Memory and Data Structures

- Arrays
- Stacks
- Queues
We have seen labels and addresses point to pieces of memory storing:

• • • • •

• Memory is just a collection of bits
• We could use it to represent integers 0...2^{24}
• Or as an arbitrary set of bits
• Instead, treat memory as a giant array of bytes
  • Compiler or programmer decides what use to make of it.
• The element numbering starts at 0
• The element number is an address
• In C:
  \[ \text{char m[size_of_array];} \]
The data is placed in memory as, e.g.,

<table>
<thead>
<tr>
<th>address</th>
<th>memory</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>$0x??</td>
<td>mychar w/ no initial value</td>
</tr>
<tr>
<td>21</td>
<td>$0x0a</td>
<td>newline w/ initial value &quot;$n&quot;</td>
</tr>
</tbody>
</table>
Storage of Integers

- Use 4 bytes beginning at an address that is multiple of 4
- So at address 20, want bytes 20, 21, 22, and 23
- Use ‘lw’ (load word) instead of ‘lb’ (load byte)

```
.data
n: .word 1
n2: .word -1
newline: .byte \n
str1: .asciiz "Enter a number 

.text
main: puts str1
geti $t1
lw $t1, n
add $t2, $t1, $t1
sw $t2, n2
```
The assembler/linker will expand the `.data` segment to:

<table>
<thead>
<tr>
<th>address</th>
<th>memory</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>0x00</td>
<td>n: 1 in 2SC</td>
</tr>
<tr>
<td>41</td>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>0x01</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>0xff</td>
<td>n2: -1 in 2SC</td>
</tr>
<tr>
<td>45</td>
<td>0xff</td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>0xff</td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>0xff</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>0x0a</td>
<td>newline: ‘\n’ in ASCII</td>
</tr>
<tr>
<td>49</td>
<td>0x45</td>
<td>str1: ‘E’ in ASCII</td>
</tr>
<tr>
<td>50</td>
<td>0x6e</td>
<td>‘n’ in ASCII</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>0x72</td>
<td>‘r’ in ASCII</td>
</tr>
<tr>
<td>61</td>
<td>0x20</td>
<td>‘ ‘ in ASCII</td>
</tr>
<tr>
<td>62</td>
<td>0x00</td>
<td>‘\0’ in ASCII</td>
</tr>
</tbody>
</table>
And the program becomes:

```assembly
.text
__start:  puts 49          # expects an address for a string
         lw $t1, 40       # expects an address
         add $t2, $t1, $t1 # expects "values"
         sw $t2, 44      # expects an address
```

- Real machine code has addresses for variables, not labels.
- Later, we will show how to store instructions and get ride of branch labels. Then the `.text` segment will look similar to the `.data` segment
Consider:

```
.data
n: .word 0x61626364       #ASCII for 'a', 'b', 'c', 'd'
addr: .word

.text
main:
    lb  $t0, n
    putc  $t0
    putc  '\n'
```

Do you think an ‘a’ or a ‘d’ is printed?
Endian Issues

- Answer: ‘a’ since simp/sim on Sun SPARC is Big Endian

  40: 0x61
  41: 0x62
  42: 0x63
  43: 0x64

  \$s1 \leftarrow M[40] = 0x61626364

Big Endian: smallest address is most significant (biggest)
Little Endian: smallest address is least significant (littlest)

- Answer: ‘d’

  40: 0x64
  41: 0x63
  42: 0x62
  43: 0x61

  \$s1 \leftarrow M[40] = 0x61626364

Bi-Endian: Select data format

- What happens when Suns and PCs are hooked together?
**Endian Issues**

- **Little Endian** – address LSB
  - VAX, Alpha, x86, PDP-11
- **Big Endian** – address MSB
  - IBM 360/370/380/390, Motorola 680x0, Sparc, DLX

**Programmable endian on data (why not instructions?)**
- i860 (Little-endian instructions)
- PowerPC (Big-endian instructions)
- MIPS

When accessing values smaller than a word must also consider alignment, sign extension, and effect on high bytes of register.

<table>
<thead>
<tr>
<th>Word Address</th>
<th>Bytes MSB → LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3 2 1 0</td>
</tr>
<tr>
<td>4</td>
<td>7 6 5 4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Word Address</th>
<th>Bytes MSB → LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 1 2 3</td>
</tr>
<tr>
<td>4</td>
<td>4 5 6 7</td>
</tr>
</tbody>
</table>
Arrays

• Array implementation is important
  • Most assembly languages have no concept of arrays
  • From an array, any other data structure we might want can be built
• Properties of arrays:
  • Each element is the same size
  • Char = 1 byte
  • Integer = 1 word
  • Elements are stored contiguously
  • First element at the smallest memory address
• In assembly language we must
  • Allocate correct amount of space for an array
  • Map array addresses to memory addresses
Arrays

MAL declarations of arrays within memory
• To allocate a portion of memory (more than a single variable’s worth)

```
variblename: type initvalue: numelements
```

• type is as before - .byte, .word, or .float
• numelements is just that, numbering starts at 0 (as in C)
• initvalue is a value given to each element of the array
Arrays

New directive:

name: .space numberofbytes

• `space` allocates space (bytes) within memory without giving an initial value.
• The type of the data within this space cannot be inferred.
• 8 character elements, number 0 – 7, initialized to 0
  arrayname: .byte 0:8
• 18 bytes of memory
  name: .space 18
Arrays

Calculating the address of an array element

array1: array [6..12] of char; /* Pascal */
char array2[7] /* C */

• If base address of array is 25
• Byte address of array1[10] = 25 + (10 - 6) = 29

• Base address + distance from the first element
In MAL:

array1: .byte 0:7

- Want the 5th element
- Array[4] is at the address array1 + 4
- If element [0] is at address 25, byte address of array[4] = 25 + 4
How do you get the address of array1?

- Take the address array1 (25 in the example) and put it into a register.
- Keep clear the difference between an address and the contents of an address.
Arrays Addresses

- To reference array1[4] in MAL, write the code...

```
la     $t0, array1
add    $t1, $t0, 4
```

- If we wanted to decrement element number 5 by 1...

```
lb     $t4, ($t1)
sub    $t4, $t4, 1
sb     $t4, ($t1)
```
A 32-bit word is 4 bytes

\[
\text{int array2[6]; // C++}
\]

\[
\text{array2a: .word 0:6 # MAL}
\]

\[
\text{array2b: .space 24 # MAL}
\]

Looks like

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
<td>84</td>
<td>88</td>
<td><strong>92</strong></td>
<td>96</td>
<td>100</td>
</tr>
</tbody>
</table>

And the byte address of \textbf{array2a[3]} = 80+4(3) = 92

Must use:

- Where the array starts (\textbf{base address})
- \textbf{Size} of an element in bytes
- What the first element is numbered
- Byte address of \textbf{element[x]} = base+size(x-first index)
\textit{lw vs. lb}

- The \textbf{type} of \textbf{lb} is ‘byte’ and of \textbf{lw} is ‘word’
- Use \textbf{lb} to manipulate bytes, \textbf{lw} words.
- The \textbf{index} of both \textbf{lb} and \textbf{lw} is a byte address.
  - \textbf{lb} \texttt{a} \#address the \texttt{a}’th byte in memory
  - The following indexes byte-array element \texttt{t1}
    \begin{verbatim}
    base: .byte 0:n
    .text
    la $t0, base
    add $t2, $t1, $t0
    lb $t3, 0($t2)
    \end{verbatim}
- \textbf{lw} \texttt{$t1, a} \# addresses the 4 bytes starting at address \texttt{a}
- Both \textbf{lw} and \textbf{lb} should \textbf{always} be used with a base address and allocated space.
2-Dimensional arrays are more complicated

- Memory is a 1-D array
- Must map 2-D array to 1-D array
- Arrays have rows and columns
  - \( r \times c \) array
  - \( r \) rows
  - \( c \) columns
- Indexing a 4 \( \times \) 2 array:
Two sensible ways to map 2-D to 1-D.

**Row major order:**
(rows are all together)

| 0,0 | 0,1 | 1,0 | 1,1 | 2,0 | 2,1 | 3,0 | 3,1 |

**Column major order:**
(columns are all together)

| 0,0 | 1,0 | 2,0 | 3,0 | 0,1 | 1,1 | 2,1 | 3,1 |
2-Dimensional Arrays

How do you calculate addresses in a 2-D array?

• Row Major:

• Column Major:
• Many HLL’s have bounds checking (not C!!!)
• Assembly languages have no implied bounds checking
• Your program is in total control of memory
• With a 5 x 3 array, what does the following address?

array: .word 0:100

.text
la $t1, array
add $t1, $t1, 15
lw $t0, ($t1)

• Bounds checking is often a good idea!!
• Most C development environments (Sun, Alpha, Visual?) include optional bounds checking.
2-Dimensional Arrays

Summary:

• Row/Column major (storage order)
• Base address
• Size of elements
• Dimensions of the array

How about 3-D arrays?
Stacks

- A data structure that stores data in the reverse order that it is used.
- The data is not known until **run time**.
- The classic analogy is a stack of dishes
  - You can put single plates on the stack
  - You can take plates off the stack
  - Last in, First out (LIFO)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3rd on</td>
<td>2nd on</td>
<td>1st on</td>
</tr>
</tbody>
</table>

- Data is added or **pushed** onto the stack.
- Data is removed or **popped** off the stack.
Stacks

- Printing out a positive integer, character by character
- Push LSB to MSB
- Pop MSB to LSB (LIFO)

integer = 1024

if integer == 0 then
  push '0'
else
  while integer != 0
    digit ← integer mod base
    char ← digit + 48
    push char onto stack
    integer ← integer div base
  while stack is not empty
    pop char
    put char
Implementation of a Stack

One implementation of a stack out of an array.

- Index the **top of stack** (tos), with a **stack pointer** (sp).

  initial state, sp points at top of stack

- **SP** is a variable that contains the address of the empty location at the top of the stack (when empty).
Stack Implementation

stack: .word 0:50
stackend: .word stack+4x50

Or

stack: .space 200

• Labels can be used as initial values
• The address (label) of the stack gets put into the variable
• Identical to use of
  ```
  la $s0, stack      # initialization of stack pointer
  ```
• PUSH
  ```
  sw   $s1, 0($s0)  # $s1 has data to push
  add  $s0, $s0, 4
  ```
• POP
  ```
  sub  $s0, $s0, 4
  lw   $s1, 0($s0)
  ```
A stack could instead be implemented such that the stack pointer points to a FULL location at the top of the stack.

- **PUSH operation:**
  
  - add $s0, $s0, 4
  - sw $s1, 0($s0)

- **POP operation:**
  
  - lw $s1, 0($s0)
  - sub $s0, $s0, 4

- The stack could “grow” from the end of the array towards the beginning.
A **queue** is a FIFO (First In, First Out).

- The classic analogy of a queue is a line.
  - Person gets on the end of the line (the **Tail**),
  - Waits,
  - Gets off at the front of the line (the **Head**).
- Getting into the queue is an operation called **enqueue**.
- Taking something off the queue is an operation called **dequeue**.
- It takes 2 pointers to keep track of the data structure,
  - Head (let’s use $s5$)
  - Tail always points to empty element ($s7$)
Queues

Initial state:

After 1 enqueue operation:

After another enqueue operation:
After a dequeue operation:

Like stacks, when an item is removed from the data structure, it is physically still present, but correct use of the structure cannot access it.
Queues

Implementation

• Storage:
  Queue: .word 0:infinity  # assume infinite for now
  .text
  la $s5, queue  # head
  la $s7, queue  # tail

• Enqueue (item):
  sw $t0, 0($s7)  # $t0 has data to store
  add $s7, $s7, 4

• Dequeue (item):
  beq $s5, $s7, queue_empty
  lw $t1, 0($s5)
  add $s5, $s5, 4

• How to add overflow, underflow detection?
Circular Queues

- To avoid infinite array, wrap around from end to beginning.
- Head == Tail means empty
- Head points to first item (for next dequeue)
- Tail points to empty location (for next enqueue)
Circular Queues

- **Storage:**

  queue: .word 0:queue_size
  queue_end: .word queue+4*queue_size
  head: .word queue  # static la $s5, queue
  tail: .word queue  # static la $s7, queue

- **Enqueue (item)**

  sw $t0, 0($s7)  # data to enqueue is in $t0
  add $s7, $s7, 4
  la $t1, queue_end
  blt $s7, $t1, continue1
  la $s7, queue  # wrap around
  continue1:
Circular Queues

• Dequeue (item):
  
  ```
  beq   $s5, $s7, queue_empty
  lw    $t0, 0($s5)
  add   $s5, $s5, 4
  la    $t1, queue_end
  blt   $s5, $t1, continue2
  la    $s5, queue # wrap around
  
  continue2:
  ```

• How to add overflow, underflow detection?