COMP1521 22T1 — MIPS Data

https://www.cse.unsw.edu.au/~cs1521/22T1/
The Memory Subsystem

- memory subsystem typically provides capability to load or store **bytes**
- 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 (not covered in this course)
- operating systems on general purpose computers typically provide **virtual memory** (covered later in this course)
most general purpose computers e.g PCs running Windows, use 64-bit addresses
still some using 32-bit addresses

CSE servers use 64-bit addresses

special purpose (embedded) CPUs may use 64, 32, 16, 8 bit addresses

on the MIPS32 machine implemented by mipsy & spim, all addresses are 32-bit
so in COMP1521 assembler we'll be using 32-bit addresses

there are 64-bit MIPS CPUs
### Accessing Memory on the MIPS

- Addresses are 32 bit

- Only load/store instructions access memory on the MIPS

- 1 byte (8-bit) loaded/stored with `lb`/`sb`

- 2 bytes (16-bit) called a **half-word**, loaded/stored with `lh`/`sh`

- 4 bytes (32-bits) called a **word**, loaded/stored with `lw`/`sw`

- Memory address used for load/store instructions is sum of a specified register and a 16-bit constant (often 0) which is part of the instruction.

- For `sb` & `sh` operations low (least significant) bits of source register are used.

- `lb`/`lh` assume byte/halfword contains a 8-bit/16-bit **signed** integer
  - High 24/16-bits of destination register set to 1 if 8-bit/16-bit integer negative

- Unsigned equivalents `lbu` & `lhu` assume integer is **unsigned**
  - High 24/16-bits of destination register always set to 0
# MIPS Load/Store Instructions

<table>
<thead>
<tr>
<th>assembly</th>
<th>meaning</th>
<th>bit pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>lb</strong> $r_t, I(r_s)$</td>
<td>$r_t = \text{mem}[r_s + I]$</td>
<td>100000ssssstttttIIIIIIIIIIIIIIII</td>
</tr>
<tr>
<td><strong>lh</strong> $r_t, I(r_s)$</td>
<td>$r_t = \text{mem}[r_s + I] \mid \text{mem}[r_s + I+1] \ll 8$</td>
<td>100001ssssstttttIIIIIIIIIIIIIIII</td>
</tr>
<tr>
<td><strong>lw</strong> $r_t, I(r_s)$</td>
<td>$r_t = \text{mem}[r_s + I] \mid \text{mem}[r_s + I+1] \ll 8 \mid \text{mem}[r_s + I+2] \ll 16 \mid \text{mem}[r_s + I+3] \ll 24$</td>
<td>100011ssssstttttIIIIIIIIIIIIIIII</td>
</tr>
<tr>
<td><strong>sb</strong> $r_t, I(r_s)$</td>
<td>$\text{mem}[r_s + I] = r_t &amp; 0xff$</td>
<td>101000ssssstttttIIIIIIIIIIIIIIII</td>
</tr>
<tr>
<td><strong>sh</strong> $r_t, I(r_s)$</td>
<td>$\text{mem}[r_s + I] = r_t &amp; 0xff \mid \text{mem}[r_s + I+1] = r_t \gg 8 &amp; 0xff$</td>
<td>101001ssssstttttIIIIIIIIIIIIIIII</td>
</tr>
<tr>
<td><strong>sw</strong> $r_t, I(r_s)$</td>
<td>$\text{mem}[r_s + I] = r_t &amp; 0xff \mid \text{mem}[r_s + I+1] = r_t \gg 8 &amp; 0xff \mid \text{mem}[r_s + I+2] = r_t \gg 16 &amp; 0xff \mid \text{mem}[r_s + I+3] = r_t \gg 24 &amp; 0xff$</td>
<td>101011ssssstttttIIIIIIIIIIIIIIII</td>
</tr>
</tbody>
</table>

---

https://www.cse.unsw.edu.au/~cs1521/22T1/
# simple example of load & storing a byte
# we normally use directives and labels

main:
  li $t0, 42
  li $t1, 0x10000000
  sb $t0, 0($t1)  # store 42 in byte at address 0x10000000
  lb $a0, 0($t1)  # load $a0 from same address
  li $v0, 1      # print $a0
  syscall
  li $a0, '
'     # print '
'
  li $v0, 11     syscall
  li $v0, 0      # return 0
  jr $ra
Assembler Directives

MIPS & SPIM have 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};
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'};
```
Code example: storing and loading a value

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

Source code for load_store.s

https://www.cse.unsw.edu.au/~cs1521/22T1/
Testing Endian-ness

C

```c
uint8_t b;
uint32_t u;
u = 0x03040506;
// load first byte of u
b = *(uint8_t *)&u;
// prints 6 if little-endian
// and 3 if big-endian
printf("%d\n", b);
```

source code for endian.c

MIPS

```mips
li $t0, 0x03040506
la $t1, u
sw $t0, 0($t1) # u = 0x03040506;
lb $a0, 0($t1) # b = *(uint8_t *)&u;
li $v0, 1 # printf("%d", a0);
syscall
li $a0, \n' # printf("%c", '\n');
li $v0, 11
syscall
li $v0, 0 # return 0
jr $ra
```

.data

```mips
u:
.space 4
```

source code for endian.s
Setting A Register to An Address

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

```assembly
la $t8, start
```

- The memory address will be fixed before the program is run, so this differs only syntactically from the `li` instruction.

- 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/SPIM short-cuts

- MIPSY & SPIM allow the constant which is part of load & store instructions can be omitted in the common case it is 0.

\[
\text{sb} \ \$t0, \ 0(\$t1) \quad \# \text{store} \ \$t0 \ \text{in byte at address in} \ \$t1 \\
\text{sb} \ \$t0, \ (\$t1) \quad \# \text{same}
\]

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

\[
\text{sb} \ \$t0, \ x \quad \# \text{store} \ \$t0 \ \text{in byte at address labelled} \ x \\
\text{sb} \ \$t1, \ x+15 \quad \# \text{store} \ \$t1 \ 15 \ \text{bytes past address labelled} \ x \\
\text{sb} \ \$t2, \ x(\$t3) \quad \# \text{store} \ \$t2 \ \$t3 \ \text{bytes past address labelled} \ x
\]

- 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
## MIPS/SPIM Memory Layout

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

```c
double val; val: .space 8
char str[20]; str: .space 20
int vec[20]; vec: .space 80
```

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

```c
int val = 5; val: ..double 5
int arr[4] = {9,8,7,6}; arr: .word 9, 8, 7, 6
char msg[7] = "Hello\n"; msg: .asciiz "Hello\n"
```
add: local variables in registers

C

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

MIPS

```mips
main:
    # x in $t0
    # y in $t1
    # z in $t2
    li $t0, 17
    li $t1, 25
    add $t2, $t1, $t0
    # ...
```
add variables in memory (uninitialized)

C

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

MIPS (.text)

```mips
main:
    li $t0, 17       # x = 17;
    la $t1, x
    sw $t0, 0($t1)
    li $t0, 25      # y = 25;
    la $t1, y
    sw $t0, 0($t1)
    la $t0, x
    lw $t1, 0($t0)
    la $t0, y
    lw $t2, 0($t0)
    add $t3, $t1, $t2 # z = x + y
    la $t0, z
    sw $t3, 0($t0)
```

MIPS (.data)

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

source code for add_memory.s
C

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

MIPS .data

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

MIPS .text

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

source code for add_memory_initialized.s

https://www.cse.unsw.edu.au/~cs1521/22T1/COMP1521 22T1 — MIPS Data
add variables in memory (array)

C

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

MIPS .data

```mips
.data
    .word 17, 25, 0
```

MIPS .text

```mips
main:
    la $t0, x
    lw  $t1, 0($t0)
    lw  $t2, 4($t0)
    add $t3, $t1, $t2  # z = x + y
    sw  $t3, 8($t0)
```

source code for `add_memory_array.s`

https://www.cse.unsw.edu.au/~cs1521/22T1/
store value in array element — example 1

C

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

int16_t x[30];

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

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]);
        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:

source code for print5.s
end:
  li  $v0, 0  # return 0
  jr  $ra

.data
numbers:  # int numbers[10] = { 3, 9, 27, 81, 243};
  .word 3, 9, 27, 81, 243
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       # printf("%d\n", i);
syscall
li $a0, '\n'   # printf("\n");
li $v0, 11      # printf("\n");
syscall
li $t2, 27     # *p = 27;
sw $t2, ($t0)   # *p = 27;
lw $a0, answer # printf("%d\n", answer);
li $v0, 1       # printf("%d\n", answer);
syscall
li $a0, '\n'   # printf("\n");
li $v0, 11      # printf("\n");
syscall
li $v0, 0      # return 0 from function main
```

source code for pointer.c

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

```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;
}
```
# 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
```c
int vec[5]={0,1,2,3,4};
// ...
int i = 0
while (i < 5) {
    printf("%d", vec[i]);
    i++;
}
// ....
```

```mips
# ...
li $s0, 0
loop:
    bge $s0, 5, end
    la $t0, vec
    mul $t1, $s0, 4
    add $t2, $t1, $t0
    lw $a0, ($t2)
    li $v0, 1
    syscall
    addi $s0, $s0, 1
    b loop
end:
# ...
.data
vec: .word 0, 1, 2, 3, 4
```

- i in $s0
Example C with unaligned accesses

```c
uint8_t bytes[32];
uint32_t *i = (int *)&bytes[1];
// illegal store - not aligned on a 4-byte boundary
*i = 0x03040506;
printf("%d\n", bytes[1]);
```

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 1
v1: .space 1
v2: .space 4
v3: .space 2
v4: .space 4
    .space 1
    .align 2  # ensure e is on a 4 (2**2) byte boundary
v5: .space 4
    .space 1
v6: .word 0  # word directive aligns on 4 byte boundary

source code for unalign.s

https://www.cse.unsw.edu.au/~cs1521/22T1/
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/~cs1521/22T1/
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 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
C

```c
int vec[5]={0,1,2,3,4};
// ...
int *p = &vec[0];
int *end = &vec[4];
while (p <= end) {
    int y = *p;
    printf("%d", y);
    p++;
}
// ....
```

MIPS

```mips
li $s0, vec
la $t0, vec
add $s1, $t0, 16
loop:
    bgt $s0, $s1, end
    lw $a0, 0($s0)
    li $v0, 1
    syscall
    addi $s0, $s0, 4
    b loop
end:
    .data
vec: .word 0, 1, 2, 3, 4
```

- p in $s0
- end in $s1
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 $s0, 0
    # sum = 0
li $s2, 0
    # row = 0

loop1: bge $s2, 6, end1
    # if (row >= 6) break
li $s4, 0
    # col = 0

loop2: bge $s4, 5, end2
    # if (col >= 5) break
la $t0, matrix
mul $t1, $s2, 20
    # t1 = row*rowsize
mul $t2, $s4, 4
    # t2 = col*intsize
add $t3, $t0, $t1
    # offset = t0+t1
add $t4, $t3, $t2
    # offset = t0+t1
lw $t5, 0($t4)
    # t0 = *(matrix+offset)
add $s0, $s0, $t5
    # sum += t0
addi $s4, $s4, 1
    # col++
j loop2

end2: addi $s2, $s2, 1
    # row++
j loop1

end1:
struct _student {
    int    id;
    char   family[20];
    char   given[20];
    int    program;
    double wam;
};
C `struct` definitions effectively define a new type.

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

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

Instances of structures can be created by allocating space:

```c
# sizeof(Student) == 56
stu1: # student_t stu1;
    .space 56
stu2: # student_t stu2;
    .space 56
stu:   # student_t *stu;
    .space 4
```
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 $s1, stu2  # stu = &stu2;
li $t0, 3707
sw $t0, 44($s1)  # stu->program = 3707;
li $t0, 5034567
sw $t0, 0($s1)  # stu->id = 5034567;
```