.

• The statement labeled `Lab1:` is executed only when $s4$ is equal to $s5$.

• The statement labeled `Lab2:` is executed always.
Conditional branch (continued):

- Comparisons other than equal or not equal?
- New instruction: **Set Less Than**.
  - Compares two registers and puts the result in the destination register:
    \[
    \text{slt} \hspace{1em} \$t0, \hspace{1em} \$s4, \hspace{1em} \$s5
    \]
  - First operand, \$t0 in this case, is the destination of the result of the comparison.
    - Note: this follows the pattern of all the MIPS arithmetic instructions.
  - Second and third operands are compared: \$s4 < \$s5
  - Result is 1 if the comparison is true.
  - Result is 0 if the comparison is false.
  - Example:

    \[
    \text{if ( } x < y \hspace{1em} \text{) } \begin{align*}
    z &= x; \\
    \text{else } \begin{align*}
    z &= y; \\
    w &= z \times 2;
    \end{align*}
    \end{align*}
    \]

    \[
    \begin{align*}
    \text{slt} \hspace{1em} \$t0, \hspace{1em} \$s2, \hspace{1em} \$s3 & \hspace{1em} \# \hspace{1em} \$t0 = (x < y) \\
    \text{beq} \hspace{1em} \$t0, \hspace{1em} \$zero, \hspace{1em} \text{yLess} & \hspace{1em} \# \hspace{1em} \text{if } z = x \\
    \text{add} \hspace{1em} \$s5, \hspace{1em} \$s2, \hspace{1em} \$zero & \hspace{1em} \# \hspace{1em} z = x \\
    \text{add} \hspace{1em} \$s5, \hspace{1em} \$s3, \hspace{1em} \$zero & \hspace{1em} \# \hspace{1em} z = y \\
    \text{add} \hspace{1em} \$s6, \hspace{1em} \$s5, \hspace{1em} \$s5 & \hspace{1em} \# \hspace{1em} w = z \times 2
    \end{align*}
    \]
# Basic idea:

# if ( $s3 ?? $s5 ) goto Athos
slt $t0, _____, _____
b__ $t0, $zero, Athos

# less than:
# if ($s5 < $t7) goto otherPlace

slt $t0, $s5, $t7
bne $t0, $zero, otherPlace

# greater than:
# reverse the order
# if ( $s4 > $s1 ) goto Batman

# becomes
# if ( $s1 < $s4 ) goto Batman

slt $t2, $s1, $s4
bne $t2, $zero, Batman
# Basic idea:

```assembly
# if ( $s3 >= $s5 ) goto Porthos
beq $s3, $s5, Porthos

# Then, we have:
# if ( $s3 > $s5 ) goto Porthos
# reverse the order to get
# if ( $s5 < $s3 ) goto Porthos
slt $t0, $s5, $s3
bne $t0, $zero, Porthos
```

# less than or equal to:
# reverse the meaning, use greater than

```assembly
# if ( $s3 <= $s2 ) goto Zorro
beq $s3, $s2, Zorro

# Two parts:
# First: if ( $s3 == $s2 ) goto Zorro
beq $s3, $s2, Zorro

# Second: if ( $s3 < $s2 ) goto Zorro
slt $t2, $s3, $s2
bne $t2, $zero, Zorro
```

# Alternative approach when =’s is present:

```assembly
# if ( $s3 <= $s2 ) goto Zorro

# Two parts:
# First: if ( $s3 == $s2 ) goto Zorro
beq $s3, $s2, Zorro

# Second: if ( $s3 < $s2 ) goto Zorro
slt $t2, $s3, $s2
bne $t2, $zero, Zorro
```

# Can use a similar approach for >=
# But, this can be confusing(!!)
# and prone to errors
- MIPS Instruction summary to date:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $s1, $s2, $s3</td>
<td>$s1 = $s2 + $s3</td>
</tr>
<tr>
<td>sub $s1, $s2, $s3</td>
<td>$s1 = $s2 - $s3</td>
</tr>
<tr>
<td>addi $t0, $s3, Number</td>
<td>add immediate: $t0 = $s3 + Number</td>
</tr>
<tr>
<td>la $s2, label</td>
<td>$s2 = address of label,  <em>pseudo-instruction</em></td>
</tr>
<tr>
<td>lw $s1, 0($s2) or lb or lh</td>
<td>$s1 = Memory[$s2], load word in Memory at position $s2 into $s1</td>
</tr>
<tr>
<td>sw $s1, 0($s2) or sb or sh</td>
<td>Memory[$s2] = $s1, store word in $s1 into Memory at position $s2</td>
</tr>
<tr>
<td>slt $t0, $s3, $s4</td>
<td>$t0 = $s3 &lt; $s4, put 1 in $t0 if true, else put 0 in $t0</td>
</tr>
<tr>
<td>bne $s4, $s5, aLabel</td>
<td>Next instruction executed is at aLabel if $s4 ≠ $s5</td>
</tr>
<tr>
<td>beq $s4, $s5, bLabel</td>
<td>Next instruction executed is at bLabel if $s4 == $s5</td>
</tr>
<tr>
<td>j cLabel</td>
<td>Next instruction executed is at cLabel</td>
</tr>
</tbody>
</table>
For Loops.

- C (or Java) code:
  ```c
  sum = 0;
  for (i = 0; i < y; i++)
      sum = sum + x;
  ```

- MIPS:
  - Assume: $s0$ has $x$, $s1$ has $y$, and $s2$ (will have) $sum$.
  - Will use $t0$ for $i$.

  ```mips
  add $s2, $zero, $zero  # sum = 0
  addi $t0, $zero, 0     # i = 0, initial value set before loop begins
  
  LoopBegin:
  # for loop does comparison at beginning of each iteration
  slt $t2, $t0, $s1       # is i < y ??
  beq $t2, $zero, LoopEnd # branch below end of loop if done
  
  # loop body
  add $s2, $s2, $s0       # sum = sum + x
  
  # increment loop index
  addi $t0, $t0, 1        # i++
  
  j LoopBegin
  
  LoopEnd:
  
  # rest of program goes here...
  ```
For Loops (continued):

- Complete MIPS program: for loop example. Available as for1.s on D2L.

```mips
.data
x:       .word  42
y:       .word  8
sum:     .word  0
answer:  .asciiz "The sum is "
newline:  .asciiz "\n"

.text
main:    # Function prologue -- even main has one
    addiu $sp, $sp, -24       # allocate stack space -- default of 24 here
    sw    $fp, 0($sp)          # save caller's frame pointer
    sw    $ra, 4($sp)          # save return address
    addiu $fp, $sp, 20         # setup main's frame pointer
    # Put x into $s0
    la    $t0, x
    lw    $s0, 0($t0)
    # Put y into $s1
    la    $t0, y
    lw    $s1, 0($t0)
    add   $s2, $zero, $zero    # sum = 0
    add   $t0, $zero, $zero    # i = 0
```


For Loops (continued):

LoopBegin:
# for loop does comparison at beginning of each iteration
slt $t2, $t0, $s1       # is i < y ??
beq $t2, $zero, LoopEnd # branch below end of loop if done

# loop body
add $s2, $s2, $s0       # sum = sum + x

# increment loop index
addi $t0, $t0, 1        # i++

j LoopBegin

LoopEnd:
# Print message
la $a0, answer
addi $v0, $zero, 4
syscall

# Print the sum
add $a0, $s2, $zero
addi $v0, $zero, 1
syscall

# Print newline
la $a0, newline
addi $v0, $zero, 4
syscall
For Loops (continued):

done:    # Epilogue for main -- restore stack & frame pointers and return
lw    $ra, 4($sp)         # get return address from stack
lw    $fp, 0($sp)         # restore the caller's frame pointer
addiu $sp, $sp, 24        # restore the caller's stack pointer
jr    $ra                 # return to caller's code
For Loops (continued):

- Print the integers in an array where the number of elements in the array is known.
- First problem to understand: How to represent an array in MIPS?
  - Handled by declaring memory in the `.data` section.
  - The label `numElements` tells us how many values are stored in the array.
  - The array has only one label, which is the name of the array.
    - The label identifies the beginning of the array.
  - Example:
    ```
    .data
    numElements:
      .word 7

    elements:
      .word 55
      .word 66
      .word 77
      .word 0
      .word -16
      .word -19
      .word 82
    ```

- Arrays in assembly differ from arrays in Java in two very important ways:
  - Nothing marks the end of an array.
  - Nothing indicates how many elements are in the array.
  - Sets up an array:
    - Named `elements`.
    - 7 locations, each of size 4 bytes (a `word`).
    - Each position has an initial value.
For Loops (continued):

- We need:
  - A register to hold the loop index, \( i \).
  - A register to hold the number of elements in the array.
  - A register to hold the starting address of the array.

```
addi  $s1, $zero, 0     # $s1 = i = 0
la    $t0, numElements
lw    $s2, 0($t0)       # $s2 = numElements
la    $t0, elements     # $t0 = address of elements[0]
```

- To get one element of the array:
  - Compute the offset (number of bytes) from the beginning of the array to the desired integer.
  - Each integer is 4 bytes (one word).
  - Can multiply \( i \) by 4, then add to the starting address of the array:

```
add    $t1, $s1, $s1
add    $t1, $t1, $t1     # $t1 = 4 * i
add    $t2, $t0, $t1     # $t2 = address of elements[i]
lw     $a0, 0($t2)        # $a0 = elements[i]
```

- Note: it is faster to do two `add`’s than to do one multiply.
For Loops (continued):

- Complete MIPS program: array print example. Available as for2.s on D2L.

  # Print the values of an array using a for loop.
  .data
  numElements:
    .word 7

  elements:
    .word 55
    .word 66
    .word 77
    .word 0
    .word -16
    .word -19
    .word 82

  newline:
    .asciiz "\n"

  .text
  main:
    # Function prologue -- even main has one
    subu $sp, $sp, 24    # allocate stack space -- default of 24 here
    sw $fp, 0($sp)      # save caller's frame pointer
    sw $ra, 4($sp)      # save return address
    addiu $fp, $sp, 20  # setup main's frame pointer
For Loops (continued):

- Complete MIPS program: array print example. Available as `for2.s` on examples from class web page and D2L.

```mips
# for ( i = 0; i < numElements; i++ )
#   print elements[i]
addi    $s1, $zero, 0   # i = 0
la      $t0, numElements
lw      $s2, 0($t0)     # $s2 = numElements
la      $t0, elements   # $t0 = address of elements[0]

loopBegin:
  # test if for loop is done
  slt     $t1, $s1, $s2   # $t1 = i < numElements
  beq     $t1, $zero, loopEnd
  # Compute address of elements[i]
  add     $t1, $s1, $s1
  add     $t1, $t1, $t1   # $t1 = 4 * i
  add     $t2, $t0, $t1   # $t2 = address of elements[i]
  lw      $a0, 0($t2)     # $a0 = elements[i]
  addi    $v0, $zero, 1
  syscall
  # print newline
  la      $a0, newline
  addi    $v0, $zero, 4
  syscall
```

Write a pseudo-code version of your algorithm. Then, put it in your code as comments. We will look for these in your programs.
For Loops (continued):

- *for2.s* continued:

```assembly
  for2.s continued:

  addi        $s1, $s1, 1    # i++
  j           loopBegin

  loopEnd:

  done:       # Epilogue for main -- restore stack & frame pointers and return
              lw   $ra, 4($sp)     # get return address from stack
              lw   $fp, 0($sp)     # restore the caller's frame pointer
              addiu $sp, $sp, 24   # restore the caller's stack pointer
              jr    $ra             # return to caller's code
```
**While Loops:**

- Consider the while loop:
  ```plaintext
  while ( save[i] != -2 )  Stop the loop when we find a -2
      if ( save[i] == k )  How many elements in the array save are equal to k?
          count += 1;
          i = i + 1;
  ```

- Here is a complete MIPS program to solve this problem. Available as `while1.s` on the class web page and D2L.

```plaintext
.data
save:    .word   19            # save[0] = 19
        .word   42            # save[1] = 42
        .word   42            # save[2] = 42
        .word   42            # save[3] = 42
        .word   42            # save[4] = 42
        .word   42            # save[5] = 42
        .word   42            # save[6] = 42
        .word   93            # save[7] = 93
        .word   -2            # save[8] = -2
k:       .word   42            # number within save that we are looking for
str:     .asciiz "The final value of count = 
newline: .asciiz "\n"
```
While loops (continued):

.text
main: # Function prologue -- even main has one
    addiu $sp, $sp, -24 # allocate stack space -- default of 24 here
    sw $fp, 0($sp) # save caller's frame pointer
    sw $ra, 4($sp) # save return address
    addiu $fp, $sp, 20 # setup main's frame pointer

    la $s6, save # $s6 = address of save[0], beginning of array

    add $s3, $zero, $zero # initial value of i is 0

    la $t0, k
    lw $s5, 0($t0) # $s5 = value of k

    addi $s1, $zero, -2 # Put ending value of -2 into $s1
    add $s2, $zero, $zero # $s2 = count = 0; start count at zero

LoopBegin:
    # Loop Test, stop loop when we find a -2 in save[i]
    add $t1, $s3, $s3 # quadruple i to get offset for save[i]
    add $t1, $t1, $t1
    add $t1, $t1, $s6 # compute address of save[i]
    lw $t0, 0($t1) # $t0 = value stored at save[i]
    beq $t0, $s1, LoopEnd # end loop if save[i] == -2
While loops (continued):

```mips
# Loop body
# if ( save[i] == k )
#   count++

bne $t0, $s5, afterIncrement  # if ( save[i] != k )
addi $s2, $s2, 1       # count += 1

afterIncrement:
  addi $s3, $s3, 1       # i++

j LoopBegin           # back to start of loop

LoopEnd:
# Print count with a label
la $a0, str            # $a0 = address of start of string
addi $v0, $zero, 4
syscall

add $a0, $s2, $zero    # $a0 = value of count
addi $v0, $zero, 1
syscall

la $a0, newline        # $a0 = address of newline string
addi $v0, $zero, 4
syscall
```

Note: The code for the if statement is entirely contained within the body of the loop. Do **not** jump outside the loop!
While loops (continued): The WRONG way to do an if statement that is inside a loop

```
# Loop body
# if ( save[i] == k )
#    count++
beq $t0, $s5, increment  # if ( save[i] != k )

afterIncrement:
    addi $s3, $s3, 1       # i++
    j LoopBegin           # back to start of loop

LoopEnd:
    # Print count with a label
    la $a0, str            # $a0 = address of start of string
    addi $v0, $zero, 4
    syscall
    add $a0, $s2, $zero    # $a0 = value of count
    addi $v0, $zero, 1
    syscall
    la $a0, newline        # $a0 = address of newline string
    addi $v0, $zero, 4
    syscall
    j done

increment:    addi $s2, $s2, 1       # count += 1
    j afterIncrement
```

Patrick’s FORTRAN story:

```
if ( x .lt. y ) goto 53
```
While loops (continued):

```assembly
done:   # Epilogue for main -- restore stack & frame pointers and return
lw     $ra, 4($sp)        # get return address from stack
lw     $fp, 0($sp)        # restore the caller's frame pointer
addiu  $sp, $sp, 24      # restore the caller's stack pointer
jr      $ra               # return to caller's code
```
**Other conditions for branching:**

- So far, we have `beq` and `bne` for branching, and we have `slt` for comparing (without branching).
- What about other possibilities? Branch on less than, branch on greater than, branch on less than or equal, etc?
- Can have an assembly language that provides all of these options.
  - MIPS does not! (I am not counting pseudo-instructions.)
- Can use `slt` with `bne` or `beq` to build the missing branch instructions.
- Example:
  - Branch if greater than (assume $x$ is in $s1$, and $y$ is in $s2$):
    ```
    if ( x > y ) then
        
        slt  $t0, $s2, $s1  # $t0 = $s2 < $s1? $t0 = y < x?
        beq  $t0, $zero, toElse
        sub  $s4, $s1, $s2  # z = x - y
        j    afterIf

    else
        
        toElse:
        sub  $s4, $s2, $s1  # z = y - x
        afterIf:             
        # continue program here
    ```
  - Note: The order of the comparison was reversed.
    - The question becomes: Is $\textbf{not}(y < x)$? If so, the branch goes to the else clause.
    - Reversing the meaning of the comparison is a common technique!
**Constant or Immediate Operands:**

- Many times, one operand of an arithmetic instruction is a small constant integer.
  - Can occur in 50% or more of the MIPS instructions in some programs!
- Possible solutions:
  - Put typical constants in memory and load them into a register when needed.
    - Can be quite slow to perform the `lw` operation so many times in a program!
  - Hard-wire some of the registers to hold the most commonly used constants.
    - The `$zero` register in MIPS.
    - Difficult in general to decide which constants (other than zero) are most needed.
- MIPS Solution (and something similar is done in many other assembly languages):
  - Add “a few” instructions that allow one operand (of the three) to be stored in the instruction itself.
    - `addi`
    - `slti`
  - Have already been using `addi`.

More Examples:

```
# Put 20 into a register
.data
twenty: .word 20
.text
la $t0, twenty
lw $s2, 0($t0)
slt $t0, $s2, $s3
```

```
addi $s7, $s2, -17  # $s7 = $s2 - 17
```

```
add  $zero, $s7, $s2  # does NOT change $zero
```

```
addi $s7, $s2, -17  # $s7 = $s2 - 17
```

```
slti $t0, $s2, 20   # $t0 = $s2 < 20
```

16-bit immediate value: -32,768 to +32,767
```
slti $t0, 20, $s2    # does NOT work!!
```
Constant or Immediate Operands (continued):

- Examples:
  
  ```assembly
  addi $t1, $s3, 4      # $t1 = $s3 + 4
  addi $t7, $s2, -27    # $t7 = $s2 + (-27)
  slti $t2, $s4, -8     # $t2 = $s4 < -8, $t2 == 0 if false, $t2 == 1 if true
  ```

- The immediate operand in all these cases is a 16-bit, signed, two’s complement integer.
  
  - Thus, the range for the immediate operand is: -32,768 to +32,767.

- Note: there is no `subi` instruction. Why?

- `la` is a *pseudo-instruction*. It will accept an argument as large as a 32-bit, signed, two’s complement integer.
  
  - The argument is assumed to be a label.
  
  - The address associated with the label is put in the specified register.
  
  - The assembler (spim in our case) will substitute one or two instructions for the `la`.
  
  - How many instructions are substituted depends on the address of the label.

  ```assembly
  la $t3, numbers
  ```
MIPS Instruction summary to date:

<table>
<thead>
<tr>
<th>Category</th>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>add</td>
<td>add $s1, $s2, $s3</td>
<td>$s1 = $s2 + $s3</td>
<td>3 operands; data in reg</td>
</tr>
<tr>
<td></td>
<td>add immediate</td>
<td>addi $s1, $s2, 17</td>
<td>$s1 = $s2 + 17</td>
<td>3 operands; data in reg</td>
</tr>
<tr>
<td></td>
<td>subtract</td>
<td>sub $s1, $s2, $s3</td>
<td>$s1 = $s2 - $s3</td>
<td>3 operands; data in reg</td>
</tr>
<tr>
<td>Data transfer</td>
<td>load word</td>
<td>lw $s1, 0($s2)</td>
<td>$s1 = Memory[$s2]</td>
<td>Data from memory to reg</td>
</tr>
<tr>
<td></td>
<td>store word</td>
<td>sw $s3, 0($t5)</td>
<td>Memory[$t5] = $s3</td>
<td>Data from reg to memory</td>
</tr>
<tr>
<td>Conditional branch</td>
<td>branch on equal</td>
<td>beq $s1,$s2,Label</td>
<td>if ($s1 == $s2) \goto Label</td>
<td>Equal test and branch</td>
</tr>
<tr>
<td></td>
<td>branch on not equal</td>
<td>bne $s1,$s2,Label</td>
<td>if ($s1 \neq $s2) \goto Label</td>
<td>Not equal test &amp; branch</td>
</tr>
<tr>
<td></td>
<td>set on less than</td>
<td>slt $s1, $s2, $s3</td>
<td>if ($s2 &lt; $s3) $s1 = 1; else $s1 = 0</td>
<td>Compare less than</td>
</tr>
<tr>
<td></td>
<td>set on less than imm</td>
<td>slti $s1, $s2, num</td>
<td>if ($s2 &lt; num) $s1 = 1; else $s1 = 0</td>
<td>Compare less than w/ immediate operand</td>
</tr>
<tr>
<td>Unconditional branch</td>
<td>jump</td>
<td>j Label</td>
<td>\goto Label</td>
<td>Branch to target address</td>
</tr>
</tbody>
</table>
Machine Language

Reading: Section 2.5 (4th edition).

- Instructions have to be stored in memory. It is a design choice as to how to represent the instructions.
- MIPS makes the choice to use a 32-bit value to represent instructions.
  - This choice could be different.
  - Some instruction sets use different sizes for different instructions.
    - Can result in multiple reads from memory to get one instruction into the CPU.
    - Example: The Motorola 68000 chip uses instruction sizes that vary in length from 16 to 80 bits.
Machine Language (continued):

- Repeating Slide #23:

- Text Segment:
  - Displays the program code.

---

Address of instruction hexadecimal

MIPS machine language 32-bit, hexadecimal

Line number from your .s file

---

<table>
<thead>
<tr>
<th>Address of instruction hexadecimal</th>
<th>MIPS machine language 32-bit, hexadecimal</th>
<th>Line number from your .s file</th>
</tr>
</thead>
<tbody>
<tr>
<td>00400024</td>
<td>addiu $29, $29, -24</td>
<td>13: addiu $sp, $sp, -24 # allocate stack space -- default of 32 bytes</td>
</tr>
<tr>
<td>00400028</td>
<td>sw $30, 0($29)</td>
<td>14: sw $fp, 0($sp) # save caller's frame pointer</td>
</tr>
<tr>
<td>0040002c</td>
<td>sw $31, 4($29)</td>
<td>15: sw $ra, 4($sp) # save return address</td>
</tr>
<tr>
<td>00400030</td>
<td>addiu $30, $29, 20</td>
<td>16: addiu $fp, $sp, 20 # setup main's frame pointer</td>
</tr>
<tr>
<td>00400034</td>
<td>lui $8, 4097 [bats]</td>
<td>18: la $t0, bats # $t0 has address of the bats memory location</td>
</tr>
<tr>
<td>00400038</td>
<td>lw $16, 0($8)</td>
<td>19: lw $s0, 0($t0) # $s0 now holds the number of bats</td>
</tr>
<tr>
<td>0040003c</td>
<td>lui $1, 4097 [balls]</td>
<td>20: la $t0, balls # $t0 has address of the balls memory location</td>
</tr>
<tr>
<td>00400040</td>
<td>ori $8, $1, 4 [balls]</td>
<td>21: lw $s1, 0($t0) # $s1 now holds the number of balls</td>
</tr>
<tr>
<td>00400044</td>
<td>lw $17, 0($8)</td>
<td>22: la $t0, gloves # $t0 has address of the gloves memory location</td>
</tr>
<tr>
<td>00400048</td>
<td>lui $1, 4097 [gloves]</td>
<td>23: lw $s2, 0($t0) # $s2 now holds the number of gloves</td>
</tr>
<tr>
<td>0040004c</td>
<td>ori $8, $1, 8 [gloves]</td>
<td>24: add $s3, $s0, $s1 # $s3 now holds the number of bats+balls</td>
</tr>
<tr>
<td>00400050</td>
<td>lw $18, 0($8)</td>
<td>25: add $s3, $s3, $s2 # $s3 now holds the number of bats+balls+bats</td>
</tr>
<tr>
<td>00400054</td>
<td>add $19, $16, $17</td>
<td>27: la $t0, sum # $t0 has address of the sum memory location</td>
</tr>
<tr>
<td>00400058</td>
<td>add $19, $19, $18</td>
<td>28: sw $s3, 0($t0) # sum now holds the number of bats+balls+bats</td>
</tr>
<tr>
<td>0040005c</td>
<td>lui $1, 4097 [sum]</td>
<td>31: lw $ra, 4($sp) # get return address from stack</td>
</tr>
<tr>
<td>00400060</td>
<td>ori $8, $1, 16 [sum]</td>
<td>32: lw $fp, 0($sp) # restore the caller's frame pointer</td>
</tr>
<tr>
<td>00400064</td>
<td>sw $19, 0($8)</td>
<td>33: addiu $sp, $sp, 24 # restore the caller's stack pointer</td>
</tr>
<tr>
<td>00400068</td>
<td>lw $31, 4($29)</td>
<td>34: jr $ra # return to caller's code</td>
</tr>
<tr>
<td>0040006c</td>
<td>lw $30, 0($29)</td>
<td></td>
</tr>
<tr>
<td>00400070</td>
<td>addiu $29, $29, 24</td>
<td></td>
</tr>
<tr>
<td>00400074</td>
<td>jr $31</td>
<td></td>
</tr>
</tbody>
</table>

Assembly language instruction, from your .s file, before assembly

Assembly language instruction, after assembly
Machine Language (continued):

- For example:

  \( \text{add } \$t1, \$s1, \$s2 \)

- Registers are represented in the machine instruction as numbers in the range 0..31.
  - \( \$t1 \) is register 9; \( \$s1 \) is register 17; \( \$s2 \) is register 18

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>10001</td>
<td>10010</td>
<td>01001</td>
<td>00000</td>
<td>100000</td>
</tr>
</tbody>
</table>

| \( 0_{\text{ten}} \) | \( 17_{\text{ten}} \) | \( 18_{\text{ten}} \) | \( 9_{\text{ten}} \) | \( 0_{\text{ten}} \) | \( 32_{\text{ten}} \) |

- Field abbreviations: SPIM Appendix section 10.2, page 50.
  - op — basic operation of the instruction, the operation code or opcode.
  - rs — first register source operand.
  - rt — second register source operand.
  - rd — register destination operand; the result of the operation goes into this register.
  - shamt — shift amount (we will see the use of this field later, for now it will always be zero).
  - funct — function. Selects the specific variant of the opcode. (Figure 10.2, page 50 in SPIM Appendix).
Machine Language (continued):

- Other points to notice about the layout:

```
<table>
<thead>
<tr>
<th></th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 bits</td>
<td>31</td>
<td>26</td>
<td>25</td>
<td>21</td>
<td>20</td>
<td>16</td>
</tr>
</tbody>
</table>
```

- It is no accident that there are 16 bits with three fields in each half of the format.
- The Most Significant Bit (MSB) is on the opcode end of the instruction.
- The Least Significant Bit (LSB) is on the funct end of the instruction.
- This instruction format is called R-type.
- This format is used for most of the arithmetic instructions.
  - The main exception are the arithmetic instructions that take immediate operands.
Machine Language (continued):

- More R-format instruction examples:

- Set-Less-Than:

  \[
  \text{\textbf{slt} \hspace{5pt} $t0$, $t1$, $t2$}
  \]

  - Register \texttt{$t0$} is \texttt{8}; Register \texttt{$t1$} is \texttt{9}; Register \texttt{$t2$} is \texttt{10}.
  - The opcode is 0 (again); the \texttt{funct} field is different from the \texttt{add}.
    - This is the only way to tell it is an \texttt{slt} rather than an \texttt{add}.

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>01001</td>
<td>01010</td>
<td>01000</td>
<td>00000</td>
<td>101010</td>
</tr>
<tr>
<td>\texttt{0}_{\text{ten}}</td>
<td>\texttt{9}_{\text{ten}}</td>
<td>\texttt{10}_{\text{ten}}</td>
<td>\texttt{8}_{\text{ten}}</td>
<td>\texttt{0}_{\text{ten}}</td>
<td>\texttt{42}_{\text{ten}}</td>
</tr>
</tbody>
</table>
Machine Language (continued):

- Subtract:
  - There are two versions of subtract:
    - `sub` assumes the values in the registers are **signed** numbers.
    - `subu` assumes the values in the registers are **unsigned** numbers.

```plaintext
sub  $s1, $t3, $t4            # $s1 is $17; $t3 is $11; $t4 is $12

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>01011</td>
<td>01100</td>
<td>10001</td>
<td>00000</td>
<td>100010</td>
</tr>
<tr>
<td>0_{ten}</td>
<td>11_{ten}</td>
<td>12_{ten}</td>
<td>17_{ten}</td>
<td>0_{ten}</td>
<td>34_{ten}</td>
</tr>
</tbody>
</table>

subu $s1, $t3, $t4           # $s1 is $17; $t3 is $11; $t4 is $12

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>01011</td>
<td>01100</td>
<td>10001</td>
<td>00000</td>
<td>100011</td>
</tr>
<tr>
<td>0_{ten}</td>
<td>11_{ten}</td>
<td>12_{ten}</td>
<td>17_{ten}</td>
<td>0_{ten}</td>
<td>35_{ten}</td>
</tr>
</tbody>
</table>
```
Machine Language (continued):

Arithmetic instructions with Immediate operands:

- Major difference: Each instruction has two registers and an immediate value (signed, 16-bit, 2’s complement integer).
- Need 16 bits (out of a total of 32 bits) for the immediate operand:
  - We can make use of the shamt field: 5 bits.
  - We only need two registers, not three. This gains: 5 bits.
  - Total so far: 10 bits. But we need 16…
- Solution: Use a different opcode for each immediate instruction.
  - Gains the funct field: 6 bits.
  - Total is now: 16 bits.
- This is the I-format instruction.
- Puts the immediate operand in the second half (right half) of the instruction:

```
addi $s7, $t0, 321    # $s7 is $23; $t0 is $8
```

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>16 bit number</th>
</tr>
</thead>
<tbody>
<tr>
<td>001000</td>
<td>0100</td>
<td>10111</td>
<td>0000 0001 0100 0001</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>8_{ten}</th>
<th>8_{ten}</th>
<th>23_{ten}</th>
<th>321_{ten}</th>
</tr>
</thead>
</table>
Machine Language (continued):

More I-format instructions:

- Conditional branch statements need to include an address.
- Addresses are 32-bit values in MIPS.
- Dilemma: How to fit the opcode, two registers, and an address into a 32-bit MIPS instruction?
  - Use PC-relative addressing.
    - Address specified in the conditional branch is relative to the current value of the Program Counter.
    - Use the specified address as a 16-bit, two’s complement value. Add this to the current value of the program counter.
    - Allows forward or backward branching.
    - Can not branch to locations that are “too far away.”
- Optimization:
  - All MIPS instructions are 32-bit. They are all word-aligned (start on an address divisible by 4).
  - Let the offset in the conditional branch instruction be an “instruction” offset, rather than a byte offset. That is, the offset is multiplied by 4 before being added to the Program Counter.
Machine Language (continued):

More I-format instructions:

- Example of conditional branch in I-format:

```
beq $t0, $s1, toThere    # $t0 is $8; $s1 is $17
add $t0, $s7, $s7        # $t0 = 2 * $s7
j    below

toThere: sub $t0, $s7, $s5  # $t0 = $s7 - $s5
below:  add $t1, $t1, $t0  # $t1 = $t1 + $t0
```

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>16 bit number</th>
</tr>
</thead>
<tbody>
<tr>
<td>000100</td>
<td>01000</td>
<td>10001</td>
<td>0000 0000 0000 0010</td>
</tr>
</tbody>
</table>

- The Program Counter contains the address of the next instruction.
- Thus, to branch to the label `toThere` we add two instructions: PC + 2 * (size of a MIPS instruction).
Machine Language (continued):

More I-format instructions — \texttt{lw} and \texttt{sw}:

- The \texttt{lw} and \texttt{sw} instructions use only two registers (similar to \texttt{bne} and \texttt{beq} in this respect).
- They also use a 16-bit, two's complement number; the offset.
- Unlike \texttt{beq}, the offset for \texttt{lw} and \texttt{sw} is a count of bytes, not words.

\begin{verbatim}
lw   $s7, 0($s2)  # $s7 is $23; $s2 is $18
sw   $s3, 168($s1)  # $s3 is $19; $s1 is $17
\end{verbatim}

\begin{center}
\begin{tabular}{|c|c|c|c|}
\hline
op & rs & rt & 16 bit number \\
\hline
100011 & 10010 & 10111 & 0000 0000 0000 0000 \\
\hline
35_{ten} & 18_{ten} & 23_{ten} & 0_{ten} \\
\hline
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{|c|c|c|c|}
\hline
op & rs & rt & 16 bit number \\
\hline
101011 & 10001 & 10011 & 0000 0000 1010 1000 \\
\hline
43_{ten} & 17_{ten} & 19_{ten} & 168_{ten} \\
\hline
\end{tabular}
\end{center}
Machine Language (continued):

Unconditional branch and the J-Format:

• The jump instruction takes a single argument, which is the address of an instruction.

• The instruction has to fit within 32-bits (as do all MIPS instructions).
  • Need an opcode: 6 bits.
  • Need an address: 32 bits.
  • That’s 38 bits(!!)
  • Can not use a 32-bit address.

• The address of an instruction is always a multiple of 4.
  • Addresses that are multiples of 4 always have two binary zeroes on the right.
  • Don’t store the two zeroes.

• Now:
  • Need an opcode: 6 bits.
  • Need an address: 30 bits.
  • That’s 36 bits; closer, but not quite there yet.
Machine Language (continued):

Unconditional branch and the J-Format (continued):

- So far:

\[
\text{j } \quad 0x0040 \ 0240 \\
0x0040 \ 0240 = 0000 \ 0000 \ 0100 \ 0000 \ 0000 \ 0010 \ 0100 \ 0000_{\text{two}}
\]

- Take the left-most 4 bits from the PC:

\[
\text{j } \quad 0x0040 \ 0240 \\
0x0040 \ 0240 = 0000 \ 0000 \ 0100 \ 0000 \ 0000 \ 0010 \ 0100 \ 0000_{\text{two}}
\]

- Allows a \text{j} instruction to go to any instruction that has the same high-order 4 bits as the PC.

- Divides the 4GB address space into 16 segments, each containing (how many?) bytes.
Logical Operations

Reading: Section 2.6 (4th edition), 2.5 (3rd edition).

- **AND:**
  
  \[ \text{and} \quad \$t0, \quad \$s1, \quad \$s2 \]

  Performs a *bit-wise* AND operation:

  \[
  \begin{align*}
  \$s1 &= 0010\ 0110\ 0110\ 0110\ 1111\ 1111\ 0000\ 1010 \\
  \$s2 &= 1111\ 0000\ 0000\ 0000\ 1111\ 0000\ 1111\ 1111 \\
  \$s1 \text{ AND} \ $s2 &= 0010\ 0000\ 0000\ 0000\ 1111\ 0000\ 0000\ 1010
  \end{align*}
  \]

- For Java and C, this operation is done with `&`

  - **Example:**
    
    \[
    x = y \ & \ z;
    \]

  \[
  \begin{array}{|c|c|c|c|c|c|}
  \hline
  \text{op} & \text{rs} & \text{rt} & \text{rd} & \text{shamt} & \text{funct} \\
  \hline
  000000 & 10001 & 10010 & 01000 & 00000 & 100100 \\
  \hline
  0ten & 17ten & 18ten & 8ten & 0ten & 36ten \\
  \hline
  \end{array}
  \]

  \[
  \begin{array}{|c|c|c|}
  \hline
  a & b & c = a \text{ AND} \ b \\
  \hline
  0 & 0 & 0 \\
  \hline
  0 & 1 & 0 \\
  \hline
  1 & 0 & 0 \\
  \hline
  1 & 1 & 1 \\
  \hline
  \end{array}
  \]

  \[
  \text{In Java:}
  \begin{align*}
  \text{short } x, y, z; \\
  x = 19; \quad // \ 0000\ 0000\ 0000\ 0011 \\
  y = 21; \quad // \ 0000\ 0000\ 0001\ 0101 \\
  z = x \ & \ y; \quad // \ 0000\ 0000\ 0001\ 0001 \\
  \text{System.out.println}(z); \quad // \ 17
  \end{align*}
  \]
Logical Operations (continued):

- **OR:**
  
  ```
  or    $t0, $s1, $s2
  ```

- Performs a *bit-wise* OR operation:

  ```
  $s1 = 0010 0110 0110 0110 1111 1111 0000 1010
  $s2 = 1111 0000 0000 0000 1111 0000 1111 1111
  $s1 OR $s2 = 1111 0110 0110 0110 1111 1111 1111 1111
  ```

- For Java and C, this operation is done with |

- **Example:**

  ```
  x = y | z;
  ```

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>10001</td>
<td>10010</td>
<td>01000</td>
<td>00000</td>
<td>100101</td>
</tr>
<tr>
<td>0ten</td>
<td>17ten</td>
<td>18ten</td>
<td>8ten</td>
<td>0ten</td>
<td>37ten</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c = a OR b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Logical Operations (continued):

- Masking using `and`

  ```
  .data
  mask: .word 0x00FF00FF # $s2 = 0000 0000 1111 1111 0000 0000 1111 1111
  .text
  la $s2, mask
  lw $s2, 0($s2)
  ... Some value gets put into $s1 ...
  and $s3, $s1, $s2
  ```

- What ends up in $s3?

  ```
  $s1 = ???? ???? ???? ???? ???? ???? ???? ???? ????
  $s2 = 0000 0000 1111 1111 0000 0000 1111 1111
  $s3 =
  ```
Logical Operations (continued):

- Masking using `or`

```assembly
.data
mask: .word 0x00FF00FF  # $s2 = 0000 0000 1111 1111 0000 0000 1111 1111
.text
la $s2, mask
lw $s2, 0($s2)
... Some value gets put into $s1 ...
or $s3, $s1, $s2
```

- What ends up in $s3?

```plaintext
$s1 = ???? ???? ???? ???? ???? ???? ???? ???? ???? ???? ???? ???? ???? ???? ???? ???? ???? ???? ???? ???? ???? $s2 = 0000 0000 1111 1111 0000 0000 1111 1111
$s3 =
```
Logical Operations (continued):

- NOR, Not OR:

  \[
  \text{nor} \quad t0, s1, s2
  \]

  - Performs a *bit-wise* NOT OR operation.
  
  - The NOT OR operation takes the OR of the two operands, then reverses (NOT’s) the result:

    \[
    \begin{align*}
    s1 &= 0010 \ 0110 \ 0110 \ 0110 \ 1111 \ 1111 \ 0000 \ 1010 \\
    s2 &= 1111 \ 0000 \ 0000 \ 0000 \ 1111 \ 0000 \ 1111 \ 1111 \\
    s1 \ OR \ s2 &= 1111 \ 0110 \ 0110 \ 0110 \ 1111 \ 1111 \ 1111 \ 1111 \\
    s1 \ NOR \ s2 &= 0000 \ 1001 \ 1001 \ 1001 \ 0000 \ 0000 \ 0000 \ 0000
    \end{align*}
    \]

  - There is **not** an operator for this operation in Java or C

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>10001</td>
<td>10010</td>
<td>01000</td>
<td>00000</td>
<td>100111</td>
</tr>
<tr>
<td>0ten</td>
<td>17ten</td>
<td>18ten</td>
<td>8ten</td>
<td>0ten</td>
<td>39ten</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c = a NOR b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Logical Operations (continued):

• NOT:
  - The bit-wise NOT operation simply reverses each bit: 0 becomes 1; 1 becomes 0.
    
    $s1 = 0010 \ 0110 \ 0110 \ 0110 \ 1111 \ 1111 \ 0000 \ 1010$
    
    NOT $s1 = 1101 \ 1001 \ 1001 \ 1001 \ 0000 \ 0000 \ 1111 \ 0101$
  
  • MIPS does not have a NOT operation.
  
  • Instead, we can use NOR with $zero$ as one of the registers. For example, to apply the NOT operation to $s1$, do the following:
    
    NOR $t0$, $s1$, $zero$
  
  • There is a NOT operator in C and Java for doing a bit-wise NOT operation:
    
    • The tilde operator:
      
      $x = \sim y$;

<table>
<thead>
<tr>
<th>a</th>
<th>NOT a</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Logical Operations (continued):

- XOR, Exclusive-OR:
  
  ```
  xor    $t0, $s1, $s2
  ```

  Performs a *bit-wise* XOR operation. Exclusive-OR has the answer 0 when both operands are 1.

- XOR

  ```
  \[
  \begin{align*}
  \text{$s1 = 0010 \ 0110 \ 0110 \ 0110 \ 1111 \ 1111 \ 0000 \ 1010$} \\
  \text{$s2 = 1111 \ 0000 \ 0000 \ 0000 \ 1111 \ 0000 \ 1111 \ 1111$} \\
  \text{$s1 \ \text{XOR} \ \text{s2} = 1101 \ 0110 \ 0110 \ 0110 \ 0000 \ 1111 \ 1111 \ 0101$}
  \end{align*}
  \]

- For Java and C, this operation is done with `^`

- Example:

  ```
  x = y ^ z;
  ```

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>10001</td>
<td>10010</td>
<td>01000</td>
<td>00000</td>
<td>100110</td>
</tr>
<tr>
<td>0_{ten}</td>
<td>17_{ten}</td>
<td>18_{ten}</td>
<td>8_{ten}</td>
<td>0_{ten}</td>
<td>38_{ten}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c = a XOR b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Logical Operations (continued):

- **XOR, Exclusive-OR:**
  
  ```
xor $t0, $s1, $s2
  ```

  Performs a *bit-wise* XOR operation. Exclusive-OR has the answer 0 when both operands are 1.

- **XOR**
  
  ```
  $s1 = 0010 0110 0110 1111 1111 0000 1010    the key
  $s2 = 1111 0000 0000 0000 1111 0000 1111 1111    the message
  
  $s1 XOR $s2 = 1101 0110 0110 0110 0000 1111 1111 0101    the encrypted message
  ```

  ```
  $s1 XOR $s2 = 1101 0110 0110 0110 0000 1111 1111 0101    the encrypted message
  
  $s1 = 0010 0110 0110 1111 1111 0000 1010    the key
  
  1111 0000 0000 0000 1111 0000 1111 1111    the message
Logical Operations (continued):

• Immediate versions of AND, OR, XOR. These are named: **andi**, **ori**, **xori**.

• They each have a 16-bit, signed, two’s complement number for the immediate operand.

• The immediate value is **zero-extended** to create a 32-bit number. Then the operation is applied.

  \[
  \begin{align*}
  \text{andi } & \quad t0, s1, 0x74A2 \quad \text{andi } \quad t0, s1, 0x74A2 \quad \text{andi } \quad t3, s4, 17 \\
  \text{\quad } s1 &= 0010 \ 0110 \ 0110 \ 0110 \ 1111 \ 1111 \ 0000 \ 1010 \\
  \text{\quad } \text{0x74A2} &= 0000 \ 0000 \ 0000 \ 0000 \ 0111 \ 0100 \ 1010 \ 0010 \\
  \text{\quad } 0000 \ 0000 \ 0000 \ 0000 \ 0111 \ 0100 \ 0000 \ 0010 \\
  \end{align*}
  \]

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>16 bit number</th>
</tr>
</thead>
<tbody>
<tr>
<td>001100</td>
<td>10001</td>
<td>10010</td>
<td>0111 0100 1010 0010</td>
</tr>
<tr>
<td>12_{ten}</td>
<td>17_{ten}</td>
<td>18_{ten}</td>
<td>29858_{ten}</td>
</tr>
</tbody>
</table>

  \[
  \begin{align*}
  \text{ori } & \quad t0, s1, 0x94A2 \\
  \text{\quad } s1 &= 0010 \ 0110 \ 0110 \ 0110 \ 1111 \ 1111 \ 0000 \ 1010 \\
  \text{\quad } \text{0x94A2} &= 0000 \ 0000 \ 0000 \ 0000 \ 1001 \ 0100 \ 1010 \ 0010 \\
  \text{\quad } 0010 \ 0110 \ 0110 \ 0110 \ 1111 \ 1111 \ 1010 \ 1010 \\
  \end{align*}
  \]

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>16 bit number</th>
</tr>
</thead>
<tbody>
<tr>
<td>001101</td>
<td>10001</td>
<td>10010</td>
<td>1001 0100 1010 0010</td>
</tr>
<tr>
<td>13_{ten}</td>
<td>17_{ten}</td>
<td>18_{ten}</td>
<td>-27486_{ten}</td>
</tr>
</tbody>
</table>
Logical Operations (continued):

Shifting:

- Logical shifts:
  - Move the bits in the register.
  - Bits “fall off” the end of the register.
- Shift Left Logical: `sll`


```
sll $t0, $s1, 4       # $t0 = $s0 shifted left 4 bits
$s1 = 1001 1100 0100 0000 1111 0101 1100 0011
$t0 = 1100 0100 0000 1111 0101 1100 0011 0000
```

- For Java and C, this operation is done with `<<`
  - Example:
    ```
x = y << 4;     x <<= 5;
```

- `rs` field is ignored (not used) since we only have two registers in the instruction.
- `shamt` field is now being used.
Logical Operations (continued):

- **Shift Right Logical:** \texttt{srl}

  
  \begin{verbatim}
  srl $t0, $s1, 4       # $t0 = $s0 shifted right 4 bits
  $s1 = 1001 1100 0100 0000 1111 0101 1100 0011
  $t0 = 0000 1001 1100 0100 0000 1111 0101 1100
  \end{verbatim}

- For C, this operation is done with `>>`
  - Example: `x = y >> 4;`

- For Java, this operation is done with `>>>`
  - Example: `x = y >>> 4;`

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>0000</td>
<td>10001</td>
<td>01000</td>
<td>00100</td>
<td>000010</td>
</tr>
<tr>
<td>\text{0_{ten}}</td>
<td>\text{0_{ten}}</td>
<td>\text{17_{ten}}</td>
<td>\text{8_{ten}}</td>
<td>\text{4_{ten}}</td>
<td>\text{2_{ten}}</td>
</tr>
</tbody>
</table>

- Notes on using Logical shifts:
  - The \texttt{shamt} field has 5 bits; enough to shift from 0 to 31 positions.
  - We can use a logical shift left by two places to multiply by 4:
    - Useful in computing array offsets. Use one \texttt{sll} instead of using two \texttt{add} instructions.
  - Can think of shift left logical as a multiply by a power of two.
Logical Operations (continued):

- Shift Right Arithmetic

  \[
  \text{sr} \ a \ \$t0, \ \$s1, \ 6 \quad \# \ \$t0 = \$s0 \text{ shifted right 6 bits}
  \]

  \[
  \begin{align*}
  \$s1 & = 1001 \ 1100 \ 0100 \ 0000 \ 1111 \ 0101 \ 1100 \ 0011 \\
  \$t0 & = 1111 \ 1110 \ 0111 \ 0001 \ 0000 \ 0011 \ 1101 \ 0111
  \end{align*}
  \]

- The arithmetic shift will copy the sign bit on the left.
  - If the number is positive, 0’s are shifted into the left side.
  - If the number is negative, 1’s are shifted into the left side (as in the example above).

- For Java, this operation is done with `>>`

- Example:
  \[
  x = y >> 6;
  \]

- For C, the definition of `>>` is an implementation-dependent feature. That is, a compiler may choose to implement `>>` as a logical shift, or as an arithmetic shift.

- Example:
  \[
  x = y >> 6;
  \]

  - For an unsigned int, this gives shift right logical.
  - For a signed int, the result is implementation-dependent!
<table>
<thead>
<tr>
<th>Category</th>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>add</td>
<td>add $s1, $s2, $s3</td>
<td>$s1 = $s2 + $s3</td>
<td>3 operands; data in reg</td>
</tr>
<tr>
<td></td>
<td>add immediate</td>
<td>addi $s1, $s2, 17</td>
<td>$s1 = $s2 + 17</td>
<td>3 operands; data in reg</td>
</tr>
<tr>
<td></td>
<td>subtract</td>
<td>sub $s1, $s2, $s3</td>
<td>$s1 = $s2 - $s3</td>
<td>3 operands; data in reg</td>
</tr>
<tr>
<td>Data transfer</td>
<td>load word</td>
<td>lw $s1, 0($s2)</td>
<td>$s1 = Memory[$s2]</td>
<td>Data from memory to reg</td>
</tr>
<tr>
<td></td>
<td>store word</td>
<td>sw $s3, 0($t5)</td>
<td>Memory[$t5] = $s3</td>
<td>Data from reg to memory</td>
</tr>
<tr>
<td>Logical</td>
<td>and</td>
<td>and $t0, $s1, $s2</td>
<td>$t0 = $s1 &amp; $s2</td>
<td>bit-wise AND</td>
</tr>
<tr>
<td></td>
<td>and immediate</td>
<td>andi $t0,$s1,0x00A3</td>
<td>$t0 = $s1 &amp; 0x00A3</td>
<td>bit-wise AND immediate</td>
</tr>
<tr>
<td></td>
<td>or</td>
<td>or $t0, $s1, $s2</td>
<td>$t0 = $s1</td>
<td>$s2</td>
</tr>
<tr>
<td></td>
<td>or immediate</td>
<td>or $t0,$s1,0x00A3</td>
<td>$t0 = $s1</td>
<td>0x00A3</td>
</tr>
<tr>
<td></td>
<td>nor</td>
<td>nor $t0, $s1, $s2</td>
<td>$t0 = $s1 NOR $s2</td>
<td>bit-wise NOR</td>
</tr>
<tr>
<td></td>
<td>xor</td>
<td>xor $t0, $s1, $s2</td>
<td>$t0 = $s1 ^ $s2</td>
<td>bit-wise XOR</td>
</tr>
<tr>
<td></td>
<td>xor immediate</td>
<td>xor $t0,$s1,0x00A3</td>
<td>$t0 = $s1 ^ 0x00A3</td>
<td>bit-wise XOR immediate</td>
</tr>
<tr>
<td></td>
<td>shift left logical</td>
<td>sll $t0, $s1, 3</td>
<td>$t0 = $s1 &lt;&lt; 3</td>
<td>shift $s1 left 3 bits</td>
</tr>
<tr>
<td></td>
<td>shift right logical</td>
<td>srl $t0, $s1, 3</td>
<td>$t0 = $s1 &gt;&gt;&gt; 3</td>
<td>shift $s1 right 3 bits, fill on left with 0’s</td>
</tr>
<tr>
<td></td>
<td>shift right arithmetic</td>
<td>sra $t0, $s1, 3</td>
<td>$t0 = $s1 &gt;&gt; 3</td>
<td>shift $s1 right 3 bits, fill on left with sign bit</td>
</tr>
<tr>
<td>Category</td>
<td>Instruction</td>
<td>Example</td>
<td>Meaning</td>
<td>Comments</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------------</td>
<td>-------------------</td>
<td>----------------------------------------------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>Conditional branch</td>
<td>branch on equal</td>
<td><code>beq $s1,$s2,Label</code></td>
<td>if ($s1 == $s2) goto Label</td>
<td>Equal test and branch</td>
</tr>
<tr>
<td></td>
<td>branch on not equal</td>
<td><code>bne $s1,$s2,Label</code></td>
<td>if ($s1 ≠ $s2) goto Label</td>
<td>Not equal test &amp; branch</td>
</tr>
<tr>
<td></td>
<td>set on less than</td>
<td><code>slt $s1, $s2, $s3</code></td>
<td>if ($s2 &lt; $s3) $s1 = 1; else $s1 = 0</td>
<td>Compare less than</td>
</tr>
<tr>
<td>Unconditional branch</td>
<td>jump</td>
<td><code>j Label</code></td>
<td>goto Label</td>
<td>Branch to target address</td>
</tr>
</tbody>
</table>