CSI 333 – Lecture 15
MIPS Assembly Language (MAL): Part III
(ARRAYS IN MAL)
Arrays in MAL

- No built-in array type in MAL.
- Program must carry out address calculations.

**MIPS Memory Organization:**

- Memory is byte addressable.
- Each address is 32 bits long.
- 1 Word = 4 (successive) bytes.

*Note:* Byte addresses will be used for characters arrays.
Allocating Space for an Array

\textbf{C segment:}

\begin{verbatim}
int val[7];
\end{verbatim}

\textbf{MAL Segment:}

\begin{verbatim}
val: .word 0:7
\end{verbatim}

- No. of words allocated: 7
- Initial value stored in each word: 0

\textbf{C Segment:}

\begin{verbatim}
\end{verbatim}

- MAL equivalent requires \underline{address} calculation.
- Successive elements of an array have consecutive addresses.
Example:

Starting address of array
= Address of val[0]
= 200

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>204</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>208</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>212</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>216</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>220</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>224</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Starting address of array
= Address of val[0]
= 200

Formula for Address Calculation:

Address of val[i] = Start address of val + 4 × i.
Obtaining the Start Address:

- Start address is the address of the label val.
- Recall: MAL has a “load address” instruction (opcode: la).

C Statement:

\[
\text{val}[2] = -15;
\]

MAL Equivalent:

```
data
val: .word 0:7
.text
li $11, -15
la $10, val
sw $11, 8($10)
```

Note: The offset (or displacement) for val[2] is 8.
**General Formula:** If a system uses \( N \) bytes to represent a variable of type `int`, then for an `int` array `A`,

\[
\text{Address of } A[i] = \text{Start address of } A + N \times i.
\]

**C to MAL Translation Involving an Array:**

- See Handout 15.1.

**A Complete MAL Program Involving an Array:**

- See Handout 15.2.

**Arrays of Characters:**

- Each character needs 1 byte.
- Address calculation is easier. For a character array `a`,

\[
\text{Address of } a[i] = \text{Start address of } a + i
\]
C Segment:

```c
char name[15]; name[3] = '0';
```

MAL Equivalent:

```assembly
.data
name: .byte 0:15
.text
li $13, 48 #ASCII for '0'.
la $12, name
sb $13, 3($12)
```
sb: Store byte.

Stores the least significant byte of the specified register in the specified byte address.

Note the difference between sb and sw.
Opcodes \texttt{lbu} and \texttt{lb}

- \texttt{lbu} : Load byte \textit{unsigned} – Sets the leading 24 bits of the specified register to zero.
- \texttt{lb} : Load byte – Sets the leading 24 bits of the specified register to the sign bit of the specified byte.
Opcodes \texttt{lbu} and \texttt{lb} (continued)

**Important Notes**

- Use \texttt{lbu} when dealing with characters.
- Use \texttt{lb} when a byte is being treated as an 8-bit signed integer.

**Example:**

```asm
.data
v1: .byte 0xFF
.text
la $3, v1
lbu $5, 0($3)
lb $6, 0($3)
```

At the end of the above MAL segment:

- $5 contains \texttt{0x000000FF}.
- $6 contains \texttt{0xFFFFFFFF}.
Implementing a Character Stack

- Holds at most 100 characters.
- Operations: push, pop, is_empty, is_full.
- Use $15 as the stack pointer. This register will contain the address of the first empty location on the stack.

**Initial Declaration:**

```
char_stack: .byte 0:100
```

**Initializing $15:**

```
la $15, char_stack
```

(Stack grows to the right)
Implementing a Character Stack (continued)

**The push Operation:**

- Ignoring full stack.

- **Assumption:** Character to be pushed is in the least significant byte of $3$.

  ```
  sb  $3, 0($15)
  addi $15, $15, 1
  ```

**The pop Operation:**

- Ignoring empty stack.

- **Assumption:** The popped character must be stored in the least significant byte of $3$ and the other bytes of $3$ are made 0.

  ```
  addi $15, $15, -1
  lbu  $3, 0($15)
  ```
Implementing a Character Stack (continued)

**Operations is_empty and is_full:**

- **Method 1:** Can use another register for keeping track of the number of characters in the stack.

- **Method 2:** Use $15$ itself to check whether the stack is empty or full.
  - If the value stored in $15$ is equal to the starting address of the stack, then the stack is empty.
  - If the value stored in $15$ is equal to $(100 + \text{the starting address of the stack})$, then the stack is full.

Both `is_full` and `is_empty` are implemented as procedures that return values in $6$. (If the return value is 1, then the condition is true; otherwise, the condition is false.)
Implementing a Character Stack (continued)

**MAL Versions of is_empty and is_full:**

```mal
is_empty:  move $6, $0
            la $5, char_stack
            bne $5, $15, ret1
            addi $6, $6, 1
ret1:      jr $31

is_full:   move $6, $0
            la $5, char_stack
            addi $5, $5, 100
            bne $5, $15, ret2
            addi $6, $6, 1
ret2:      jr $31
```

**Exercise**

Write push and pop also as procedures. Make sure that they call is_full and is_empty respectively. (Bear in mind that push and pop may need to save/restore $31.)
**Goal:** To understand how `syscall` prints out an integer value.

**Example:**
- Suppose $17$ contains the decimal integer 389.
- To print out this value, the value 389 must be converted into the following sequence of three characters: '3', '8' and '9'.

**Method:**
- Generate the digits by successively dividing by 10. (This generates the digits from the least significant digit to the most significant digit.)
- Use a stack to get them in the right order.

**MAL Program:** Handout 15.3.
Two Dimensional Arrays in MAL

C Segment:

```c
char twod_str[5][10];
int val[10][8];
```

MAL Equivalent:

```mal
twod_str: .byte 0:50
val: .word 0:80
```

Storage Order:

- MIPS memory is a 1-dimensional array.
- Need to map a 2-dimensional array into memory.
- **Standard Techniques:** Row-major order and Column-major order.
- Address of an array element depends on the storage order.
(a) **Row-Major Order:** Used by many languages including C.

```c
int a[3][4];
```

```
0 1 2 3
0
1
2
Row 0
Row 1
Row 2
```
(b) **Column-Major Order:** Used in FORTRAN.

```c
int a[3][4];
0 1 2 3
0
1
2
a[0][0]
a[2][3]
3
a[1][0]
a[2][0]
a[1][3]
a[0][3]
Column 0
Column 1
Column 2
Column 3
```
**Formula for Row-Major Order:** Assume that we have a 2-dimensional array of type `int`.

Address of `a[i][j]` = Start address + 4 × i × No. of columns + 4 × j

**Example:**

```
0 1 2 3
0
2
1
```

Address of `a[1][2]` = Start + 4 × 1 × 4 + 4 × 2
= Start + 24

Address of `a[2][3]` = Start + 4 × 2 × 4 + 4 × 3
= Start + 44
**Formula for Column-Major Order:** Assume that we have a 2-dimensional array of type `int`.

\[
\text{Address of } a[i][j] = \text{Start address} + 4 \times j \times \text{No. of rows} + 4 \times i
\]

**Notes:**

- Above formulas work for 2-dimensional arrays of type `float` also.
- In general, for a data type that uses \( N \) bytes per element, the 4’s in the formula must be replaced by \( N \).
  - For the `char` data type, \( N = 1 \).
  - For the `double` data type, \( N = 8 \).
MAL Program with a 2-Dimensional Array:

- See Handout 15.4.

Exercise

Study the MAL program given in Handout 15.4. Write the MAL equivalent assuming that the array is stored in column-major order.