DPST1092 23T2 — MIPS Data

https://www.cse.unsw.edu.au/~dp1092/23T2/

mipsy Memory Layout

MIPS addresses are 32 bits

Region	Address	Notes
text	0x00400000	contains only instructions; read-only; cannot expand
data	0x10000000	data objects; readable/writeable; can be expanded
stack	0x7fffefff	grows from that address; readable/writeable
k_text	0x80000000	kernel code; read-only; only accessible kernel mode
k_data	0x90000000	kernel data; read/write; only accessible kernel mode

Note: there is no heap like there is in C, but the data segment can expand.

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

    .globl         # make symbol available globally

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};
bb: .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'};
```

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 or static variable

Global/Static Variables

Global and static variables need an appropriate number of bytes allocated in `.data` segment, using **`.space`**:

```
char c;                    c: .space 1
int  val;                val: .space 4
char str[20];            str: .space 20
int  vec[20];            vec: .space 80
```

Initialised to 0 by default ... other directives allow initialisation to other values:

```
char c = 'A';                c: .byte 'A'
int val = 5;               val: .word 5
int arr[4] = {9,8,7,6};    arr: .word 9, 8, 7, 6
char msg[7] = "Hello\n";   msg: .asciiz "Hello\n"
```


Operand Sizes

MIPS instructions can manipulate different-sized operands

- single bytes, two bytes ("halfword"), four bytes ("word")

Many instructions also have variants for signed and unsigned

Leads to many opcodes for a (conceptually) single operation, e.g.

- `lb/sb` ... load or store one byte from or to specified address
- `lh/sh` ... load or store two bytes from or to specified address
- `lw/sw` ... load or store four bytes (one word) from or to specified address
- `lbu` ... load unsigned byte from specified address
- `lhu` ... load unsigned 2-bytes from specified address

All of the above specify a destination register

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

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
  - eg if you are using lw, you must be loading the 4 bytes from an address divisible by 4
  - eg if you are using sh, you must be storing the 2 bytes at an address divisible by 2

Exercise: Operand Sizes

Consider the following memory contents and instructions

```
Label Size     Content      Address       Instructions
w:    .word    0x00010101   0x10000000    * la   $t0, w
x:    .word    0x00008000   0x10000004    * lw   $t0, w
y:    .byte    0x00000061   0x10000008    * lh   $t0, w
z:    .byte    0x00000062   0x10000009    * lh   $t0, x
                                          * lhu  $t0, x
                                          * lbu  $t0, x
                                          * lb   $t0, z
                                          * lw   $t0, z
                                          * lw   $t0, y
```

What will be the value (in hexadecimal) of the destination register after each of the starred `mipsy` instructions is executed?

Addressing Modes

Memory addresses can be given by

- symbolic name (label) (effectively, a constant)
- indirectly via a register (effectively, pointer dereferencing)

Examples:

```
prog:
a:    lw    $t0, var      # direct addressing via name
bb:   lw    $t0, ($s0)    # indirect addressing
c:    lw    $t0, 4($s0)   # indexed addressing
d:    lw    $t0, vec($s1)# indexed addressing
e:    lw    $t0, vec+4    # direct addressing via name + bytes
```

If `$s0` contains `0x10000000`,`$s1` contains `0x00000008` , `&var` = `0x100000008` and `&vec` = `0x1000000C`

- computed address for `a:` is `0x100000008`
- computed address for `b:` is `0x100000000`
- computed address for `c:` is `0x100000004`
- computed address for `d:` is `0x100000014`
- computed address for `e:` is `0x100000010`

Addressing Modes

Format	Address computation
(register)	address = contents of register
k	address = k
k(register)	address = k + contents of register
symbol	address = &symbol = address of symbol
symbol ± k	address = &symbol ± k
symbol ± k(register)	address = &symbol ± (k + contents of register)

where *k* is a literal constant value (e.g. 4 or `0x10000000`)

Addressing Modes Example

Examples of load/store and addressing:

```
main:
    la   $t0, vec       # reg[t0] = &vec
    li   $t1, 5         # reg[t1] = 5
    sw   $t1, ($t0)     # vec[0] = reg[t1]
    li   $t1, 13        # reg[t1] = 13
    sw   $t1, 4($t0)    # vec[1] = reg[t1]
    li   $t1, -7        # reg[t1] = -7
    sw   $t1, vec+8     # vec[2] = reg[t1]
    li   $t2, 12        # reg[t2] = 12
    li   $t1, 42        # reg[t1] = 42
    sw   $t1, vec($t2)  # vec[3] = reg[t1]

    .data
vec: .space   16        # int vec[4];
```


Exercise: Addressing Modes

Consider the following memory contents and MIPS instructions

```
Label  Address     Content        Instructions
x:     0x10010000  0x00010101   * la $t0, x
y:     0x10010004  0x10010000   * lw $t0, x
z:     0x10010008  0x0000002A     la $s0, z
eol:   0x1001000C  0x0000000A   * lw $t0, ($s0)
                                  li $s0, 8
                                * lw $t0, y($s0)
                                  lw $s0, y
                                * lw $t0  ($s0)
                                  li $s0, 4
                                * lw $t0, x+4($s0)
```

What will be (a) the computed address, (b) the value of the destination register (`$t0` or `$s0`) after each of the starred MIPS instructions is executed?

Implementing Pointers in MIPS

C

```c
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;
}
```

source code for pointer.c

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
li   $v0, 0      # return 0 from function
```

source code for pointer.s

1-d Arrays in MIPS

Can be named/initialised as:

```
    vec:  .space 40      # could be either int vec[10] or char vec[40]


    nums: .word 1, 3, 5, 7, 9         # int nums[6] = {1,3,5,7,9}
    str:  .byte 'a', 'b', 'c', '\0' # char str[] = {'a','b','c','\0'}
    str2: .asciiz "abc"               # char str2[] = "abc"
```

Can access elements via index or pointer

- either approach needs to account for size of elements

Arrays passed to functions via pointer to first element

- must also pass array size, since not available elsewhere

Printing Array: C to simplified C

C

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

source code for print5.c

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;
}
```

source code for print5.simple.c

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("%c", '\n');
    li   $v0, 11
    syscall
    addi $t0, $t0, 1     #   i++
    b    loop            # goto loop
end:
```

source code for print5.s

Printing Array: MIPS (continued)

```
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

Reading and printing 10 Numbers

C

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

source code for read10.c

MIPS

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

source code for read10.s

Reading and Printing 10 Numbers #2

C

```
i = 0;
while (i < 10) {
    printf("%d\n", numbers[i]);
    i++;
}
```

source code for read10.c

MIPS

```
    li   $t0, 0          # i = 0
loop1:
    bge  $t0, 10, end1   # while (i < 10)
    mul  $t1, $t0, 4     #   calculate &nu
    la   $t2, numbers    #
    add  $t3, $t1, $t2   #
    lw   $a0, ($t3)      #   load numbers[
    li   $v0, 1          #   printf("%d",
    syscall
    li   $a0, '\n'       #   printf("%c",
    li   $v0, 11
    syscall
    addi $t0, $t0, 1     #   i++
    b    loop1           # }
end1:
    li   $v0, 0          # return 0
    jr   $ra
```


Printing Array with Pointers: C to simplified C

C

```
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;
}
```

source code for pointer5.c

Simplified C

```
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;
}
```

source code for pointer5.simple.c

Printing Array with Pointers: MIPS

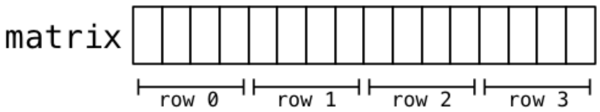
```
# p in $t0, q in $t1
main:
    la   $t0, numbers    # int *p = &numbers[0];
    la   $t0, numbers    # int *q = &numbers[4];
    addi $t1, $t0, 16    #
loop:
    bgt  $t0, $t1, end   # if (p > q) goto end;
    lw   $a0, 0($t0)     # int j = *p;
    li   $v0, 1
    syscall
    li   $a0, '\n'       #   printf("%c", '\n');
    li   $v0, 11
    syscall
    addi $t0, $t0, 4     #   p++
    b    loop            # goto loop
end:
```

source code for pointer5.s

2-d arrays in MIPS

2-d array representation:

```
int matrix[4][4];
```




```
matrix: .space 64
```

Now consider summing all elements

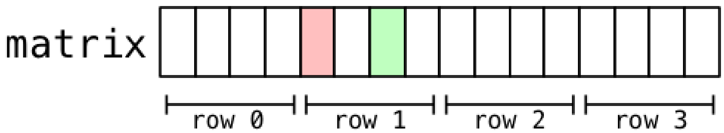
```
int i, j, sum = 0;
for (i = 0; i < 4; i++)
   for (j = 0; j < 4; j++)
      sum += matrix[i][j];
```


2-d arrays in MIPS

Accessing elements:

```
x = matrix[1][2];
```

Find start of row 1, then add offset 2 within row



2-d arrays in MIPS

Computing sum of all elements in `int matrix[6][5]` in 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; col++) {
      sum += matrix[row][col];
   }
}
```

2-d arrays in MIPS

Computing sum of all elements for `int matrix[6][5]`

```
        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:
```


Printing 2-d Array: C to simplified 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;
}
```

source code for print2d.c

Simplified 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;
}
```


Printing 2-d Array: 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
```

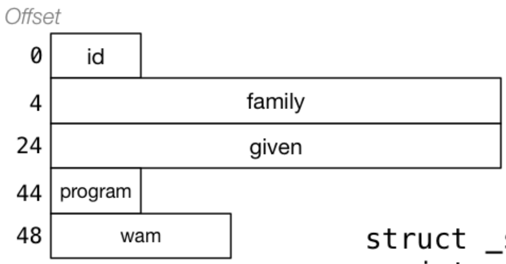
Printing 2-d Array: MIPS (continued)

```
    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
```

source code for print2d.s

structs in MIPS

C `structs` hold a collection of values accessed by name




```
struct _student {
    int     id;
    char    family[20];
    char    given[20];
    int     program;
    double  wam;
};
```


structs in MIPS

C `struct` definitions effectively define a new type.

```
// new type called "struct _student"
struct _student {...};
// new type called Student
typedef struct _student Student;
```

Instances of structures can be created by allocating space:

```
                        # sizeof(Student) == 56
stu1:                   #Student stu1;
    .space 56
stu2:                   #Student stu2;
    .space 56
stu:
    .space 4            #Student *stu;
```

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;
```


Exercise: Printing out details from a struct

Implement the following in MIPS

```
struct details {
    uint16_t  postcode;
    char      first_name[7];
    uint32_t  zid;
};
struct details student = {2052, "Andrew", 5123456};
int main(void) {
    printf("%d", student.zid);
    putchar(' ');
    printf("%s", student.first_name);
    putchar(' ');
    printf("%d", student.postcode);
    putchar('\n');
    return 0;
}
```

source code for student.c