Syscall 5

- **System call 5 allows input of numerical data from the keyboard while a program is running.**
- **Syscall 5 is a little unusual. It requires the use of register $v0 twice. In syscall 5 (as for all system calls), the number 5 is loaded into $v0 before executing the "syscall" instruction.**
- **However, after a number is input from the keyboard (input completion is signified by pressing the "Enter" key), this number is stored in $v0.**
- **When inputting a number during syscall 5, the user must hit Enter to complete the syscall.**

Syscall 5 (2)

- **Since $v0 is used for all system calls, the programmer needs to immediately provide for the input data stored in $v0 to be moved elsewhere (memory or register).**
- **Another characteristic of syscall 5: it will only input enough information to fill $v0 (i.e., 0x ffff ffff). The rest is lost (most significant digit first – and MSB is still the sign bit!).**
- **Thus the biggest positive number that can be input on syscall 5 is ~2,145,000,000. Keep that in mind when programming numerical inputs from the keyboard.**
- **The syscall 5 form would be:**
  - **li $v0,5**
  - **syscall**

At the end of this system call, the number input from the keyboard is in $v0.

Syscall 8

- Syscall 8 allows ASCII character input from the keyboard.
- Syscall 8 uses three registers:
  - $a0 – starting address in memory to store string
  - $a1 – character length (max. number of characters to be input)
  - $v0 – (as usual) holds the system call number (8)
- To load $a0:
  - Load $a0 with the address of first byte of memory storage area.
  - Example: "la $a0, buffer," ("buffer" = address label of the first byte of storage space)
  - Reserve the storage area with ".space" directive:
  - Example: buffer: .space,36 (reserves and labels 36 bytes in memory and clears to 0)

Syscall 8 (2)

- **To load $a1:**
  - **li $a1,number – Desired character length → $a1 (e.g., li $a1,36)**
  - **Warning: If [$a1] > reserved memory area, the program can overwrite other data declared in the program!**
  - **If less than that number in $a1 is input, the "Enter" key must be pressed. If input number = [$a1], input automatically ends.**
- **Note: To assure a null-terminated character string, declare a larger reserved area than is actually needed. That is:**
  - **buffer: .space 37 (if max. character number = 36)**
- **A typical complete syscall 8 would look like this:**
  - **la $a0, buffer** ← **"Buffer" is the label of the reserved space in memory.**
  - **li $a1, 36** ← **36 = max. character number (memory space in buffer).**
  - **li $v0, 8**
  - **syscall**

  } **Normal form of a syscall.**

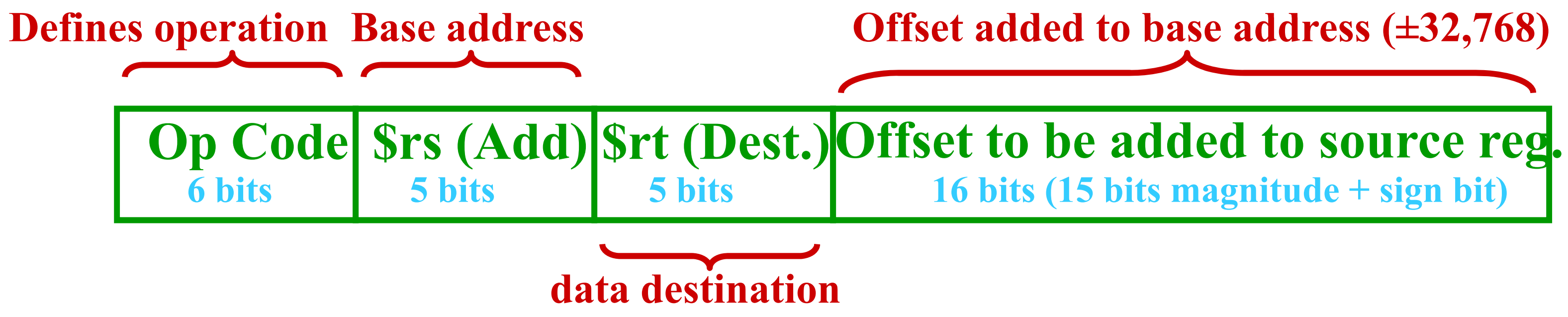
Note on Other Development Environments

- **MIPSter and MARS are two other SPIM programming environments.**
- **MIPSter url is: http://www.downcastsystems.com/mipster/ (it is a free download).**
- **http://courses.missouristate.edu/KenVollmar/MARS/ is the MARS url.**
- **MIPSter has nice features, but its diagnostics are not as good as SPIM, in my opinion. I know very little about MARS, except that it appears to work properly.**
- **I prefer the flexibility of Dr. Larus' PCSPIM (or the newer version, QtSPIM).**
- **You can explore MIPSter or MARS on your own time if you wish.**
- **However, in class and for all homework assignments, we will use ONLY PCSPIM, or the newer version QSPIM, and write all our programs in NotePad.**

MIPS Data Memory Instructions

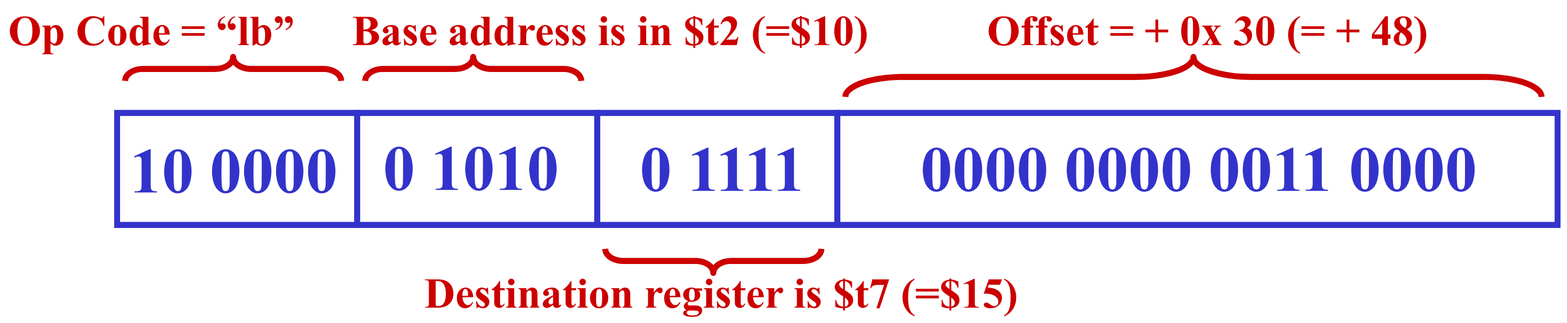
- The MIPS load-store architecture limits memory access.
  - Data transfers occur only via load/store operations.
    - Register indirect addressing – register contains memory address of operand; used in load/store instructions ("i-format" instructions).
    - Displacement-based or indexed addressing – register plus numerical offset in instruction are added for memory address; used in load/store instructions (this is a variation of register indirect addressing, also "i-format").
  - Instruction memory (via the program counter) is accessed only by jump or branch instructions.
    - Direct addressing – instruction contains an absolute memory address; used in jump instructions ("j-format" instructions).
    - Register indirect addressing – register contains memory address of operand; used in jr instructions ("i-format" instruction).
    - Relative addressing – numerical offset in instruction added algebraically to program counter to obtain operand address; used in branch instructions (discussed in a later lecture – also an "i-format" instruction).

Load Instruction Format


Defines operation
Base address
Offset added to base address (±32,768)
Op Code
6 bits
$rs (Add)
5 bits
$rt (Dest.)
5 bits
Offset to be added to source reg.
16 bits (15 bits magnitude + sign bit)
data destination

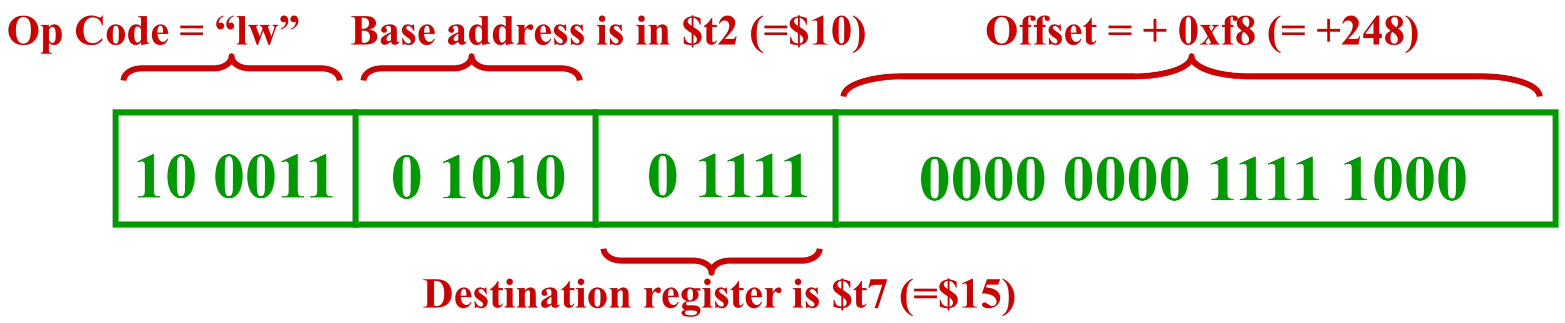
- Load instructions use the i-format.
- Load instructions transfer a 32-bit word, a 16-bit halfword, or an 8-bit byte into a designated register from memory. The “Op Code” defines the load instruction. Examples: 0x 23 (10 0011) = load word, load halfword is 0x21, and load byte is 0x20.
- $rs is the 5-bit address of a source register. The location of the data is calculated as [$rs] + Imm. = data address. ([ ] = “contents of”)
- The data is loaded from memory into $rt, the destination register.

Load Byte Instruction


Op Code = “lb”
Base address is in $t2 (=$10)
Offset = + 0x 30 (= + 48)
10 0000
0 1010
0 1111
0000 0000 0011 0000
Destination register is $t7 (=$15)

- Load byte (lb) is the simplest of the load instructions.
- An example is shown above: “Load the byte at the address comprised of [$t2] + 0x30 into destination register $t7.”
- In this case, only the single byte at the address ([$t2] + 48) is loaded into the lowest 8 bits of $t7. The other 24 bits of $t7 match the sign bit of the byte – this is called “sign extension.”
- Note: Memory is byte-addressable. That is, every single byte in MIPS memory has a unique, individual address.

Load Word



- Load word and load byte have identical formats; only the op codes are different.
- The example shown is: “Load the 32-bit word at memory location [$t2] + 0x f8 into register $t7.” The instruction would be written as:
  lw $t7,248($t2)
- Note: Since memory is byte addressed, the address of any word is the address of the first byte. Therefore a load word instruction means “load the byte at the given address, plus the next three bytes, into the destination register.”

Load Instruction Examples

- **Load instruction form: lw $rd,offset($rs). In a load, the sequence of events is [memory location (offset+[$rs])] → $rd. Examples:**
  - **lw $t7, 248($t2) – [memory address (248 + [$t2])] →$t7.**
  - **lb $s2,16($t5) – "Load bottom 8 bits of $s2 with the byte at (16+[$t5]).**
  - **lw $t9,0($a3) – "Load 32-bit word at address (0+[$a3]) into $t9.**
- **The SPIM assembler allows data to be labeled. Since data is labeled, the assembler allows loads with respect to the address label:**
  - **lw $t5, address – The 32-bit word at <u>address</u>→$t5.**
  - **lb $t0, num1 – Byte at <u>num1</u>→lowest 8 bits of $t0 (sign-extended).**
  - **lw $a1, const – The 32-bit word at <u>const</u>→$a1.**

*** [ ] = "Contents of.."**

Loading Labeled Data

- Instructions of the form "lb $rd,label" are pseudoinstructions.
- Suppose the instruction reads: lw $t7, const. The MIPS computer can only interpret instructions of the form "lw $rt,offset($rs)."
- For our simulated MIPS computer on PCSPIM, data storage always starts at memory location 0x 1001 0000.
- Suppose that "const" above is the label for data stored at memory address 0x 1001 00f8 (location 1001 0000 + 248 bytes [62 words]).
- SPIM would load the upper half of $at with 4097 (= 0x1001) via an lui instruction (lui $at, 4097). Thus $at = 1001 0000.
- Then SPIM would generate a second instruction: lw $t7, 248($at).
- The lw $t7, const instruction results in two MIPS instructions. Another reason why $at is reserved.

Examples: Loading Labeled Data

- **Remember that all load instructions follow the "indirect register plus offset" memory addressing scheme, i.e.: lw $rd,offset($rs)**
- **The SPIM assembler uses $at to convert load pseudoinstructions involving labels to the standard format for the MIPS computer.**
- **Assume labeled memory locations mem1 (location 0x1001 0020), mem2 (0x1001 0100), parm (0x1001 0000), and data5 (0x1001 1000). Then (since 4097 = 0x 1001):**
  - **lw $t2, mem1 assembles as lui $at,4097; lw $t2,32($at)**
  - **lw $s4, mem2 assembles as lui $at,4097; lw $s4,256($at)**
  - **lb $s7, parm → lui $at,4097; lb $s7,0($at) – Could also omit the 0.**
  - **lb $t8, data5 → lui $at,4097; lb $t8,4096($at)**
- **These "translations" are visible in the text window of the SPIM assembler**

Why the Complicated Load Routine?

- A natural question is: "Why does the MIPS use such a complicated load/store address calculation?"
- Easy answer – not enough bits:

– Op code for load/store	-- 6 bits
– Register source/destination of data	-- 5 bits
– Memory address[1]	-- 32 bits
– Total	-- 43 bits

  - But MIPS computer data word is only 32 bits!
- Therefore a complicated addressing technique is necessary to "get around" limited size of data word.[2]

Notes: 1 – Each byte has a unique address (up to 4 Gbyte); 2 – Thus the push for 64-bit CPU's!

Other Load Instructions

- **We will not use the load halfword instructions in examples (although you are welcome to use them in programming if you wish).**
- **We will also not use the "load unsigned halfword" or "load unsigned byte" instructions, as they are only marginally useful in our programs.**
- **Other instructions that are available, but which will not be covered here, include load word coprocessor, load doubleword, and several instructions that may cause memory loads that are not aligned with memory.**

Alignment in Load Word Instructions

- Alignment means "properly addressing" a word.
- Remember, all memory addresses are byte addresses.
- Since MIPS words are 32 bits (4 bytes), any word address is always the address of the first byte of that word.
- All memory is byte-addressed. Addresses are positive numbers, from 0x 0000 0000 to 0x ffff ffff.
- Since words are 4 bytes, word addresses end in binary "00." This means that hex memory addresses always end only in 0x0, 0x4, 0x8, or 0xc.
- Therefore, aligned word addresses end in 0x 0, 4, 8, or c.

Alignment (2)

- **Load word (lw) always loads aligned words from memory.**
- **However, some instructions load words (or parts of words) into memory that may not be properly aligned:**
  - **Load word left (lwl); load word right (lwr); unaligned load halfword (ulh); unaligned load word (ulw)**
- **Use of these instructions may cause the incorrect loading of data in some cases, and should in general NOT be used, unless the programmer is very sure of the result. In general, for our exercises, avoid these instructions.**

Load Address Instruction

- Load Address (la) is a very useful pseudo instruction.
- Form: la $t2, mem1 – address of the data labeled "mem1" is loaded into register t2.
- The "la" instruction is used as a part of syscall 4. Register $a0 is loaded with the starting address of an output character string.
  - la $a0, greeting
  - li $v0, 4
  - syscall

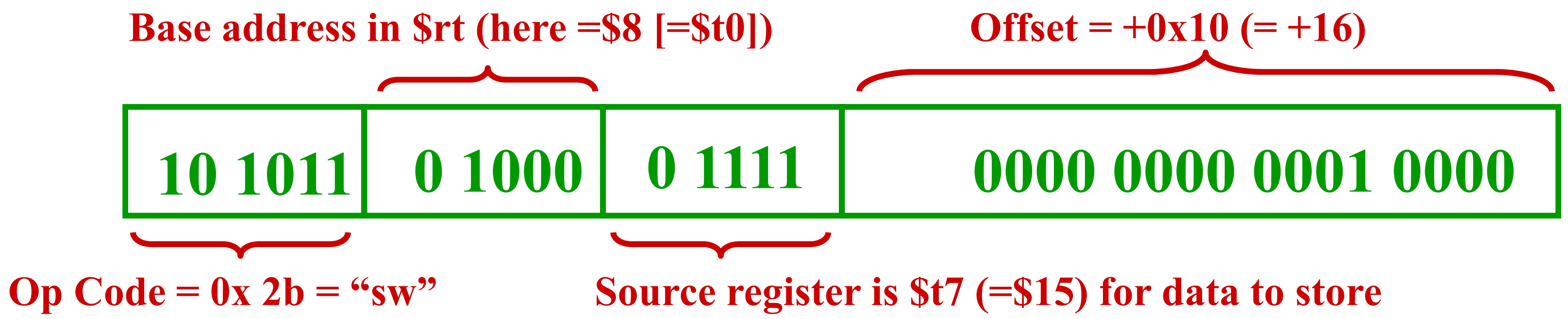
The address of the first ASCII character in the string labeled "greeting" is loaded in $a0. Execution of syscall 4 then prints out this ASCII string.

- Load address is also useful in constructing loops, where an address is loaded into a register, and the register is incremented for each pass through the loop.

Store Instructions

- Store instructions have exactly the same format as load, and form memory addresses the same way:
  - sb $t2, 24($s2) – The low 8 bits of $t2 are stored at the memory byte address comprised of [$s2] + 24.
  - sw $s3, 8($s0) – The 32-bit contents of $s3 are stored at location ([$s0]+8) plus the next 3 consecutive byte addresses.
- When storing to a labeled location, the store becomes a pseudo instruction, just as for load:
  - sb $t3, const – This becomes lui $1, 4097; sb $t3, [offset]($at).
  - Similarly, sw $t4, num → lui $1, 4097; sw $t4, [offset]($at).

Store Word Instruction Form


Base address in $rt (here =$8 [=$t0])
Offset = +0x10 (= +16)
10 1011
0 1000
0 1111
0000 0000 0001 0000
Op Code = 0x 2b = "sw"
Source register is $t7 (=$15) for data to store

- **Load and store have identical formats, except for the op codes. Here, the sw op code is 0x 2b (=10 1011).**
- **The above instruction is coded as: sw $t7,16($t0)**
- **"Store byte" (sb) would also be the same except that the op code is 0x28 (= 10 1000).**
- **As with load, there are other store instructions such as store halfword (sh), some of which may store unaligned data. In general, <u>avoid these instructions as a novice</u>.**

Reading the Memory Windows in SPIM

- **The following is a brief tutorial on the SPIM data window.**
- **Remember this PCSPIM convention (all SPIM readouts in hex):**
  - **Programs begin at memory location 0x 0040 0000**
  - **Data memory always starts at 0x 1001 0000**
  - **Memory is byte-addressed**
- **The address of a 4-byte word is the address of the first byte.**
- **Memory addresses are positive binary numbers (i.e., not 2's complement), starting at 0x 0000 0000 and going to 0x ffff ffff.**
- **Because memory is byte-addressed, the order in which the four bytes of a word in memory are assembled to form a number or an instruction is a matter of convention (example on next slide).**

Byte addresses	Contents		Word Addresses
[0x00000007]	0xce	>	
[0x00000006]	0x8a	>	
[0x00000005]	0x46	>	
[0x00000004]	0x02	>	0x00000004
[0x00000003]	0xef	}	
[0x00000002]	0xcd	}	
[0x00000001]	0xab	}	
[0x00000000]	0x12	}	0x00000000

How do we determine which is the "first byte" of a word? Two common conventions:*

BIG-ENDIAN byte ordering (most architectures other than VAX or 80x86): The words at addresses 0x00000000 and 0x00000004 in the above example are read as 0x12abcdef and 0x02468ace, respectively. The most significant ("biggest") byte has the lowest address; that byte address is the address of the whole word. (The word address is the address of the "big end".)

LITTLE-ENDIAN byte ordering (used in the 80x86 architecture/all PCs): The words at addresses 0x00000000 and 0x00000004 in the above example are read as 0xefcdab12 and ce8a4602, respectively. The least significant ("littlest") byte has the lowest address; that byte address is the address of the whole word. (The word address is the address of the "little end".)

*** Possibly suggested by the famous Jonathan Swift story, Gulliver's Travels.**

Note that the store algorithm influences how the data is interpreted on the previous slide. For example, the word at address 0x00000000 would be read as 313,249,263 (decimal) in the 32-bit two's complement representation if stored "big endian," but –271,733,998 if stored "little endian." The word at address 0x00000004 would be interpreted as 38,177,486 (decimal) in big endian, but as a negative number in little endian, –829,798,910. You need to know the data format before you can read it!

Data Memory Window Read-outs:

SPIM runs on both big-endian and little-endian architectures. We will look at a big-endian example first, since that is the MIPS R-2000 format. Since our MIPS simulation on the PC uses the Intel little-endian format, we also show that format later on. Note that a large number of architectures are Big-Endian, including the Motorola 680x0, IBM PowerPC, Sun SPARC, DEC Alpha, HP PA-RISC, and MIPS.

Following is part of a SPIM display of the data segment of a program, with extra annotations to explain what's going on. We assume Big-Endian for this first example. → (next page)

Big-Endian Data Memory Read-out

[0x10010000]	0x43000000	0x00007fff	0x46fffe00	0x48656c6c
address of first word on this line is in brackets	address is 0x10010000	address is 0x10010004 = add. of 1st word + 4	address is 0x10010008 = add. of 1st word + 8	address is 0x1001000c = add. of 1st word + c
[0x10010010]	0x6f2c2077	0x6f726c64	0x21004865	0x6c6c6f2c
address of first word on this line is in brackets	address is 0x10010010	address is 0x10010014 = add. of 1st word + 4	address is 0x10010018 = add. of 1st word + 8	address is 0x1001001c = add. of 1st word + c

Byte addressing examples:

The byte at address 0x1001000d is 0x65, and it is "more significant" than the byte at address 0x1001000f, which is 0x6c, in the big-endian addressing format. Thus the word at address 0x1001000c is read 0x48656c6c.

Little-Endian Data Memory Read-out

[0x10010000]	0x43000000	0x00007fff	0x46fffe00	0x48656c6c
address of first word on this line is in brackets	address is 0x10010000	address is 0x10010004 = add. of 1st word + 4	address is 0x10010008 = add. of 1st word + 8	address is 0x1001000c = add. of 1st word + c
[0x10010010]	0x6f2c2077	0x6f726c64	0x21004865	0x6c6c6f2c
address of first word on this line is in brackets	address is 0x10010010	address is 0x10010014 = add. of 1st word + 4	address is 0x10010018 = add. of 1st word + 8	address is 0x1001001c = add. of 1st word + c

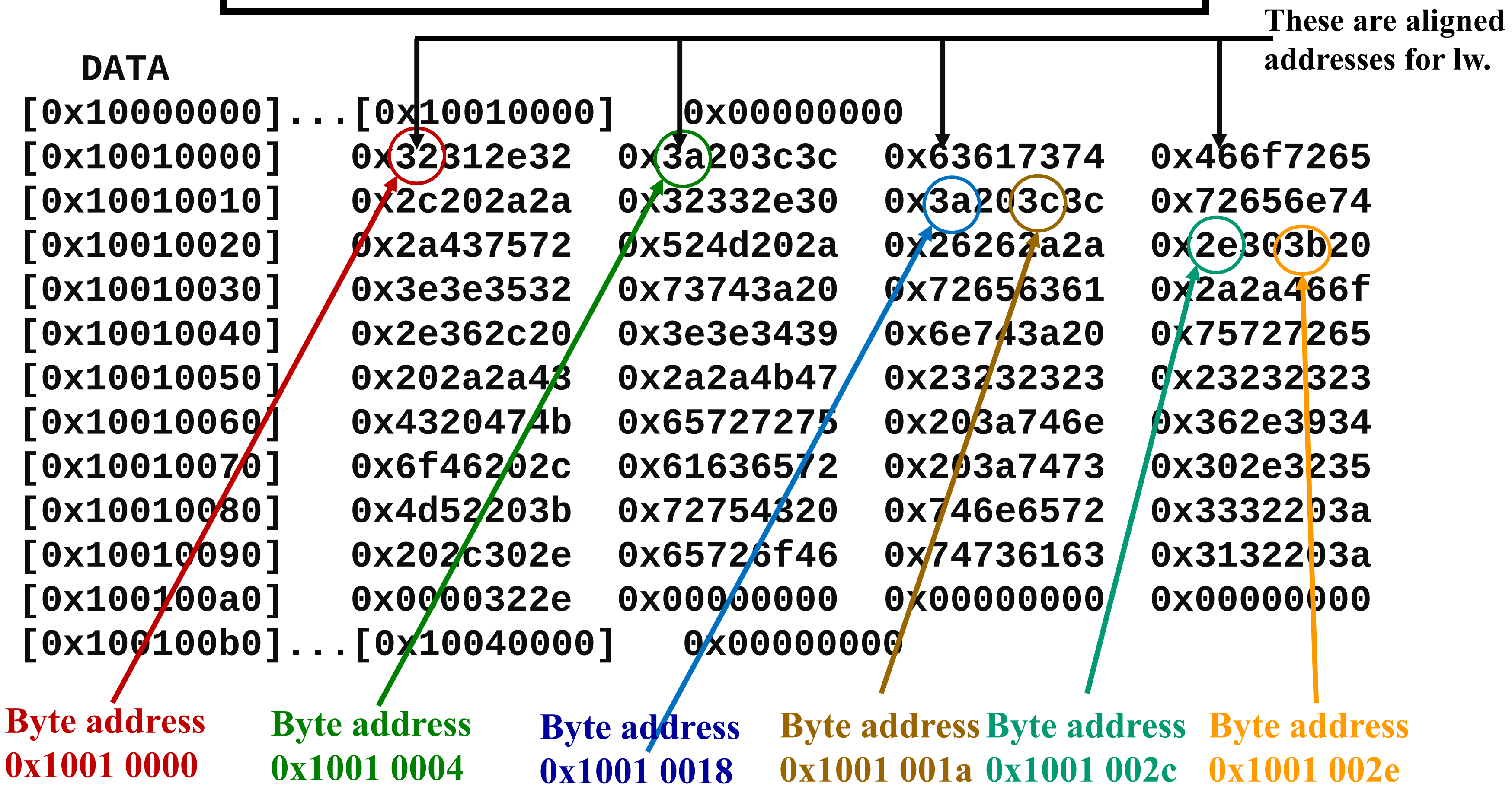
Byte addressing examples:
Data shown above is identical to that shown for that in the "big-endian" format. In this case, we assume the "little-endian" format is used, so that the byte at address 0x1001000f (0x6c) is now more significant, and the byte at address 0x1001000d (0x65) is now less significant. The word is now read (from most- to least-significant bytes) as 0x6c6c6548.

Two notes about Little-Endian formatting:

1. Since the lowest-order byte is given first, we must read the bytes in the word "backwards" to note what the actual 4-byte word is.
2. Ah, but not completely backwards! Each byte is give in the correct left-to-right order! Thus, as noted on the previous slide, the byte at the byte at address 0x1001000d is 0x65, not 0x56. Likewise, the byte at address 0x1001000e is 0x6c, not 0xc6. Bytes are in correct order, but the magnitude of the bytes in the word is in reverse order in little-endian formatting.

Unfortunately, this is the style on the PC! Thus, when looking at documentation in the book or in our notes, assume big-endian style (like the real MIPS computer). When reading a memory dump in PCSPIM, remember that it is little-endian. There are some more eccentricities related to PCSPIM and Little Endian that we will cover in the demonstrations.

SPIM Data Window Readout



```
   DATA
[0x10000000]...[0x10010000]   0x00000000
[0x10010000]   0x32312e32  0x3a203c3c  0x63617374  0x466f7265
[0x10010010]   0x2c202a2a  0x32332e30  0x3a203c3c  0x72656e74
[0x10010020]   0x2a437572  0x524d202a  0x26262a2a  0x2e303b20
[0x10010030]   0x3e3e3532  0x73743a20  0x72656361  0x2a2a466f
[0x10010040]   0x2e362c20  0x3e3e3439  0x6e743a20  0x75727265
[0x10010050]   0x202a2a43  0x2a2a4b47  0x23232323  0x23232323
[0x10010060]   0x4320474b  0x65727275  0x203a746e  0x362e3934
[0x10010070]   0x6f46202c  0x61636572  0x203a7473  0x302e3235
[0x10010080]   0x4d52203b  0x72754320  0x746e6572  0x3332203a
[0x10010090]   0x202c302e  0x65726f46  0x74736163  0x3132203a
[0x100100a0]   0x0000322e  0x00000000  0x00000000  0x00000000
[0x100100b0]...[0x10040000]   0x00000000
```

Reminder: Composing MIPS Programs

- Write SPIM programs in NotePad. If you wish, save the programs as name.s, per Pervin, to make visible in the "open" pop-up without clicking "all files."
- Program run procedure:
  1. Launch QtSpim or PCSPIM and open your program.
     - (Assembles until encountering an error, then stops)
  2. Open text editor, edit offending instruction, and re-save.
  3. Restart the program and proceed, stopping on next error, etc.
  4. If the program assembles, it will show up in the text window.
  5. Click "Go" (under "Simulator" tab). In the pop-up window labeled "Run Parameters," the starting address should show "0x 00400000." If not, enter that number. Click "OK" to run.
  6. If your program writes to the console, check it for desired result.
  7. Next, single step the program to watch instructions execute.

Program 1

- **Construct a program to input a number using system call 5 and then output it to the console using system call 1.**
- **In the same program, then output a carriage return/line feed using system call 11. The ASCII hex value of CR/LF is 0x0a.**
- **Finally, input the character string "Hello, world!" using system call 8 (which, you will remember, stores the string in memory, so you must reserve that memory using the .space command). Then, output the string using system call 4.**
- **To save time, don't worry about outputting directions such as "input a number," or "input 'Hello,world!'" You will therefore have to remember what the program wants you to do, when it halts, awaiting your input.**

Program 2

- Declare the following data in your data statement:

```
num1:  .word 35478
num2:  .word 98787
num3:  .word 0
num4:  .word 0
```

Remember: All data storage starts at memory address 1001 0000.

(The last two data declarations are "place holders.")

- Write a program to load num1 into a register and store it in the num3 position. Then load num2 into a register and store it in the num4 position. Then output num3 and num4 to the screen. Output a CR/LF between the numbers so that they are on different lines.
- Note: When loading num2 and storing it in num4, and when loading num4 to output it, use the register-contents-plus-offset method, NOT the address symbol.

Program 3

- Declare the following data:

```
A:    .word 57
B:    .word 23
C:    .word 42
D:    .word 0
```

- Write a program to load A, B, and C into $t0 one at a time and then store $t0 into D. When you load and store C, do so using the register-contents-plus-offset method. After storing C, output D using syscall 1, then end the program. Note that since you are only outputting D after storing C, the value output should be 42.