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
- addresses are 32 bit on the MIPS CPU we are using
- most general purpose computers now use 64-bit addresses (and there are 64-bit MIPS)
- typically small group of (1,2,4,8,..) bytes can be loaded/stored in single operations
- general purpose computers typically have complex caching 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)

Accessing Memory on the MIPS

- addresses are 32 bit (but there are 64-bit MIPS CPUs)
- 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

assembly	meaning	bit pattern
`lb` r_t, `I`(r_s)	r_t = `mem`[r_s+`I`]	`100000ssssstttttIIIIIIIIIIIIIIII`
`lh` r_t, `I`(r_s)	r_t = `mem`[r_s+`I`] | `mem`[r_s+`I`+1] `<< 8`	`100001ssssstttttIIIIIIIIIIIIIIII`
`lw` r_t, `I`(r_s)	r_t = `mem`[r_s+`I`] | `mem`[r_s+`I`+1] `<<  8` | `mem`[r_s+`I`+2] `<< 16` | `mem`[r_s+`I`+3] `<< 24`	`100011ssssstttttIIIIIIIIIIIIIIII`
`sb` r_t, `I`(r_s)	`mem`[r_s+`I`] = r_t `& 0xff`	`101000ssssstttttIIIIIIIIIIIIIIII`
`sh` r_t, `I`(r_s)	`mem`[r_s+`I`] = r_t `& 0xff` `mem`[r_s+`I`+1] = r_t `>> 8 & 0xff`	`101001ssssstttttIIIIIIIIIIIIIIII`

`sw` r_t, `I`(r_s)	`mem`[r_s+`I`] = r_t `& 0xff`
	`mem`[r_s+`I`+1] = r_t `>>  8 & 0xff`
	`mem`[r_s+`I`+2] = r_t `>> 16 & 0xff`
	`mem`[r_s+`I`+3] = r_t `>> 24 & 0xff`
		`101011ssssstttttIIIIIIIIIIIIIIII`

Code example: storing and loading a value (no labels)

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

source code for load_store_no_label.s

Assembler Directives

SPIM has directive to initialize memory and associate labels with addresses.

```
    .text           # following instructions placed in text
    .data           # following objects placed in data
    .globl          # make symbol available globally
a:  .space 18       # int8_t a[18];
    .align 2        # align next object on 4-byte addr
i:  .word 2         # int32_t i = 2;
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 s[3] {'a','b','c'};
```

Code example: storing and loading a value

```
# simple example of load & storing 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
    syscall
    li   $v0, 0       # return 0
    jr   $ra
.data
x:  .space 1          # set aside 1 byte and associate label x with
```

source code for load_store.s

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

```
    li   $t0, 0x03040506
    la   $t1, u
    sw   $t0, 0($t1) # u = 0x0304
    lb   $a0, 0($t1) # b = *(uint
    li   $v0, 1      # printf("%d
    syscall
    li   $a0, '\n'   # printf("%c
    li   $v0, 11
    syscall
    li   $v0, 0     # return 0
    jr   $ra
    .data
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.

```
la $t8, start
```

- The memory address will be fixed before the program is run, so this differs only syntatctically from the `li` instruction.
- For example, if `vec` is the label for memory address `0x10000100` then these two instructions are equivalent:

```
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 the `lw` instruction.

Specifying Addresses - some SPIM short cuts

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

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

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

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

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

SPIM memory layout

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

Global/Static Variables

- global/static variables need appropriate number of bytes allocated in data segment using `.space`:

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

initialized to 0 by default, other directives allow initialization to other values:

```
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

```
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 (.data)

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

MIPS (.text)

```
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)
```

source code for add_memory.s

add variables in memory (initialized)

C

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

MIPS .data

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

MIPS .text

```
main:
    la   $t0, x
    lw   $t1, 0($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

add variables in memory (array)

C

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

MIPS .text

```
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

MIPS .data

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

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
```

store value in array element - example 2

C

```c
#include <stdint.h>

int16_t x[30];

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

MIPS

```
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
```

Printing Array: C to simplified 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

```
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++
    j    loop              # goto loop
```

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

Printing Array with Pointers: C to simplified C

C

```
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 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++
    j    loop             # goto loop
end:
```


source code for pointer5.s

Printing 1-d Arrays in MIPS - v1

C

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

- i in $s0

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
```

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

Example MIPS with unaligned accesses

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

source code for unalign.s

Example MIPS with unaligned accesses

```
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
```

source code for unalign.s

Data Structures and MIPS

C data structures and their MIPS representations:

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

A `char`, `int` or `double`

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

Printing 1-d Array in MIPS -v2

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++;
 }
 // ....
```

- p in $s0
- end in $s1

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
```

Computing sum of 2-d Array : C

Assume we have a 2d-array:

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

We can sum its value like this in C

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

MIPS directives for an equivalent 2d-array

mips   .data matrix: .space 120 # 6 * 5 == 30 array elements each 4 bytesmips .text

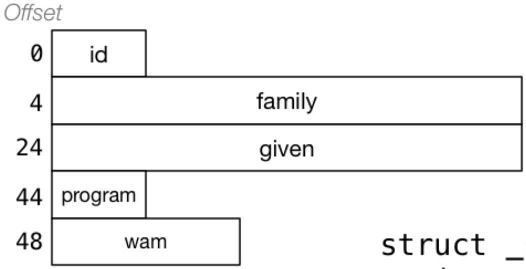
"'

Computing sum of 2-d Array : MIPS

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:
```

Structs in MIPS




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

Implementing Structs in MIPS

C `struct` definitions effectively define a new type.

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

Instances of structures can be created by allocating space:

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

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;
```