Memory transfer instructions

- How to get values to/from memory?
  - Also called memory access instructions

- Only two types of instructions
  - Load: move data from memory to register ("load the register")
    - e.g., lw $s5, 4($t6)        # $s5 <= memory[$t6 + 4]
  - Store: move data from register to memory ("save the register")
    - e.g., sw $s7, 16($t3)      # memory[$t3+16] = $s7

- In MIPS (32-bit architecture) there are memory transfer instructions for
  - 32-bit word: "int" type in C (lw, sw)
  - 16-bit half-word; "short" type in C (lh, sh; also unsigned lhu)
  - 8-bit byte: "char" type in C (lb, sb; also unsigned lbu)

Memory view

- Memory is a large, single-dimension 8-bit (byte) array with an address to each 8-bit item ("byte address")
- A memory address is just an index into the array

loads and stores give the index (address) to access
Memory view

- Memory is a large, single-dimension 8-bit (byte) array with an address to each 8-bit item ("byte address")

- A memory address is just an index into the array

0  BYTE #0
1  BYTE #1
2  BYTE #2
3  BYTE #3
4  BYTE #4
5  BYTE #5
6  BYTE #6
7  BYTE #7
8  BYTE #8

- loads and stores give the index (address) to access

address 4 gets this word
lw $t1,0($t0)
sw $t2,0($t0)
Effective Address calculation

- **Effective memory address** specified as immediate($register)
  - Register to keep the base address
  - Immediate to determine an offset from the base address
  - Thus, address is contents of register + immediate
  - The offset can be positive or negative, 16-bit value (uses I-format)
- Suppose base register $t0=64, then:
  1. `lw $t0, 12($t1)`  \[ \text{address} = 64 + 12 = 76 \]
  2. `lw $t0, -12($t1)`  \[ \text{address} = 64 - 12 = 52 \]
- MIPS uses this simple address calculation; other architectures such as PowerPC and x86 support different methods

Hint on addresses (la - load address)

- Often, you need to reference a particular variable.
  1. `data`  
  2. `var: .word 1000`  
  
  \[ \text{assembler directive to declare data (word)} \]
- How to reference `var`?
  1. `la $t0, var`  
  2. `lw $t1, 0($t0)`  
  
  \[ \text{puts the address of variable "var" into $t0} \]
  \[ \text{value at the address in $t0 is loaded into $t1} \]
- `la` is a “pseudo-instruction”. It is turned into a sequence to put a large address constant into $t0.
  1. `lui $at,upperbitsofaddress`
  2. `ori $t0,$1,lowerbitsofaddress`
Let’s try an in-class exercise together!

- Create a word (integer) variable “myVar”
- Give the variable the value 20
- Print the value to the console (Run I/O window)
- Terminate the program
- Extension: Add 10 to the value, store it to myVar, print it

To do this, we’ll need to use:
  - Data segment declaration with a word variable type
  - Instruction segment declaration
  - Load word instruction
  - Syscall instruction
  - Assorted la and li instructions

Let’s try an in-class example together

- Consider the C program and rewrite as MIPS

```c
void fun(void) {
    int a=10,b=20,c=30;
    a=a+10;
    b=0;
    c=a+b;
}
```
Machine code example

```c
void swap(int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

Let's try it in MARS!!!! (mips4.asm)

Memory organization

- 32-bit byte address
  - $2^{32}$ bytes with byte addresses from 0 to $2^{32} - 1$
  - $2^{30}$ words with byte addresses 0, 4, 8, ..., $2^{32} - 4$
- Words are aligned
  - 2 least significant bits (LSBs) of an address are 0s
- Half words are aligned
  - LSB of an address is 0
- Addressing within a word
  - Which byte appears first and which byte the last?
  - Big-endian vs. little-endian
    - "Little end (LSB) comes first (at low address)"
    - "Big end (MSB) comes first (at low address)"

Let's try it in MARS!!!! (mips5.asm)
More on alignment

- A misaligned access
  - Assume $t0=0$, then lw $s4, 3($t0)
- How do we define a word at address?
  - Data in byte 0, 1, 2, 3
    - If you meant this, use the address 0, not 3
  - Data in byte 3, 4, 5, 6
    - If you meant this, it is indeed misaligned!
    - Certain hardware implementation may support this; usually not
    - If you still want to obtain a word starting from the address 3 – get a byte from address 3, a word from address 4 and manipulate the two data to get what you want
- Alignment issue does not exist for byte access

Shift instructions

<table>
<thead>
<tr>
<th>Name</th>
<th>Fields</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</td>
<td>op  NOT USED</td>
<td>rt  rd  shamt  funct</td>
</tr>
<tr>
<td></td>
<td>shamt</td>
<td>shamt is “shift amount”</td>
</tr>
</tbody>
</table>

- Bits change their positions inside a word

  \(<op> <r_{target}> <r_{source}> <\text{shift\_amount}>\)

- Examples
  - sll $s3, $s4, 4  # $s3 \Leftarrow \text{8-bit}{\text{shift\_amount}}$
  - srl $s6, $s5, 6  # $s6 \Leftarrow \text{8-bit}{\text{shift\_amount}}$

- Shift amount can be in a register (“shamt” is not used)

- Shift right arithmetic (sra) keeps the sign of a number
  - sra $s7, $s5, 4

Let's try it in MARS!!! (mips6.asm)