Computer Systems Fundamentals

load_store_no_label.s

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
    syscall

    li   $v0, 0       # return 0
    jr   $ra
```

load_store_label.s

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
    syscall

    li   $v0, 0           #  return 0
    jr   $ra

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

load_store.s

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

sample_data.s

An example data segment - load the program into mipsy_web and check it out

```
        .text
main:

        li      $t0, 42                 #
        sw      $t0, g                  # g = 42;

        li      $v0, 1                  # syscall 1: print_int
        la      $t0, g                  #
        lw      $a0, ($t0)              #
        syscall                 # printf("%d", g);

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

        .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;
```

global_increment.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');
}
```

global_increment.s

Increment a global variable.

```
        .text
main:
        # Locals:
        # - $t0: int *global_counter

        # Method 1: Implicitly load from the
        # address of global_counter.
        # mipsy will automatically load the address
        # into a register behind the scenes by
        # generating multiple real instructions.
        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;
```

add_memory.c

```
#include <stdio.h>

int x, y, z;

int main(void) {
    x = 17;
    y = 25;
    z = x + y;
    printf("%d", z);
    printf("\n");
    return 0;
}
```

add_memory.s

Add 17 and 25 using variables stored in memory and print result.

```
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
        la      $t0, z          #
        lw      $a0, 0($t0)     #
        syscall                 # printf("%d", z);

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

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

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

add_memory_initialized.s

Add 17 and 25 using variables stored in memory and print result.

```
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
        la      $t0, z          #
        lw      $a0, 0($t0)     #
        syscall                 # printf("%d", z);

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

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

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

add_memory_array.c

```
#include <stdio.h>

int x[] = {17, 25, 0};
int main(void) {
    x[2] = x[0] + x[1];
    printf("%d", x[2]);
    printf("\n");
    return 0;
}
```

add_memory_array.s

Add 17 and 25 using variables stored in an array, and print the result.

```
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
        jr      $ra             # return 0;

        .data

x:      .word 17, 25, 0         # int x[] = {17, 25, 0}
```

store_array_element.c

Simple example of accessing an array element

```
#include <stdio.h>

int array[10];

int main(void) {
    array[3] = 17;
}
```

store_array_element.s

```
main:
        li      $t0, 3                  # (3 *
        mul     $t0, $t0, 4             #   sizeof(int)
        la      $t1, x                  #
        add     $t2, $t1, $t0           #   + x) = &x[3]

        li      $t3, 17
        sw      $t3, ($t2)              # x[3] = 17;

        .data
x:      .space  4*10                    # int x[10];
```

array_element_address.c

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>


int main(void) {
    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]);
}
```

print5.c

print array of ints

```
#include <stdio.h>

int numbers[5] = { 3, 9, 27, 81, 243};

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

print5.simple.c

print array of ints

```
#include <stdio.h>

int numbers[5] = { 3, 9, 27, 81, 243};

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;
}
```

print5.s

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
```

change_array.c

```
#include <stdio.h>

int numbers[5] = {3,1,4,1,5};

int main() {
    int i;
    i = 0;
    while (i < 5) {
        numbers[i] *= 42;
        i++;
    }
    i = 0;
    while (i < 5) {
        printf("%d\n", numbers[i]);
        i++;
    }
    return 0;
}
```

change_array.s

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:
    li   $t0, 0

loop2:
    bge  $t0, 5, done   # while (i < 5) {

    mul  $t1, $t0, 4    # printf("%d", numbers[i])
    la   $t2, numbers   # calculate &numbers[i]
    add  $t1, $t1, $t2  #
    lw   $a0, ($t1)     # load numbers[i] into $a0
    li   $v0, 1
    syscall

    li   $a0, '\n'      # printf("%c", '\n');
    li   $v0, 11
    syscall

    addi $t0, $t0, 1    # i++
    b    loop2

done:
    li    $v0, 0        # return 0
    jr    $ra

.data

numbers:
    .word 3,1,4,1,5           # int numbers[5] = {3,1,4,1,5};

string0:
    .asciiz "Enter a number: "
```

read10.c

read 10 numbers into an array then print the 10 numbers

```
#include <stdio.h>

int numbers[10] = { 0 };

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

read10.s

read 10 numbers into an array then print the 10 numbers

i in register $t0 registers $t1, $t2 & $t3 used to hold temporary results

```
main:

    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:

    li   $t0, 0          # i = 0
loop1:
    bge  $t0, 10, end1   # while (i < 10) {

    mul  $t1, $t0, 4     #   calculate &numbers[i]
    la   $t2, numbers    #
    add  $t3, $t1, $t2   #
    lw   $a0, ($t3)      #   load numbers[i] into $a0
    li   $v0, 1          #   printf("%d", numbers[i])
    syscall

    li   $a0, '\n'       #   printf("%c", '\n');
    li   $v0, 11
    syscall

    addi $t0, $t0, 1     #   i++
    b    loop1           # }
end1:

    li   $v0, 0          # return 0
    jr   $ra

.data

numbers:                # int numbers[10];
     .word 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

string0:
    .asciiz "Enter a number: "
```

reverse10.c

read 10 integers then print them in reverse order

```c
#include <stdio.h>

int numbers[10];

int main() {
    int count;

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

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

    return 0;
}
```

reverse10.s

read 10 integers then print them in reverse order

count in register $t0 registers $t1 and $t2 used to hold temporary results

```
main:
    li   $t0, 0           # count = 0

read:
    bge  $t0, 10, print   # while (count < 10) {
    la   $a0, string0     # printf("Enter a number: ");
    li   $v0, 4
    syscall

    li   $v0, 5           #   scanf("%d", &numbers[count]);
    syscall               #
    mul  $t1, $t0, 4      #   calculate &numbers[count]
    la   $t2, numbers     #
    add  $t1, $t1, $t2    #
    sw   $v0, ($t1)       #   store entered number in array

    addi $t0, $t0, 1      #   count++;
    b    read             # }

print:
    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:

    li   $v0, 0           # return 0
    jr   $ra

.data

numbers:                 # int numbers[10];
     .word 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

string0:
    .asciiz "Enter a number: "
string1:
    .asciiz "Reverse order:\n"
```

2d_array_element_address.c

```c
#include <stdio.h>

#define X 3
#define Y 4

int main(void) {
    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]);
        }
    }
}
```

print2d.c

print a 2d array

```c
#include <stdio.h>

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

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;
}
```

print2d.simple.c

print a 2d array

```c
#include <stdio.h>

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

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;
}
```

print2d.s

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
    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[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
```

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");
    }
}
```

flag.simple.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) {
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:
    return 0;
}
```

flag.s

```
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

        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 '#', '#', '#', '#', '#', '.', '.', '#', '#', '#', '#', '#'
```

unalign.c

```c
#include <stdio.h>
#include <stdint.h>

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

unalign.s

```
main:
    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 succeeed because v2 is 2-byte aligned
    sw   $t0, v2  # will fail because v2 is not 4-byte aligned

    sh   $t0, v3  # will succeeed because v3 is 2-byte aligned
    sw   $t0, v3  # will fail because v3 is not 4-byte aligned

    sh   $t0, v4  # will succeeed because v4 is 2-byte aligned
    sw   $t0, v4  # will succeeed because v4 is 4-byte aligned

    sw   $t0, v5  # will succeeed because v5 is 4-byte aligned

    sw   $t0, v6  # will succeeed because v6 is 4-byte aligned

    li   $v0, 0
    jr   $ra   # return

    .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
```

student.c

access fields of a simple struct

```
#include <stdio.h>
#include <stdint.h>

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('\n');
    return 0;
}
```

student.unpadded.s

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

offset in bytes of fields of struct details

```
OFFSET_POSTCODE   = 0
OFFSET_WAM = 2
OFFSET_ZID        = 3

main:

    ### 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)

    ### Load values from struct ###

    la   $t0, student           # printf("%d", student.zid);
    add  $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 $a0, $t0, OFFSET_WAM
    li   $v0, 1
    syscall

    li   $a0, ' '               #   putchar(' ');
    li   $v0, 11
    syscall

    la   $t0, student           # printf("%d", student.postcode);
    addi $t1, $t0, OFFSET_POSTCODE
    lhu  $a0, ($t1)
    li   $v0, 1
    syscall

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

    li   $v0, 0                 # return 0
    jr   $ra

.data

student:                 # struct details student;
     .space 7
```

student.s

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" // Becase 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

main:

    ### 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)

    ### Load values from struct ###

    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
    lbu  $a0, ($t1)
    li   $v0, 1
    syscall

    li   $a0, ' '               #   putchar(' ');
    li   $v0, 11
    syscall

    la   $t0, student           # printf("%d", student.postcode);
    addi $t1, $t0, OFFSET_POSTCODE
    lhu  $a0, ($t1)
    li   $v0, 1
    syscall

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

    li   $v0, 0                 # return 0
    jr   $ra

.data

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

struct.c

An example program making use of structs.

```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 = "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;

}
```

struct.s

A demo of accessing fields of structs in MIPS.

Offsets for fields in $structstudent$

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

        la      $t0, xavier

        li      $v0, 4                          # syscall 4: print_string
        la      $a0, zid_msg                    #
        syscall                                 # printf("zID: z");

        li      $v0, 1                          # syscall 1: print_int
        lw      $a0, STUDENT_OFFSET_ZID($t0)    #
        syscall                                 # printf("%d", selection->zid);

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

        li      $v0, 4                          # syscall 4: print_string
        la      $a0, first_name_msg             #
        syscall                                 # printf("First name: ");

        li      $v0, 4                          # syscall 4: print_string
        la      $a0, STUDENT_OFFSET_FIRST($t0)  #
        syscall                                 # printf("%s", selection->first);

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

        li      $v0, 4                          # syscall 4: print_string
        la      $a0, last_name_msg              #
        syscall                                 # printf("Last name: ");

        li      $v0, 4                          # syscall 4: print_string
        la      $a0, STUDENT_OFFSET_LAST($t0)   #
        syscall                                 # printf("%s", selection->last);

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

        li      $v0, 4                          # syscall 4: print_string
        la      $a0, program_msg                #
        syscall                                 # printf("Program: ");

        li      $v0, 1                          # syscall 1: print_int
        lw      $a0, STUDENT_OFFSET_PROGRAM($t0)#
        syscall                                 # printf("%d", selection->program);

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

        li      $v0, 4                          # syscall 4: print_string
        la      $a0, alias_msg                  #
        syscall                                 # printf("Alias: ");

        li      $v0, 4                          # syscall 4: print_string
        la      $a0, STUDENT_OFFSET_ALIAS($t0)  #
        syscall                                 # printf("%s", selection->alias);

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


        li      $v0, 0                          #
        jr      $ra                             # return 0;
        .data
abiram:                                         # struct student abiram {
        .word   5308310                         #   int zid;
        .asciiz "Abiram"                        #   char first[20];
        .space  20 - 7
        .asciiz "Nadarajah"                     #   char last[20];
        .space  20 - 10
        .word   3778                            #   int program;
        .asciiz "abiramn"                       #   char alias[10];
        .space  10 - 8
        .align  2
                                                # }

xavier:                                         # struct student xavier {
        .word   5417087                         #   int zid;
        .asciiz "Xavier"                        #   char first[20];
        .space  20 - 7
        .asciiz "Cooney"                        #   char last[20];
        .space  20 - 7
        .word   3778                            #   int program;
        .asciiz "xavc"                          #   char alias[10];
        .space  10 - 5
                                                # }

zid_msg:
        .asciiz "zID: z"

first_name_msg:
        .asciiz "First name: "

last_name_msg:
        .asciiz "Last name: "

program_msg:
        .asciiz "Program: "

alias_msg:
        .asciiz "Alias: "
```

struct_address.c

```c
#include <stdio.h>
#include <stdint.h>

struct s1 {
    uint32_t    i0;
    uint32_t    i1;
    uint32_t    i2;
    uint32_t    i3;
};

struct s2 {
    uint8_t     b;
    uint64_t    l;
};

int main(void) {
    struct s1 v1;

    printf("&v1       = %p\n", &v1);
    printf("&(v1.i0) = %p\n", &(v1.i0));
    printf("&(v1.i1) = %p\n", &(v1.i1));
    printf("&(v1.i2) = %p\n", &(v1.i2));
    printf("&(v1.i3) = %p\n", &(v1.i3));

    printf("\nThis shows struct padding\n");

    struct s2 v2;
    printf("&v2       = %p\n", &v2);
    printf("&(v2.b)   = %p\n", &(v2.b));
    printf("&(v2.l)   = %p\n", &(v2.l));
}
```

struct_packing.c

```
$ dcc struct_packing.c -o struct_packing
$ ./struct_packing
sizeof v1 = 32
sizeof v2 = 20
alignment rules mean struct s1 is padded
&(v1.c1) = 0x7ffdfc02f560
&(v1.l1) = 0x7ffdfc02f564
&(v1.c2) = 0x7ffdfc02f568
&(v1.l2) = 0x7ffdfc02f56c
struct s2 is not padded
&(v2.c1) = 0x7ffdfc02f5a0
&(v2.l1) = 0x7ffdfc02f5a4
$
```

```c
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>

void print_bytes(void *v, int n);

struct s1 {
    uint8_t     c1;
    uint32_t    l1;
    uint8_t     c2;
    uint32_t    l2;
    uint8_t     c3;
    uint32_t    l3;
    uint8_t     c4;
    uint32_t    l4;
};

struct s2 {
    uint32_t    l1;
    uint32_t    l2;
    uint32_t    l3;
    uint32_t    l4;
    uint8_t     c1;
    uint8_t     c2;
    uint8_t     c3;
    uint8_t     c4;
};

int main(void) {
    struct s1 v1;
    struct s2 v2;

    printf("sizeof v1 = %lu\n", sizeof v1);
    printf("sizeof v2 = %lu\n", sizeof v2);

    printf("alignment rules mean struct s1 is padded\n");

    printf("&(v1.c1) = %p\n", &(v1.c1));
    printf("&(v1.l1) = %p\n", &(v1.l1));
    printf("&(v1.c2) = %p\n", &(v1.c2));
    printf("&(v1.l2) = %p\n", &(v1.l2));

    printf("struct s2 is not padded\n");

    printf("&(v1.l1) = %p\n", &(v1.l1));
    printf("&(v1.l2) = %p\n", &(v1.l2));
    printf("&(v1.l4) = %p\n", &(v1.l4));
    printf("&(v2.c1) = %p\n", &(v2.c1));
    printf("&(v2.c2) = %p\n", &(v2.c2));
}
```

pointer.c

demonstrate implementaion of pointers by an address

```c
#include <stdio.h>

int answer = 42;

int main(void) {
    int i;
    int *p;

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

    return 0;
}
```

pointer.s

demonstrate implementation of pointers by an address p in register $t0 i in register $t1 $t2 used for temporary value

```
main:
    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

    li   $v0, 0      # return 0 from function main
    jr   $ra         #


    .data
answer:
    .word 42         # int answer = 42;
```

pointer5.c

print an array using pointers

```
#include <stdio.h>

int numbers[5] = { 3, 9, 27, 81, 243};

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

pointer5.simple.c

print an array using pointers

```
#include <stdio.h>

int numbers[5] = { 3, 9, 27, 81, 243};

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;
}
```

pointer5.s

print an array using pointers 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:

    li   $v0, 0          # return 0
    jr   $ra

.data

numbers:                 # int numbers[10] = { 3, 9, 27, 81, 243};
     .word 3, 9, 27, 81, 243
```

pointer5.faster.s

print 5 numbers - 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, '\n'       #   printf("%c", '\n');
    li   $v0, 11
    syscall
    addi $t0, $t0, 4     #   p++
    ble  $t0, $t1, loop  # if (p <= q) goto loop;

    li   $v0, 0          # return 0
    jr   $ra

.data

numbers:                 # int numbers[10] = { 3, 9, 27, 81, 243};
     .word 3, 9, 27, 81, 243
```

emulating_array_indexing.c

non-portable code illustrating array indexing this relies on pointers being implemented by memory addresses which most compiled C implementations do

```c
#include <stdio.h>
#include <stdint.h>

uint32_t array[10] = {10, 11, 12, 13, 14, 15, 16, 17, 18, 19};

int main(void) {
    // use a typecast to assign array address to integer variable i
    // better than uint64_t would be uintptr_t - an unsigned integer type the same size as a pointer.
    uint64_t i = (uint64_t)&array;

    i += 7 * sizeof array[0]; // add 28 to i

    // use a typecast to assign  i to a pointer vaiable
    uint32_t *y = (uint32_t *)i;

    printf("*y = %d\n", *y); // prints 17

    // compare to pointer arithmetic where adding 1
    // moves to the next array element
    uint32_t *z = array;
    z += 7;
    printf("*z = %d\n", *z); // prints 17
}
```

emulating_2d_array_indexing.c

non-portable code illustrating 2d-array indexing this relies on pointers being implemented by memory addresses which most compiled C implementations do

```c
#include <stdio.h>
#include <stdint.h>


uint32_t array[3][4] = {{10, 11, 12, 13}, {14, 15, 16, 17}, {18, 19, 20, 21}};

int main(void) {
    // use a typecast to assign array address to integer variable i
    // `uintptr_t': unsigned integer type the same size as a pointer.

    int index1 = 1;
    int index2 = 2;
    printf("array[%d][%d] = %d\n", index1, index2, array[index1][index2]); // prints 16

    uint64_t i = (uint64_t)&array;
    // i += (index1 *  4 * 4) + index2 * 4
    i += (index1 * sizeof array[0]) + index2 * sizeof array[0][0];

    // use a typecast to assign  i to a pointer vaiable
    uint32_t *y = (uint32_t *)i;

    printf("*y = %d\n", *y); // prints 16

}
```

emulating_struct_addressing.c

non-portable code illustrating access to a struct field this relies on pointers being implemented by memory addresses which most compiled C implementations do

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>

struct simple {
    char     c;
    uint32_t i;
    double   d;
};

struct simple s = { 'Z', 42, 3.14159 };

int main(void) {
    // use a typecast to assign struct address to integer variable i
    // `uintptr_t': unsigned integer type the same size as a pointer.
    uintptr_t i = (uintptr_t)&s;

    // 3 bytes of padding - likely but not guaranteed
    i += (sizeof s.c) + 3;
    // use a typecast to assign  i to a pointer vaiable
    uint32_t *y = (uint32_t *)i;

    printf("*y = %d\n", *y); // prints 42
}
```

sizeof.c

Print the sizes of various types.
Compile with the -m32 flag to target a 32-bit platform.

```
#include <stdio.h>

int main(void) {

    printf("sizeof(char) is %zu bytes\n", sizeof(char));
    printf("sizeof(int) is %zu bytes\n", sizeof(int));
    printf("sizeof(float) is %zu bytes\n", sizeof(float));
    printf("sizeof(double) is %zu bytes\n", sizeof(double));
    // All pointers are just memory addresses - which
    // are all the same size:
    printf("sizeof(char *) is %zu bytes\n", sizeof(char *));
    printf("sizeof(int *) is %zu bytes\n", sizeof(int *));
    printf("sizeof(void *) is %zu bytes\n", sizeof(void *));
    
}
```

array.c

Print an array of characters.

```
#include <stdio.h>

#define ARRAY_LEN 5

int main(void) {
    char array[ARRAY_LEN] = {'h', 'e', 'l', 'l', 'o'};

    for (int i = 0; i < ARRAY_LEN; i++) {
        printf("array[%d] = %c = %c = %c, ", i, array[i], i[array], *(array + i));
        printf("&array[%d] = %p = %p\n", i, &array[i], array + i);
        // &array[i] = array + 1 * i  - for an array of characters
        // &array[i] = array + sizeof(element) * i  - in general

        // Because addition is commutative, the following are equivalent:
        // array[i] = *(array + i);
        // i[array] = *(i + array)
    }
    return 0;
}

// What if we had
// int array[ARRAY_LEN] = {3, 1, 4, 1, 5}; ?
// &array[i] = array + 4 * i for an array of integers
```

array_bytes.s

Print each element from an array of bytes.

```
ARRAY_LEN = 5

        .text
main:
        # Locals:
        # - $t0: int i
        # - $t1: temporary result

array_loop__init:
        li      $t0, 0                          # int i = 0;
array_loop__cond:
        bge     $t0, ARRAY_LEN, array_loop__end # while (i < ARRAY_LEN) {
array_loop__body:
        li      $v0, 11                         #   syscall 11: print_char

        # Method 1: performing the arithmetic
        # from scratch.
        # la    $t1, array                      #   (array
        # add   $t1, $t1, $t0                   #      + i)
        # lb    $a0, ($t1)                      #
        # syscall                               #   putchar(*(array + i));

        # Method 2: Letting mipsy generate the
        # appropriate instructions from this
        # pseudoinstruction.
        lb      $a0, array($t0)                 #   (array + i)
        syscall                                 #   putchar(*(array + i));


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

array_loop__step:
        addi    $t0, $t0, 1                     #   i++;
        b       array_loop__cond                # }
array_loop__end:
        li      $v0, 0
        jr      $ra                             # return 0;

        .data
array:
        .byte   'h', 'e', 'l', 'l', 'o'         # char array[ARRAY_LEN] = {'h', 'e', 'l', 'l', 'o'};
```

array_words.s

Print each element from an array of integers.

```
ARRAY_LEN = 5

        .text
main:
        # Locals:
        # - $t0: int i
        # - $t1: temporary result

array_loop__init:
        li      $t0, 0                          # int i = 0;
array_loop__cond:
        bge     $t0, ARRAY_LEN, array_loop__end # while (i < ARRAY_LEN) {
array_loop__body:
        li      $v0, 1                          #   syscall 1: print_int
        mul     $t1, $t0, 4                     #   (4 * i
        lw      $a0, array($t1)                 #    + array)
        syscall                                 #   printf("%d", *(array + 4 * i));

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

array_loop__step:
        addi    $t0, $t0, 1                     #   i++;
        b       array_loop__cond                # }
array_loop__end:
        li      $v0, 0
        jr      $ra                             # return 0;

        .data
array:
        .word   3, 1, 4, 1, 5                   # int array[ARRAY_LEN] = {3, 1, 4, 1, 5};
```

scan_and_print.c

Scan in 10 integers into an array and then print them out

```
#include <stdio.h>
#define N_ELEMENTS 10

int main(void) {
    int i;
    int numbers[N_ELEMENTS] = {0};

read_loop__init:
    i = 0;
read_loop__cond:
    if (i >= N_ELEMENTS) goto read_loop__end;
read_loop__body:
    scanf("%d", &numbers[i]);   // &numbers[i] == &numbers[0] + i * sizeof(int)
                                //              = numbers + i * 4
read_loop__step:
    i++;
    goto read_loop__cond;

read_loop__end:

print_loop__init:
    i = 0;
print_loop__cond:
    if (i >= N_ELEMENTS) goto print_loop__end;
print_loop__body:
    printf("%d", numbers[i]);   // &numbers[i] == &numbers[0] + i * sizeof(int)
                                //              = numbers + i * 4
    putchar(' ');
print_loop__step:
    i++;
    goto print_loop__cond;
print_loop__end:
    return 0;
}
```

scan_and_print.s

Scan in 10 integers into an array and then print them.

```
N_ELEMENTS = 10

main:
        # Locals:
        # $t0: int i
        # $t1: intermediate result

read_loop__init:
        li      $t0, 0                          # i = 0;

read_loop__cond:
        bge     $t0, N_ELEMENTS, read_loop__end #

        li      $v0, 5                          # syscall 5: read_int
        syscall                                 #
        mul     $t1, $t0, 4                     # (4 * i
        add     $t1, $t1, numbers               #   + numbers)
        sw      $v0, ($t1)                      # scanf("%d", &numbers[i]);

read_loop__step:
        addi    $t0, $t0, 1                     # i++;
        b       read_loop__cond

read_loop__end:


print_loop__init:
        li      $t0, 0                          # int i = 0;

print_loop__cond:
        bge     $t0, N_ELEMENTS, print_loop__end

print_loop__body:

        li      $v0, 1                          # syscall 1: print_int
        mul     $t1, $t0, 4                     # (4 * i
        lw      $a0, numbers($t1)               #   + numbers)
        syscall                                 # printf("%d", numbers[i]);


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

print_loop__step:
        addi    $t0, $t0, 1                     # i++;
        b       print_loop__cond

print_loop__end:

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


        .data
numbers:
        .word   0:N_ELEMENTS
```

COMP1521 24T2: Computer Systems Fundamentals is brought to you by
the School of Computer Science and Engineering
at the University of New South Wales, Sydney.
For all enquiries, please email the class account at cs1521@cse.unsw.edu.au
CRICOS Provider 00098G