The Memory Subsystem

- memory subsystem typically provides capability to load or store **bytes** (not bits)
  - 1 byte == 8 bits (on general purpose modern machines)
- each byte has unique **address**, think of:
  - memory as implementing a gigantic array of bytes
  - and the address is the array index
- typically, a small (1,2,4,8,...) group of bytes can be loaded/stored in a single operation
- general purpose computers typically have complex cache systems to improve memory performance
  - if we have time we'll look at cache systems a little, late in this course

Virtual Memory - Quick Summary

- we'll come back to **virtual memory** if any time left in week 10
- operating systems on general purpose computers typically provide **virtual memory**
- **virtual memory** make it look to every running program that it has entire address space
  - hugely convenient for multi-process systems
- disconnects addresses running programs (processes) use from actual RAM address.
- operating system translates (virtual) address a process uses to an physical (actual) RAM address.
- translation needs to be really fast - needs to be largely implemented in hardware (silicon)
- **virtual memory** can be several times larger than actual RAM size
- multiple processes can be in RAM, allowing fast switching
- part of processes can be load into RAM on demand.
- provides a mechanism to share memory between processes.
**Address Size**

- most modern general purpose computers use 64-bit addresses
  - CSE servers use 64-bit addresses
- some (older) general purpose computers use 32-bit addresses
- many special purpose (embedded) CPUs use 32-bit addresses
  - but some use 64-bit addresses
  - some use 16-bit addresses
- on the MIPS32 machine implemented by mipsy, all addresses are 32-bit
  so in COMP1521 assembler we’ll be using 32-bit addresses
- there are 64-bit MIPS CPUs

**Accessing Memory on the MIPS**

- addresses are 32 bits
- only load/store instructions access memory on the MIPS
  - 1 byte (8-bit) loaded/stored with `lb/sb`
  - 2 bytes (16-bit) called a half-word, loaded/stored with `lh/sh`
  - 4 bytes (32-bits) called a word, loaded/stored with `lw/sw`
- memory address used for load/store instructions is sum of a specified register and a 16-bit constant (often 0) which is part of the instruction
- for `sb` & `sh` operations low (least significant) bits of source register are used.
- `lb/lh` assume byte/halfword contains a 8-bit/16-bit **signed** integer
  - high 24/16-bits of destination register set to 1 if 8-bit/16-bit integer negative
- unsigned equivalents `lbu` & `lhu` assume integer is **unsigned**
  - high 24/16-bits of destination register always set to 0
- signed and unsigned integer representations covered later in course

**MIPS Load/Store Instructions**

<table>
<thead>
<tr>
<th>assembly</th>
<th>meaning</th>
<th>bit pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lb r_t, I(r_s)</code></td>
<td><code>r_t = mem[r_s+I]</code></td>
<td><code>100000ssssstttttIIIIIIIIIIIIIIII</code></td>
</tr>
<tr>
<td><code>lh r_t, I(r_s)</code></td>
<td><code>r_t = mem[r_s+I]</code></td>
<td><code>100001ssssstttttIIIIIIIIIIIIIIII</code></td>
</tr>
<tr>
<td><code>lw r_t, I(r_s)</code></td>
<td><code>r_t = mem[r_s+I]</code></td>
<td><code>100011ssssstttttIIIIIIIIIIIIIIII</code></td>
</tr>
<tr>
<td><code>sb r_t, I(r_s)</code></td>
<td><code>mem[r_s+I] = r_t &amp; 0xff</code></td>
<td><code>101000ssssstttttIIIIIIIIIIIIIIII</code></td>
</tr>
<tr>
<td><code>sh r_t, I(r_s)</code></td>
<td><code>mem[r_s+I] = r_t &amp; 0xff</code></td>
<td><code>101001ssssstttttIIIIIIIIIIIIIIII</code></td>
</tr>
<tr>
<td><code>sw r_t, I(r_s)</code></td>
<td><code>mem[r_s+I] = r_t &gt;&gt; 8 &amp; 0xff</code></td>
<td><code>101011ssssstttttIIIIIIIIIIIIIIII</code></td>
</tr>
</tbody>
</table>
# simple example of load & storing a byte
# we normally use directives and labels
# lb & sb require address in a register, but mipsy will do this for us

def main:
    li $t0, 42
    sb $t0, 0x10000000 # store 42 in byte at address 0x10000000
    lb $a0, 0x10000000 # load $a0 from same address
    li $v0, 1 # print $a0 which will nows contain 42
    syscall
    li $a0, '\n' # print '
'
    li $v0, 11
    syscall
    li $v0, 0 # return 0
    jr $ra

source code for load_store_no_label.s
https://www.cse.unsw.edu.au/~cs1521/24T1/ COMP1521 24T1 — MIPS Data 7 / 75

Assembler Directives

mipsy has directives to initialise memory, and to associate labels with addresses.

```
.text # following instructions placed in text segment

.data # following objects placed in data segment

a: .space 18 # int8_t a[18];
.i: .align 2 # align next object on 4-byte addr
i: .word 42 # int32_t i = 42;
v: .word 1,3,5 # int32_t v[3] = {1,3,5};
h: .half 2,4,6 # int16_t h[3] = {2,4,6};
b: .byte 7,5 # int8_t b[5] = {7,7,7,7,7};
f: .float 3.14 # float f = 3.14;
s: .asciiz "abc" # char s[4] {'a','b','c','\0'};
t: .ascii "abc" # char t[3] {'a','b','c'};
```

source code for load_store_no_label.s
https://www.cse.unsw.edu.au/~cs1521/24T1/ COMP1521 24T1 — MIPS Data 7 / 75

Code example: storing and loading a value with a label

# simple example of load & storing a byte
# we normally use directives and labels
# lb & sb require address in a register, but mipsy will do this for us

def main:
    li $t0, 42
    sb $t0, answer # store 42 in byte at address labelled answer
    lb $a0, answer # load $a0 from same address
    li $v0, 1 # print $a0 which will nows contain 42
    syscall
    li $a0, '\n' # print '
'
    li $v0, 11
    syscall
    li $v0, 0 # return 0
    jr $ra

.data
answer:
    .space 1 # set aside 1 byte and associate label answer with its address

source code for load_store_label.s
https://www.cse.unsw.edu.au/~cs1521/24T1/ COMP1521 24T1 — MIPS Data 8 / 75
**Code example: storing and loading a value with address in register**

```assembly
# simple example of storing & loading a byte
main:
    li $t0, 42
    la $t1, answer
    sb $t0, 0($t1)  # store 42 in byte at address labelled answer
    lb $a0, 0($t1)  # load $a0 from same address
    li $v0, 1       # print $a0 which will nows contain 42
    syscall
    li $a0, '\n'   # print '\n'
    li $v0, 11
    syscall
    li $v0, 0       # return 0
    jr $ra
.data
answer:        # set aside 1 byte and associate label answer with its address
```

**Source code for load_store.s**

[https://www.cse.unsw.edu.au/~cs1521/24T1/](https://www.cse.unsw.edu.au/~cs1521/24T1/)

### Setting A Register to An Address

- Note the `la` (load address) instruction is normally used to set a register to a labelled memory address.

- `la $t8, start`

- `mipsy` converts labels to addresses (numbers) before a program is run,
  - no real difference between `la` and `li` instructions
  - For example, if `vec` is the label for memory address `0x10000100` then these two instructions are equivalent:
    ```assembly
    la $t7, vec
    li $t7, 0x10000100
    ```

- In both cases the constant is encoded as part of the instruction(s).

- Neither `la` or `li` access memory! They are very different to `lw` etc

### Specifying Addresses: Some `mipsy` short-cuts

- `mipsy` allows the constant which is part of load & store instructions can be omitted in the common case it is 0.

- `sb $t0, 0($t1)  # store $t0 in byte at address in $t1
  sb $t0, ($t1)   # same`

- For convenience, MIPSY allows addresses to be specified in a few other ways and will generate appropriate real MIPS instructions

- `sb $t0, x       # store $t0 in byte at address labelled x
  sb $t1, x+15   # store $t1 15 bytes past address labelled x
  sb $t2, x($t3) # store $t2 $t3 bytes past address labelled x`

- These are effectively pseudo-instructions.
- You can use these short cuts but won’t help you much
- Most assemblers have similar short cuts for convenience
MIPSY Memory Layout

<table>
<thead>
<tr>
<th>Region</th>
<th>Address</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>.text</td>
<td>0x0040000000</td>
<td>instructions only; read-only; cannot expand</td>
</tr>
<tr>
<td>.data</td>
<td>0x1000000000</td>
<td>data objects; read/write; can be expanded</td>
</tr>
<tr>
<td>.stack</td>
<td>0x7fffffff</td>
<td>this address and below; read/write</td>
</tr>
<tr>
<td>.ktext</td>
<td>0x80000000</td>
<td>kernel code; read-only; only accessible in kernel mode</td>
</tr>
<tr>
<td>.kdata</td>
<td>0x90000000</td>
<td>kernel data; only accessible in kernel mode</td>
</tr>
</tbody>
</table>

Data Structures and MIPS

C data structures and their MIPS representations:

- **char** ... as byte in memory, or register
- **int** ... as 4 bytes in memory, or register
- **double** ... as 8 bytes in memory, or $f? register
- **arrays** ... sequence of bytes in memory, elements accessed by index (calculated on MIPS)
- **structs** ... sequence of bytes in memory, accessed by fields (constant offsets on MIPS)

A **char**, **int** or **double**

- can be stored in register if local variable and no pointer to it
- otherwise stored on stack if local variable
- stored in data segment if global variable

Global Variables

Labels and *directives* used to allocate space for global variables in the .data segment.

```
.data
a:
    .word 16 # int a = 16;
b:
    .space 4 # int b;
c:
    .space 4 # char c[4];
d:
    .byte 1, 2, 3, 4 # char d[4] = \{1, 2, 3, 4\};
e:
    .byte 0:4 # char e[4] = \{0, 0, 0, 0\};
f:
    .asciiz "hello" # char *f = "hello";
    .align 2
g:
    .space 4 # int g;
```

source code for sample_data.s
#include <stdio.h>
int global_counter = 0;
int main(void) {
    // Increment the global counter.
    // The following is the same as global_counter = global_counter + 1 (generally)
    global_counter++;
    printf("%d", global_counter);
    putchar('\n');
}

Source code for global_increment.c

Incrementing a Global Variable: MIPS

lw $t1, global_counter
addi $t1, $t1, 1
sw $t1, global_counter # global_counter = global_counter + 1;
# Method 2: Explicitly load the address of
# global_counter into a register.
li $v0, 1
# syscall 1: print_int
la $t0, global_counter #
lw $a0, ($t0)
syscall # printf("%d", global_counter);
li $v0, 11 # syscall 11: print_char
li $a0, '\n'
syscall # putchar('\n');
li $v0, 0
jr $ra # return 0;
.data
global_counter:
.word 0 # int global_counter = 0;

Source code for global_increment.s

add: local variables in registers

C
int main(void) {
    int x, y, z;
    x = 17;
y = 25;
z = x + y;
    // ...
}

MIPS
main:
    # x in $t0
    # y in $t1
    # z in $t2
    li $t0, 17
    li $t1, 25
    add $t2, $t1, $t0
    # ...

Source code for global_increment.s
C
int x, y, z;
int main(void) {
    x = 17;
    y = 25;
    z = x + y;
}

MIPS (.data)
.data
    x: .space 4
    y: .space 4
    z: .space 4

MIPS (.text)
main:
    li $t0, 17
    la $t1, x
    sw $t0, ($t1)  # x = 17;
    li $t0, 25
    la $t1, y
    sw $t0, ($t1)  # y = 25;
    la $t0, x
    lw $t1, ($t0)
    la $t0, y
    lw $t2, ($t0)
    add $t3, $t1, $t2
    la $t0, z
    sw $t3, 0($t0)  # z = x + y;
    li $v0, 1  # syscall 1: print_int

source code for add_memory.s
https://www.cse.unsw.edu.au/~cs1521/24T1/ COMP1521 24T1 — MIPS Data 19 / 75

add variables in memory (initialized)
C
int x=17;
int y=25
int z;
int main(void) {
    z = x + y;
}

MIPS .data
.data
    x: .word 17
    y: .word 25
    z: .space 4

MIPS .text
main:
    la $t0, x
    lw $t1, ($t0)
    la $t0, y
    lw $t2, ($t0)
    add $t3, $t1, $t2
    la $t0, z
    sw $t3, 0($t0)  # z = x + y;
    li $v0, 1  # syscall 1: print_int

source code for add_memory_initialized.s
https://www.cse.unsw.edu.au/~cs1521/24T1/ COMP1521 24T1 — MIPS Data 20 / 75

add variables in memory (uninitialized)
C
int x, y, z;
int main(void) {
    x = 17;
    y = 25;
    z = x + y;
}

MIPS (.data)
.data
    x: .space 4
    y: .space 4
    z: .space 4

MIPS (.text)
main:
    li $t0, 17
    la $t1, x
    sw $t0, ($t1)  # x = 17;
    li $t0, 25
    la $t1, y
    sw $t0, ($t1)  # y = 25;
    la $t0, x
    lw $t1, ($t0)
    la $t0, y
    lw $t2, ($t0)
    add $t3, $t1, $t2
    la $t0, z
    sw $t3, 0($t0)  # z = x + y;
    li $v0, 1  # syscall 1: print_int

source code for add_memory.s
https://www.cse.unsw.edu.au/~cs1521/24T1/ COMP1521 24T1 — MIPS Data 21 / 75

add variables in memory (initialized)
C
int x=17;
int y=25
int z;
int main(void) {
    z = x + y;
}

MIPS .data
.data
    x: .word 17
    y: .word 25
    z: .space 4

MIPS .text
main:
    la $t0, x
    lw $t1, ($t0)
    la $t0, y
    lw $t2, ($t0)
    add $t3, $t1, $t2
    la $t0, z
    sw $t3, 0($t0)  # z = x + y;
    li $v0, 1  # syscall 1: print_int

source code for add_memory_initialized.s
https://www.cse.unsw.edu.au/~cs1521/24T1/ COMP1521 24T1 — MIPS Data 20 / 75

add variables in memory (uninitialized)
C
int x, y, z;
int main(void) {
    x = 17;
    y = 25;
    z = x + y;
}

MIPS (.data)
.data
    x: .space 4
    y: .space 4
    z: .space 4

MIPS (.text)
main:
    li $t0, 17
    la $t1, x
    sw $t0, ($t1)  # x = 17;
    li $t0, 25
    la $t1, y
    sw $t0, ($t1)  # y = 25;
    la $t0, x
    lw $t1, ($t0)
    la $t0, y
    lw $t2, ($t0)
    add $t3, $t1, $t2
    la $t0, z
    sw $t3, 0($t0)  # z = x + y;
    li $v0, 1  # syscall 1: print_int

source code for add_memory.s
https://www.cse.unsw.edu.au/~cs1521/24T1/ COMP1521 24T1 — MIPS Data 21 / 75
add variables in memory (initialized)

C
```
int x=17;
int y=25
int z;
int main(void) {
    z = x + y;
}
```

MIPS
```
main:
    la $t0, x
    lw $t1, ($t0)
    la $t0, y
    lw $t2, ($t0)
    add $t3, $t1, $t2
    la $t0, z
    sw $t3, 0($t0)  # z = x + y;
```

Address of C 1-d Array Elements - Code
```
double array[10];
for (int i = 0; i < 10; i++) {
    printf("array[%d]=%p\n", i, &array[i]);
}
printf("\nExample computation for address of array element\n");
uintptr_t a = (uintptr_t)&array[0];
printf("array[0] + 7 * sizeof (double) = 0x%lx\n", a + 7 * sizeof (double));
printf("array[0] + 7 * %lx = 0x%lx\n", sizeof (double), a + 7 * sizeof (double));
printf("0x%lx + 7 * %lx = 0x%lx\n", a, sizeof (double), a + 7 * sizeof (double));
printf("array[7] = %p\n", &array[7]);
```

- this code uses types covered later in the course
Address of C 1-d Array Elements - Output

```
$ dcc array_element_address.c -o array_element_address
$ ./array_element_address

array[0]=0x7fffdd841d00
array[1]=0x7fffdd841d08
array[2]=0x7fffdd841d10
array[3]=0x7fffdd841d18
array[4]=0x7fffdd841d20
array[5]=0x7fffdd841d28
array[6]=0x7fffdd841d30
array[7]=0x7fffdd841d38
array[8]=0x7fffdd841d40
array[9]=0x7fffdd841d48
```

Example computation for address of array element

```
array[0] + 7 * sizeof (double) = 0x7fffdd841d38
array[0] + 7 * 8 = 0x7fffdd841d38
0x7fffdd841d00 + 7* 8 = 0x7fffdd841d38
```

store value in array element — example 1

C

```c
int x[10];

int main(void) {
    // sizeof x[0] == 4
    x[3] = 17;
}
```

MIPS

```mips
main:
    li $t0, 3

    # each array element is 4 bytes
    mul $t0, $t0, 4
    la $t1, x
    add $t2, $t1, $t0
    li $t3, 17
    sw $t3, 0($t2)

.data
    x: .space 40
```

store value in array element - example 2

C

```c
#include <stdint.h>

int16_t x[30];

int main(void) {
    // sizeof x[0] == 2
    x[13] = 23;
}
```

MIPS

```mips
main:
    li $t0, 13

    # each array element is 2 bytes
    mul $t0, $t0, 2
    la $t1, x
    add $t2, $t1, $t0
    li $t3, 23
    sh $t3, 0($t2)

.data
    x: .space 60
```
C

```c
int main(void) {
    int i = 0;
    while (i < 5) {
        printf("%d
", numbers[i]);
        i++;
    }
    return 0;
}
```

Simplified C

```c
int main(void) {
    int i = 0;
    loop:
        if (i >= 5) goto end;
        printf("%d", numbers[i]);
        printf("\n");
        i++;
        goto loop;
    end:
        return 0;
}
```

Printing Array: MIPS

```
# print array of ints
# i in $t0
main:
    li $t0, 0           # int i = 0;
loop:
    bge $t0, 5, end    # if (i >= 5) goto end;
    la $t1, numbers   # int j = numbers[i];
    mul $t2, $t0, 4
    add $t3, $t2, $t1
    lw $a0, 0($t3)    # printf("%d", j);
    li $v0, 1
    syscall
    li $a0, '\n'     # printf("\n");
    li $v0, 11
    syscall
    addi $t0, $t0, 1  # i++
    b loop            # goto loop
end:
    li $v0, 0         # return 0
    jr $ra
.data
numbers:  # int numbers[10] = { 3, 9, 27, 81, 243};
    .word 3, 9, 27, 81, 243
```

source code for print5.s
Changing an Array: C

```c
int i;
i = 0;
while (i < 5) {
    numbers[i] *= 42;
i++;
}
```

source code for change_array.c

Changing an Array MIPS

```mips
# i in register $t0
# registers $t1..$t3 used to hold calculations
main:
li $t0, 0 # i = 0
loop1:
    bge $t0, 5, end1 # while (i < 5) {
    mul $t1, $t0, 4 #
    la $t2, numbers # calculate &numbers[i]
    add $t1, $t1, $t2 #
    lw $t3, ($t1) # load numbers[i] into $t3
    mul $t3, $t3, 42 # numbers[i] *= 42;
    sw $t3, ($t1) # store scaled number in array
    addi $t0, $t0, 1 # i++;
    b loop1
end1:
```

source code for change_array.s

Reading into an Array: C

```c
int i = 0;
while (i < 10) {
    printf("Enter a number: ");
    scanf("%d", &numbers[i]);
    i++;
}
```

source code for read10.c
Reading into an Array: MIPS

```mips
li $t0, 0  # i = 0
loop0:
  bge $t0, 10, end0  # while (i < 10) {
  la $a0, string0  # printf("Enter a number: ");
  li $v0, 4
  syscall
  li $v0, 5  # scanf("%d", &numbers[i]);
  syscall
  mul $t1, $t0, 4  # calculate &numbers[i]
  la $t2, numbers  #
  add $t3, $t1, $t2  #
  sw $v0, ($t3)  # store entered number in array
  addi $t0, $t0, 1  # i++;
  b loop0  # }
end0:
```

source code for read10.s

Printing in reverse order: C

```c
printf("Reverse order:\n");
count = 9;
while (count >= 0) {
  printf("%d\n", numbers[count]);
  count--;
}
```

source code for reverse10.c

Printing in reverse order: C

```mips
la $a0, string1  # printf("Reverse order:\n");
li $v0, 4
syscall
li $t0, 9  # count = 9;
next:
  blt $t0, 0, end1  # while (count >= 0) {
  mul $t1, $t0, 4  # printf("%d", numbers[count])
  la $t2, numbers  # calculate &numbers[count]
  add $t3, $t1, $t2  #
  lw $a0, ($t3)  # load numbers[count] into $a0
  li $v0, 1
  syscall
  li $a0, '\n'  # printf("%c", '\n');
  li $v0, 11
  syscall
  addi $t0, $t0, -1  # count--;
  b next  # }
end1:
```

source code for reverse10.s
```c
int array[X][Y];
printf("sizeof array[2][3] = %lu\n", sizeof array[2][3]);
printf("sizeof array[1] = %lu\n", sizeof array[1]);
printf("sizeof array = %lu\n", sizeof array);
printf("&array=%p\n", &array);
for (int x = 0; x < X; x++) {
    printf("&array[%d]=%p\n", x, &array[x]);
    for (int y = 0; y < Y; y++) {
        printf("&array[%d][%d]=%p\n", x, y, &array[x][y]);
    }
}
```

• this code uses types covered later in the course

### Computing sum of 2-d Array : C

Assume we have a 2d-array:

```c
int32_t matrix[6][5];
```

We can sum its value like this in C

```c
int row, col, sum = 0;
// row-by-row
for (row = 0; row < 6; row++) {
    // col-by-col within row
    for (col = 0; col < 5; row++) {
        sum += matrix[row][col];
    }
}
```

MIPS directives for an equivalent 2d-array

```mips
.data
matrix: .space 120 # 6 * 5 == 30 array elements each 4 bytes
```
Computing sum of 2-d Array : MIPS

li $t0, 0  # sum = 0
li $t1, 0  # row = 0
loop1: bge $t1, 6, end1  # if (row >= 6) break
li $t2, 0  # col = 0
loop2: bge $t2, 5, end2  # if (col >= 5) break
la $t3, matrix
mul $t4, $t1, 20  # t1 = row*rowsize
mul $t5, $t2, 4  # t2 = col|intsize
add $t6, $t3, $t4  # offset = t0+t1
add $t7, $t6, $t5  # offset = t0+t1
lw $t5, 0($t7)  # t0 = *(matrix+offset)
addi $t2, $t2, 1  # col++
j loop2
add $t1, $t1, 1  # row++
j loop1
end2:
addi $t0, $t0, $t5  # sum += t0
addi $t0, $t0, $t5

Printing 2-d Array: C to simplified C

```c
int main(void) {
    int i = 0;
    loop1:
        if (i >= 3) goto end1;
        int j = 0;
        loop2:
            if (j >= 5) goto end2;
            printf("%d", numbers[i][j]);
            printf("%c", ' ');
            j++;
        goto loop2;
    end2:
        printf("%c", '
');
        i++;
    goto loop1;
end1:
    return 0;
}
```

source code for print2d.simple.c

Printing 2-d Array: MIPS

```mips
# print a 2d array
# i in $t0
# j in $t1
# $t2..$t6 used for calculations
main:
    li $t0, 0  # int i = 0;
loop1:
    bge $t0, 3, end1  # if (i >= 3) goto end1;
    li $t1, 0  # int j = 0;
loop2:
    bge $t1, 5, end2  # if (j >= 5) goto end2;
    la $t2, numbers  # printf("%d", numbers[i][j]);
    mul $t3, $t0, 20
    add $t4, $t3, $t2
    mul $t5, $t1, 4
    add $t6, $t5, $t4
    lw $a0, 0($t6)
    li $v0, 1
    syscall
```

source code for print2d.s
Printing 2-d Array: MIPS (continued)

```assembly
li $a0, ' '    # printf("%c", ' ');
li $v0, 11
syscall
addi $t1, $t1, 1    # j++;
b loop2    # goto loop2;
end2:
li $a0, '\n'    # printf("%c", '\n');
li $v0, 11
syscall
addi $t0, $t0, 1    # i++
b loop1    # goto loop1
end1:
li $v0, 0    # return 0
jr $ra
```

```
.data
# int numbers[5][5] = {{3, 9, 27, 81, 243}, {4, 16, 64, 256, 1024}, {5, 25, 125, 625, 3125}};
numbers:
.word 3, 9, 27, 81, 243, 4, 16, 64, 256, 1024, 5, 25, 125, 625, 3125
```

Printing a Flag: C

```c
#include <stdio.h>
#define N_ROWS 6
#define N_COLS 12
char flag[N_ROWS][N_COLS] = {
    {'#', '#', '#', '#', '#', '.', '.', '#', '#', '#', '#', '#'},
    {'#', '#', '#', '#', '#', '.', '.', '#', '#', '#', '#', '#'},
    {'.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.'},
    {'.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.'},
    {'#', '#', '#', '#', '#', '.', '.', '#', '#', '#', '#', '#'},
    {'#', '#', '#', '#', '#', '.', '.', '#', '#', '#', '#', '#'};

int main(void) {
    for (int row = 0; row < N_ROWS; row++) {
        for (int col = 0; col < N_COLS; col++) {
            printf("%c", flag[row][col]);
        }
        printf("\n");
    }
}
```

Printing a Flag: simplified C

```c
row_loop__init:
int row = 0;
row_loop__cond:
    if (row >= N_ROWS) goto row_loop__end;
row_loop__body:
    int col = 0;
    col_loop__init:
    col_loop__cond:
        if (col >= N_COLS) goto col_loop__end;
    col_loop__body:
        printf("%c", flag[row][col]);
        // &flag[row][col] = flag + offset * sizeof(element)
        // = flag + (row * N_COLS + col) * sizeof(element)
    col_loop__step:
        col++;
        goto col_loop__cond;
    col_loop__end:
    printf("\n");
row_loop__step:
    row++;
    goto row_loop__cond;
row_loop__end:
```
Printing a Flag: MIPS

N_ROWS = 6
N_COLS = 12

main:

    # Locals:
    # - $t0: int row
    # - $t1: int col
    # - $t2: temporary result

main__row_loop_init:
    li $t0, 0
      # int row = 0;

main__row_loop_cond:
    bge$t0, N_ROWS, main__row_loop_end
      # if (row >= N_ROWS) goto main__row_loop_end;

main__row_loop_body:

main__col_loop_init:
    li $t1, 0
      # int col = 0;

main__col_loop_cond:
    bge$t1, N_COLS, main__col_loop_end
      # if (col >= N_COLS) goto main__col_loop_end;

main__col_loop_body:
    li $v0, 11
      # syscall 11: print_char
    lb $a0, flag($t2)
      # printf("%c", flag[row][col]);

main__col_loop_step:
    addi $t1,$t1,1
      # col++;

main__row_loop_step:
    addi $t0,$t0,1
      # row++;

main__row_loop_end:
    li $v0,0
      # return 0;
    jr $ra

.data
flag:
.byte ' ', '#', '#', '#', '#', '#', '.', '.', '#', '#', '#', '#', '#',
.byte ' ', '#', '#', '#', '#', '#', '.', '.', '#', '#', '#', '#', '#',
.byte ' ', '#', '#', '#', '#', '#', '.', '.', '#', '#', '#', '#', '#',
.byte ' ', '#', '#', '#', '#', '#', '.', '.', '#', '#', '#', '#', '#',
.byte ' ', '#', '#', '#', '#', '#', '.', '.', '#', '#', '#', '#', '#',
.byte ' ', '#', '#', '#', '#', '#', '.', '.', '#', '#', '#', '#', '#',
.byte ' ', '#', '#', '#', '#', '#', '.', '.', '#', '#', '#', '#', '#',
.byte ' ', '#', '#', '#', '#', '#', '.', '.', '#', '#', '#', '#', '#',
.byte ' ', '#', '#', '#', '#', '#', '.', '.', '#', '#', '#', '#', '#',
.byte ' ', '#', '#', '#', '#', '#', '.', '.', '#', '#', '#', '#', '#',
.byte ' ', '#', '#', '#', '#', '#', '.', '.', '#', '#', '#', '#', '#',
.byte ' ', '#', '#', '#', '#', '#', '.', '.', '#', '#', '#', '#', '#',

Alignment

- C standard requires simple types of size N bytes to be stored only at addresses which are divisible by N
  - if `int` is 4 bytes, must be stored at address divisible by 4
  - if `double` is 8 bytes, must be stored at address divisible by 8
- compound types (arrays, structs) must be aligned so their components are aligned
- MIPS requires this alignment
- on other architectures aligned access faster
Example C with unaligned accesses

```c
char bytes[32];
int *i = (int *)&bytes[1];
// illegal store - not aligned on a 4-byte boundary
*i = 42;
printf("%d\n", *i);
```

source code for unalign.c

Example MIPS with unaligned accesses

```mips
.data
# data will be aligned on a 4-byte boundary
# most likely on at least a 128-byte boundary
# but safer to just add a .align directive
.align 2
.space 1
v1: .space 1
v2: .space 4
v3: .space 2
v4: .space 4
.space 1
.align 2 # ensure e is on a 4 (2**2) byte boundary
v5: .space 4
.space 1
v6: .word 0 # word directive aligns on 4 byte boundary
```

source code for unalign.s

Example MIPS with unaligned accesses

```mips
li $t0, 1
sb $t0, v1 # will succeed because no alignment needed
sh $t0, v1 # will fail because v1 is not 2-byte aligned
sw $t0, v1 # will fail because v1 is not 4-byte aligned
sh $t0, v2 # will succeed because v2 is 2-byte aligned
sw $t0, v2 # will fail because v2 is not 4-byte aligned
sh $t0, v3 # will succeed because v3 is 2-byte aligned
sw $t0, v3 # will fail because v3 is not 4-byte aligned
sh $t0, v4 # will succeed because v4 is 2-byte aligned
sw $t0, v4 # will succeed because v4 is 4-byte aligned
sw $t0, v5 # will succeed because v5 is 4-byte aligned
sw $t0, v6 # will succeed because v6 is 4-byte aligned
li $v0, 0 # return
jr $ra # return
```

source code for unalign.s
Implementing Structs in MIPS

C `struct` definitions effectively define a new type.

```c
// new type called "struct student"
struct student {...};

// new type called student_t
typedef struct student student_t;
```

Instances of structures can be created by allocating space:

```c
stu1:
    # sizeof(Student) == 56
    .space 56

stu2:
    # student_t stu2;
    .space 56

stu:
    # student_t *stu;
    .space 4
```

Accessing structure components is by offset, not name.

```assembly
li $t0, 5012345
la $t1, stu1
sw $t0, 0($t1) # stu1.id = 5012345;
li $t0, 3778
sw $t0, 44($t1) # stu1.program = 3778;

la $t2, stu2
li $t0, 3707
sw $t0, 44($t2) # stu->program = 3707;
li $t0, 5034567
sw $t0, 0($t2) # stu->id = 5034567;
```
# Source Code for student.c

```c
struct details {
    uint16_t postcode;
    uint8_t wam;
    uint32_t zid;
};

struct details student;

int main(void) {
    student.postcode = 2052;
    student.wam = 95;
    student.zid = 5123456;
    printf("%d", student.zid);
    putchar(' ');  
    printf("%d", student.wam);
    putchar(' ');  
    printf("%d", student.postcode);
    putchar(score());

    return 0;
}
```

# Source Code for student.s

```assembly
# Save values into struct 
la $t0, student  # student.postcode = 2052;
addi $t1, $t0, OFFSET_POSTCODE
li $t2, 2052
sh $t2, ($t1)
la $t0, student  # student.wam = 95;
addi $t1, $t0, OFFSET_WAM
li $t2, 95
sb $t2, ($t1)
la $t0, student  # student.zid = 5123456
addi $t1, $t0, OFFSET_ZID
li $t2, 5123456
sw $t2, ($t1)
```
More complex student info: C

```c
#include <stdio.h>

struct student {
    int zid;
    char first[20];
    char last[20];
    int program;
    char alias[10];
};

struct student abiram = {
    .zid = 5308310,
    .first = "Abiram",
    .last = "Nadarajah",
    .program = 3778,
    .alias = "abiram"
};

struct student xavier = {
    .zid = 5417087,
    .first = "Xavier",
    .last = "Cooney",
    .program = 3778,
    .alias = "xavc"
};

int main(void) {
    struct student *selection = &abiram;
    printf("zID: z%d
", selection->zid);
    printf("First name: %s
", selection->first);
    printf("Last name: %s
", selection->last);
    printf("Program: %d
", selection->program);
    printf("Alias: %s
", selection->alias);
    // What's the size of each field of this struct,
    // as well as the overall struct?
    printf("sizeof(zid) = %zu
", sizeof(selection->zid));
    printf("sizeof(first) = %zu
", sizeof(selection->first));
    printf("sizeof(last) = %zu
", sizeof(selection->last));
    printf("sizeof(program) = %zu
", sizeof(selection->program));
    printf("sizeof(alias) = %zu
", sizeof(selection->alias));
    // What's the size of the overall struct?
    printf("sizeof(struct student) = %zu
", sizeof(struct student));
    // We can see that two extra padding bytes were added to the end
    // of the struct, to ensure that the next struct in memory is aligned
    // to a word boundary.
    return 0;
}
```
# A demo of accessing fields of structs in MIPS.

# Offsets for fields in `struct student`

STUDENT_OFFSET_ZID = 0

STUDENT_OFFSET_FIRST = 4

STUDENT_OFFSET_LAST = 20 + STUDENT_OFFSET_FIRST

STUDENT_OFFSET_PROGRAM = 20 + STUDENT_OFFSET_LAST

STUDENT_OFFSET_ALIAS = 4 + STUDENT_OFFSET_PROGRAM

# sizeof the struct - note that there are 2 padding bytes at the end of the struct.

SIZEOF_STRUCT_STUDENT = 10 + STUDENT_OFFSET_ALIAS + 2

.text

main:

source code for struct.s

https://www.cse.unsw.edu.au/~cs1521/24T1/ COMP1521 24T1 — MIPS Data

Array of Structs: C

// simple example of accessing struct within array within struct

#include <stdio.h>

#define MAX_POLYGON 6

struct point {
    int x;
    int y;
};

struct polygon {
    int degree;
    struct point vertices[MAX_POLYGON]; // C also allows variable size array here
};

void print_last_vertex(struct polygon *p);

struct polygon triangle = {3, {{0,0}, {3,0}, {0,4}}};
int main(void) {
    print_last_vertex(&triangle); // prints 0,4
    return 0;
}

Array of Structs: MIPS

```c
# simple example of accessing struct within array within struct
# struct point {
#    int x;
#    int y;
#};
#
# struct polygon {
#    int degree;
#    struct point vertices[6];
#};

OFFSET_POINT_X = 0
OFFSET_POINT_Y = 4
SIZEOF_POINT = 8
OFFSET_POLYGON_DEGREE = 0
OFFSET_POLYGON_VERTICES = 4
SIZEOF_POLYGON = 52

main:
```

Array of Structs: MIPS

```asm
push $ra
la $a0, triangle
jal print_last_vertex # print_last_vertex(&triangle);
li $v0, 0
pop $ra
jr $ra
```

```
print_last_vertex:
    # $a0: p
    # $t0: n
    # $t1: last
    # $t2..$t5: temporaries
    lw $t2, OFFSET_POLYGON_DEGREE($a0) # int n = p->degree - 1;
    addi $t0, $t2, -1
    addi $t3, $a0, OFFSET_POLYGON_VERTICES # calculate &(p->vertices[n])
    mul $t4, $t0, SIZEOF_POINT
    add $t1, $t3, $t4
    lw $a0, OFFSET_POINT_X($t1) # printf("%d", last->x);
    li $v0, 1
    syscall
    li $a0, ','
    li $v0, 11
    syscall
    lw $a0, OFFSET_POINT_Y($t1) # printf("%d", last->y);
    li $v0, 1
    syscall
    li $a0, '
    li $v0, 11
    syscall
    jr $ra
```
### Array of Structs: MIPS

```
.data
# struct polygon triangle = {3, {{0,0}, {3,0}, {0,4}}};
triangle:
   .word 3
   .word 0,0, 3,0, 0,4, 0,0, 0,0, 0,0
```

### Implementing Pointers in MIPS

C

```
int i;
int *p;
p = &answer;
i = *p;
// prints 42
printf("%d
", i);
*p = 27;
// prints 27
printf("%d
", answer);
```

MIPS

```
la $t0, answer # p = &answer;
lw $t1, ($t0) # i = *p;
move $a0, $t1 # printf("%d\n", i);
li $v0, 1
syscall
li $a0, '\n' # printf("%c", '\n');
li $v0, 11
syscall
li $t2, 27 # *p = 27;
sw $t2, ($t0) #
lw $a0, answer # printf("%d\n", answer);
li $v0, 1
syscall
li $a0, '\n' # printf("%c", '\n');
li $v0, 11
syscall
```

### Example - Accessing Struct within Array within Struct (main)

```
// simple example of accessing struct within array within struct
#include <stdio.h>
#define MAX_POLYGON 6
struct point {
   int x;
   int y;
};
struct polygon {
   int degree;
   struct point vertices[MAX_POLYGON]; // C also allows variable size array here
};
void print_last_vertex(struct polygon *p);
struct polygon triangle = {3, {{0,0}, {3,0}, {0,4}}};
```

```
Example - Accessing Struct within Array within Struct (main)

```c
int main(void) {
    print_last_vertex(&triangle); // prints 0,4
    return 0;
}
```

Example - Accessing Struct within Array within Struct (C)

```c
void print_last_vertex(struct polygon *p) {
    printf("%d", p->vertices[p->degree - 1].x);
    putchar(',');
    printf("%d", p->vertices[p->degree - 1].y);
    putchar('
');
}
```

Example - Accessing Struct within Array within Struct (MIPS)

```mips
void print_last_vertex(struct polygon *p) {
    int n = p->degree - 1;
    struct point *last = &(p->vertices[n]);
    printf("%d", last->x);
    putchar(',');
    printf("%d", last->y);
    putchar('
');
}
```
C

```c
int main(void) {
    int *p = &numbers[0];
    int *q = &numbers[4];
    while (p <= q) {
        printf("%d\n", *p);
        p++;
    }
    return 0;
}
```

Simplified C

```c
int main(void) {
    int *p = &numbers[0];
    int *q = &numbers[4];
    loop:
        if (p > q) goto end;
        int j = *p;
        printf("%d", j);
        printf("%c", '\n');
        p++;
        goto loop;
    end:
        return 0;
}
```

Printing Array with Pointers: MIPS

```mips
main:
    la $t0, numbers # int *p = &numbers[0];
    la $t0, numbers # int *q = &numbers[4];
    addi $t1, $t0, 16 #

loop:
    bgt $t0, $t1, end # if (p > q) goto end;
    lw $a0, 0($t0) # int j = *p;
    li $v0, 1
    syscall
    li $a0, '\n' # printf("%c", '\n');
    li $v0, 11
    syscall
    addi $t0, $t0, 4 # p++
    b loop # goto loop

end:
```

Printing Array with Pointers: MIPS - faster

```mips
main:
    la $t0, numbers # int *p = &numbers[0];
    addi $t1, $t0, 16 # int *q = &numbers[4];

loop:
    lw $a0, ($t0) # printf("%d", *p);
    li $v0, 1
    syscall
    li $a0, '\n' # printf("%c", '\n');
    li $v0, 11
    syscall
    addi $t0, $t0, 4 # p++
    ble $t0, $t1, loop # if (p <= q) goto loop;
```

https://www.cse.unsw.edu.au/~cs1521/24T1/COMP1521-24T1 — MIPS Data