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 anyt 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.
- 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
• addresses are 32 bits
• only load/store instructions access memory on the MIPS
• 1 byte (8-bit) loaded/stored with \textbf{lb/sb}
• 2 bytes (16-bit) called a \textbf{half-word}, loaded/stored with \textbf{lh/sh}
• 4 bytes (32-bits) called a \textbf{word}, loaded/stored with \textbf{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 \textbf{sb} & \textbf{sh} operations low (least significant) bits of source register are used.
• \textbf{lb/lh} assume byte/halfword contains a 8-bit/16-bit \textbf{signed} integer
  • high 24/16-bits of destination register set to 1 if 8-bit/16-bit integer negative
• unsigned equivalents \textbf{lbu} & \textbf{lhu} assume integer is \textbf{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</code> $r_t$, $I(r_s)$</td>
<td>$r_t = \text{mem}[r_s + I]$</td>
<td>100000ssssstttttIIIIIIIIIIIIIIII</td>
</tr>
<tr>
<td><code>lh</code> $r_t$, $I(r_s)$</td>
<td>$r_t = \text{mem}[r_s + I]$</td>
<td>100001ssssstttttIIIIIIIIIIIIIIII</td>
</tr>
<tr>
<td></td>
<td>$\text{mem}[r_s + I+1] \ll 8$</td>
<td></td>
</tr>
<tr>
<td><code>lw</code> $r_t$, $I(r_s)$</td>
<td>$r_t = \text{mem}[r_s + I]$</td>
<td>100011ssssstttttIIIIIIIIIIIIIIII</td>
</tr>
<tr>
<td></td>
<td>$\text{mem}[r_s + I+1] \ll 8$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\text{mem}[r_s + I+2] \ll 16$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\text{mem}[r_s + I+3] \ll 24$</td>
<td></td>
</tr>
<tr>
<td><code>sb</code> $r_t$, $I(r_s)$</td>
<td>$\text{mem}[r_s + I] = r_t &amp; 0xff$</td>
<td>101000ssssstttttIIIIIIIIIIIIIIIIII</td>
</tr>
<tr>
<td><code>sh</code> $r_t$, $I(r_s)$</td>
<td>$\text{mem}[r_s + I] = r_t &amp; 0xff$</td>
<td>101001ssssstttttIIIIIIIIIIIIIIIIII</td>
</tr>
<tr>
<td></td>
<td>$\text{mem}[r_s + I+1] = r_t \gg 8 &amp; 0xff$</td>
<td></td>
</tr>
<tr>
<td><code>sw</code> $r_t$, $I(r_s)$</td>
<td>$\text{mem}[r_s + I] = r_t &amp; 0xff$</td>
<td>101011ssssstttttIIIIIIIIIIIIIIIIII</td>
</tr>
<tr>
<td></td>
<td>$\text{mem}[r_s + I+1] = r_t \gg 8 &amp; 0xff$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\text{mem}[r_s + I+2] = r_t \gg 16 &amp; 0xff$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\text{mem}[r_s + I+3] = r_t \gg 24 &amp; 0xff$</td>
<td></td>
</tr>
</tbody>
</table>
Code example: storing and loading a value (no labels)

# 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

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 '\n'
    li  $v0, 11         # return 0
    syscall

    jr  $ra

source code for load_store_no_label.s

https://www.cse.unsw.edu.au/~cs1521/24T2/
mipsy has directives to initialise memory, and to associate labels with addresses.

```plaintext
.text  # following instructions placed in text segment
.data  # following objects placed in data segment

a: .space 18  # int8_t a[18];
    .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'};
```
# 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

```
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 '\n'
    li  $v0, 11      # print '
'
    syscall
    li  $v0, 0       # return 0
    jr  $ra

.data

.answer:

    .space 1  # set aside 1 byte and associate label answer with its address
```
# 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 now contain 42
    syscall
    li $a0, '\n'    # print '\n'
    li $v0, 11      # return 0
    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.s
https://www.cse.unsw.edu.au/~cs1521/24T2/
• Note the **la** (load address) instruction is normally used to set a register to a labelled memory address.

```assembly
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.

```plaintext
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

```plaintext
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
<table>
<thead>
<tr>
<th>Region</th>
<th>Address</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>.text</td>
<td>0x00400000</td>
<td>instructions only; read-only; cannot expand</td>
</tr>
<tr>
<td>.data</td>
<td>0x10000000</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>
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
Labels and **directives** used to allocate space for global variables in the `.data` segment.

```assembly
.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
https://www.cse.unsw.edu.au/~cs1521/24T2/COMP1521 24T2 — MIPS Data
Incrementing a Global Variable: C

```c
#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

https://www.cse.unsw.edu.au/~cs1521/24T2/
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 Increments

https://www.cse.unsw.edu.au/~cs1521/24T2/
int main(void) {
    int x, y, z;
    x = 17;
    y = 25;
    z = x + y;
    // ...
}

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

add: local variables in registers
add variables in memory (uninitialized)

C

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

MIPS (.data)

```mips
.data
x:
.space 4
y:
.space 4
z:
.space 4
```

MIPS (.text)

```mips
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/~cs1521/24T2/
C

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

MIPS .data

```mips
.data
x:
    .word 17
y:
    .word 25
z:
    .space 4
```

MIPS main:

```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;
```

source code for add_memory_initialized.s

https://www.cse.unsw.edu.au/~cs1521/24T2/COMPE52172 — MIPS Data
```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
```
C
int x=17;
int y=25
int z;
int main(void) {
    z = x + y;
}
add variables in memory (array)

C

```c
int x[] = {17, 25, 0};
int main(void) {
    x[2] = x[0] + x[1];
}
```

MIPS .text

```mips
main:
    la $t0, x
    lw $t1, 0($t0)
    lw $t2, 4($t0)
    add $t3, $t1, $t2  # x[2] = x[0] + x[1];
    sw $t3, 8($t0)
    li $v0, 1    # syscall 1: print_int
    lw $a0, 8($t0)  #
    syscall       # printf("%d", x[2]);
    li $v0, 11   # syscall 11: print_char
    li $a0, '\n'  #
    syscall       # putchar('\n');
    li $v0, 0    # return 0;
    jr $ra        # return 0;
.data
x: .word 17, 25, 0    # int x[] = {17, 25, 0}
```

https://www.cse.unsw.edu.au/~cs1521/24T2/COMP1521-24T2-MIPS-DATA
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 
", sizeof (double), a + 7 * sizeof (double));
printf("0x%lx + 7 * %lx = 0x%lx 
", a, sizeof (double), a + 7 * sizeof (double));
printf("&array[7] = %p \n", &array[7]);
Address of C 1-d Array Elements - Output

$ gcc 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
&array[7] = 0x7fffdd841d38
int x[10];

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

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
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\n", 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("%c", '\n');
        i++;
        goto loop;
    end:
        return 0;
}
```
# 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("%c", '\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

https://www.cse.unsw.edu.au/~cs1521/24T2/
int i;
i = 0;
while (i < 5) {
    numbers[i] *= 42;
i++;
}

source code for change_array.c
# 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:
```c
int i = 0;
while (i < 10) {
    printf("Enter a number: ");
    scanf("%d", &numbers[i]);
    i++;
}
```
Reading into an Array: MIPS

```asm
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

https://www.cse.unsw.edu.au/~cs1521/24T2/COMP1521-24T2-MIPS-data
```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

```assembly
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 $t1, $t1, $t2 #
    lw $a0, ($t1)     # 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

https://www.cse.unsw.edu.au/~cs1521/24T2/
```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
$ gcc 2d_array_element_address.c -o 2d_array_element_address
$ ./2d_array_element_address

sizeof array[2][3] = 4
sizeof array[1] = 16
sizeof array = 48

&array=0x7ffd93bb16c0
&array[0]=0x7ffd93bb16c0
&array[0][0]=0x7ffd93bb16c0
&array[0][1]=0x7ffd93bb16c4
&array[0][2]=0x7ffd93bb16c8
&array[0][3]=0x7ffd93bb16cc
&array[1]=0x7ffd93bb16d0
&array[1][0]=0x7ffd93bb16d0
&array[1][1]=0x7ffd93bb16d4
&array[1][2]=0x7ffd93bb16d8
&array[1][3]=0x7ffd93bb16dc
&array[2]=0x7ffd93bb16e0
&array[2][0]=0x7ffd93bb16e0
&array[2][1]=0x7ffd93bb16e4
&array[2][2]=0x7ffd93bb16e8
&array[2][3]=0x7ffd93bb16ec
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)
add $t0, $t0, $t5 # sum += t0
addi $t2, $t2, 1 # col++
j loop2

end2: addi $t1, $t1, 1 # row++
j loop1

end1:
C

```c
int main(void) {
    int i = 0;
    while (i < 3) {
        int j = 0;
        while (j < 5) {
            printf("%d", numbers[i][j]);
            printf("%c", ' ");
            j++;
        }
        printf("%c", '\n' );
        i++;
    }
    return 0;
}
```

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", '\n' );
            i++;
            goto loop1;
    end1:
    return 0;
}
```
# 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

https://www.cse.unsw.edu.au/~cs1521/24T2/COMP1521 24T2 — MIPS Data
li $a0, ' ' # printf("%c", ' ');
li $v0, 11
syscall
addi $t1, $t1, 1 # j++;
b  loop2  # goto loop2;
end2:
li $a0, '
' # printf("%c", '
');
li $v0, 11
syscall
addi $t0, $t0, 1 # i++
b loop1  # goto loop1
end1:
li $v0, 0  # return 0
jr $ra
.data
# int numbers[3][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

source code for print2d.s

https://www.cse.unsw.edu.au/~cs1521/24T2/COMP1521 — MIPS Data

43 / 75
Printing a Flag: C

// Print a 2D array of characters.
#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");
    }
}

https://www.cse.unsw.edu.au/~cs1521/24T2/COMP1521 24T2 — MIPS Data 44 / 75
Printing a Flag: simplified C

row_loop__init:
    int row = 0;
row_loop__cond:
    if (row >= N_ROWS) goto row_loop__end;
row_loop__body:
col_loop__init:
    int col = 0;
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:
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
Printing a Flag: MIPS

```mips
mul $t2, $t0, N_COLS # (row * N_COLS
add $t2, $t2, $t1 # + col)
lb $a0, flag($t2) #
syscall # printf("%c", flag[row][col]);

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

main__col_loop_end:
  li $v0, 11 # syscall 11: print_char
  li $a0, '\n' #
syscall # putchar('\n');

main__row_loop_step:
  addi $t0, $t0, 1 # i++;
  j main__row_loop_cond

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

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

source code for flag.s

https://www.cse.unsw.edu.au/~cs1521/24T2/
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
.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 2
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

https://www.cse.unsw.edu.au/~cs1521/24T2/
Example MIPS with unaligned accesses

```
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
jr $ra    # return
```

source code for unalign.s

https://www.cse.unsw.edu.au/~cs1521/24T2/
```c
struct _student {
    int    id;
    char   family[20];
    char   given[20];
    int    program;
    double wam;
};
```
C `struct` definitions effectively define a new type.

// 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
# sizeof(Student) == 56
stu1:
    # student_t stu1;
    .space 56

stu2:
    # student_t stu2;
    .space 56

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

https://www.cse.unsw.edu.au/~cs1521/24T2/
Implementing Structs in MIPS

Accessing structure components is by offset, not name

```
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 # stu = &stu2;
li $t0, 3707
sw $t0, 44($t2) # stu->program = 3707;
li $t0, 5034567
sw $t0, 0($t2) # stu->id = 5034567;
```
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('
');
    return 0;
}
# access fields of a simple struct
# struct details {
#   uint16_t postcode; // Size = 2 bytes, Offset = 0 bytes
#   uint8_t wam; // Size = 1 byte, Offset = 2 bytes
#   // Hidden 1 byte of "padding"
#   // Because the Offset of each field must be a multiple of the Size of that field
#   uint32_t zid; // Size = 4 bytes, Offset = 4 bytes
# }; // Total Size = 8
# // The Total Size must be a multiple of the Size of the largest field in the struct
# // More padding will be added to the end of the struct to make this true
# // (not needed in this example)

# offset in bytes of fields of struct details
OFFSET_POSTCODE = 0
OFFSET_WAM = 2
OFFSET_ZID = 4 # unused padding byte before zid field to ensure it is on a 4-byte boundary
### Save values into struct ###

```mips
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)
```
### Load values from struct ###

```mips
la $t0, student  # printf("%d", student.zid);
addi $t1, $t0, OFFSET_ZID
lw $a0, ($t1)
li $v0, 1
syscall
li $a0, ' '  # putchar(' ');
li $v0, 11
syscall
la $t0, student  # printf("%d", student.wam);
addi $t1, $t0, OFFSET_WAM
lb $a0, ($t1)
li $v0, 1
syscall
li $a0, ' '  # putchar(' ');
li $v0, 11
syscall
la $t0, student  # printf("%d", student.postcode);
addi $t1, $t0, OFFSET_POSTCODE
lh $a0, ($t1)
li $v0, 1
syscall
li $a0, '\n'  # putchar('\n');
li $v0, 11
syscall
li $v0, 0  # return 0
jr $ra
```

.source code for student.s

```
student:  # struct details student;
  .space 8  # 1 unused padding byte included to ensure zid field aligned on 4-byte boundary
```

https://www.cse.unsw.edu.au/~cs1521/24T2/
// An example program making use of structs.

#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 = "abiramn"
};

struct student xavier = {
  .zid = 5417087,
  .first = "Xavier",
  .last = "Cooney",
  .program = 3778,
  .alias = "xavc"
};
int main(void) {
    struct student *selection = &abiram;
    printf("zID: z%d\n", selection->zid);
    printf("First name: %s\n", selection->first);
    printf("Last name: %s\n", selection->last);
    printf("Program: %d\n", selection->program);
    printf("Alias: %s\n", selection->alias);
    // What's the size of each field of this struct,
    // as well as the overall struct?
    printf("sizeof(zid) = %zu\n", sizeof(selection->zid));
    printf("sizeof(first) = %zu\n", sizeof(selection->first));
    printf("sizeof(last) = %zu\n", sizeof(selection->last));
    printf("sizeof(program) = %zu\n", sizeof(selection->program));
    printf("sizeof(alias) = %zu\n", sizeof(selection->alias));
    // What's the size of the overall struct?
    printf("sizeof(struct student) = %zu\n", 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;
}

source code for struct.c
https://www.cse.unsw.edu.au/~cs1521/24T2/
# 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 structs
More complex student info: MIPS

```mips
# Locals:
# - $t0: struct student *selection

la $t0, xavier
li $v0, 4
    syscall 4: print_string
la $v0, selection
    syscall 4: print_string
la $t0, xavier
    syscall 4: print_string
li $a0, zid_msg
    syscall 1: print_int
li $a0, selection->zid
    syscall 4: print_string
li $a0, (\n)
    syscall 4: print_string
li $a0, first_name_msg
    syscall 4: print_string
li $a0, (\n)
    syscall 4: print_string
la $a0, first_name
    syscall 4: print_string
li $a0, (\n)
    syscall 4: print_string
la $a0, last_name_msg
    syscall 4: print_string
li $a0, (\n)
    syscall 4: print_string
la $a0, last_name
    syscall 4: print_string
li $a0, (\n)
    syscall 4: print_string
la $a0, program_msg
    syscall 4: print_string
li $a0, (\n)
    syscall 4: print_string
la $a0, program
    syscall 4: print_string
li $a0, (\n)
    syscall 4: print_string
la $a0, alias_msg
    syscall 4: print_string
li $a0, (\n)
    syscall 4: print_string
la $a0, alias
    syscall 4: print_string
li $a0, (\n)
    syscall 4: print_string
la $a0, student_offset_alias($t0)
    syscall 4: print_string
li $a0, (\n)
    syscall 4: print_string
la $a0, student_offset_program($t0)
    syscall 4: print_string
li $a0, (\n)
    syscall 4: print_string
la $a0, (\n)
    syscall 4: print_string
li $a0, (\n)
    syscall 4: print_string
jr $t2
    return 0;
```

https://www.cse.unsw.edu.au/~cs1521/24T2/
// 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;
}

source code for struct_array.c

https://www.cse.unsw.edu.au/~cs1521/24T2/COMP1521-24T2-MIPS-data
# simple example of accessing struct within array within struct

```c
# 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

```mips
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, ','  # putchar(',');
li $v0, 11
syscall
lw $a0, OFFSET_POINT_Y($t1)  # printf("%d", last->y);
li $v0, 1
syscall
li $a0, '
'  # putchar('\n');
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

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

#### MIPS

```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("\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("\n");
    li $v0, 11
    syscall
```

[Source code for pointer.c](https://www.cse.unsw.edu.au/~cs1521/24T2/)
// 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;
}

source code for struct_array.c

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

source code for struct_array.s

https://www.cse.unsw.edu.au/~cs1521/24T2/COMP1521-24T2-MIPS-data
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('
');
}

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('
');
}
Example - Accessing Struct within Array within Struct (MIPS)

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, ','                         # putchar(',');
li   $v0, 11
syscall
lw   $a0, OFFSET_POINT_Y($t1)        # printf("%d", last->y);
li   $v0, 1
syscall
li   $a0, '
'                          # putchar('\n');
li   $v0, 11
syscall
jr   $ra
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

# p in $t0, q in $t1

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:

source code for pointer5.s

https://www.cse.unsw.edu.au/~cs1521/24T2/
# this is closer to the code a compiler might produce
# p in $t0
# q in $t1

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, '
'  # printf("\n", '\n');
    li $v0, 11
    syscall
    addi $t0, $t0, 4  # p++
    ble $t0, $t1, loop  # if (p <= q) goto loop;

source code for pointer5.faster.s
https://www.cse.unsw.edu.au/~cs1521/24T2/