CSE 220: System Fundamentals I
Unit 14: MIPS Assembly: Multi-dimensional Arrays
Memory Alignment

• Perhaps at some point in your MIPS assembly programming you tried to perform a `lw` and received an error that the data you tried to access was not aligned on a word boundary

• When accessing multi-byte quantities (e.g., words), the CPU will read or write data only in an aligned fashion

• 1-byte values can be read from or written to at any address

• 2-byte values can be accessed at even-numbered addresses

• 4-byte values can be accessed only at addresses that are multiples of 4

• You will want to keep these rules in mind as you start to do more work with arrays in your MIPS assembly programs
Single-dimensional Arrays Recap

- To calculate the effective (actual) address of element \(i\) of a single-dimensional array we need to do some basic arithmetic:

\[
\text{address} = \text{base_addr} + i \times \text{elem_size_in_bytes}
\]

- We could then do a \texttt{lb} or \texttt{lw} at that address to get the value

- What about two-dimensional arrays? How do we calculate the effective address of an element?

- The answer depends on how we lay out the values in memory: \texttt{row-major order} or \texttt{column-major order}
Element Order

• Suppose we have this 3 × 5, 2D array:
  \[
  \begin{array}{ccccc}
  a & b & c & d & e \\
  f & g & h & i & j \\
  k & l & m & n & o \\
  \end{array}
  \]

• If we use row-major order, elements are stored in memory like this: \[a b c d e f g h i j k l m n o\]

• But if we use column-major order, elements are stored in this order: \[a f k b g l c h m d i n e j o\]

• The technique that is used is usually dependent on the programming language

• With MIPS we can use either because we have to lay out everything ourselves, anyway!

• For simplicity we will use row-major order in this course
Accessing Array Elements

• Assuming we use row-major order to store elements in a 2D array, the memory address of element \([i][j]\) is

\[
\text{addr} = \text{base\_addr} + \left( i \times \text{size\_of\_a\_row\_in\_bytes} + j \times \text{size\_of\_a\_column\_in\_bytes} \right)
\]

• This translates into:

\[
\text{addr} = \text{base\_addr} + \left( i \times \text{num\_columns} \times \text{elem\_size\_in\_bytes} + j \times \text{elem\_size\_in\_bytes} \right)
\]

\[
\text{addr} = \text{base\_addr} + \text{elem\_size\_in\_bytes} \times (i \times \text{num\_columns} + j)
\]
Example: 2d_array.asm

• This is a simple program that shows the calculation in practice
• It stores the values 65 through 79 (A through N) in a $3 \times 5$ 2D array of words stored in row-major order
The Frame Pointer ($fp$)

• You may recall this pair of pictures from earlier in the term.
• Figure (a) shows the stack before the callee is invoked
• In figure (b), the callee has been invoked
The Frame Pointer ($fp$)

- Consider the activation record (stack frame):
  - Suppose during execution of our function we need to make reference to the “additional arguments” section
  - Normally we would simple use an offset relative to $sp$
  - However, there’s a problem. What if the callee starts pushing/popping data on the stack during execution?
The Frame Pointer ($fp$)

- Although a value buried in the stack frame (e.g., additional arguments) will have a fixed memory address, its location relative to $sp$ will change as $sp$ itself changes due to pushes and pops of the runtime stack.
The Frame Pointer ($fp$)

• As an example, perhaps when the function begins executing we can use $40 (\$sp)$ to grab an extra argument from the additional arguments section

• But as time goes on, the function pushes and pops data to the stack, so that argument is sometimes at $56 (\$sp)$, sometimes at $40 (\$sp)$ again, then later it’s at $48 (\$sp)$

• This is going to make our code really confusing

• So instead of using $\$sp$ for our references into the stack, we can use the frame pointer, which is register $\$fp$

• At the start of the function’s execution we move $\$fp$, $\$sp$ and then do our accesses into the stack frame using $\$fp$, which will not change during the duration of the function call
The Frame Pointer ($fp$)

This is only just one example of how we could use $fp$. The main idea here is not to rely only on $sp$ to get data from the stack because $sp$ can change during a function call. Note that $fp$ is a preserved register.
The Frame Pointer ($fp$)

- Here’s another example of how the frame pointer is useful
- Suppose we have a complex function with a lot of paths through it: an if-statement with a lot of “else if” clauses
- Now in each case of the if-statement we push a different number of items onto the stack
- When the function is ready to return, we need to pop off everything we pushed, but we have lost track of how much data we have pushed!
- If we had set $fp$ equal to $sp$ at the start of the function call, we could now simply do move $sp$, $fp$