The Memory Subsystem

- memory subsystem typically provides capability to load or store bytes (not bits)
  - 1 byte == 8 bits (on general purpose modern machines)
- each byte has unique address, think of:
  - memory as implementing a gigantic array of bytes
  - and the address is the array index
- typically, a small (1, 2, 4, 8, ...) group of bytes can be loaded/stored in a single operation
- general purpose computers typically have complex cache systems to improve memory performance
  - if we have time we'll look at cache systems a little late in this course

Virtual Memory

- operating systems on general purpose computers typically provide virtual memory
- not covered in this course
- allows every program to run as though it has entire address space
  - convenient for programming/compilers
- disconnect address processes use from actual RAM address.
- operating system translates (virtual) address a process uses to an physical (actual) RAM address.
- translation needs to be really fast - needs to be largely implemented in hardware (silicon)
- virtual memory can be several times larger than actual RAM size
- multiple processes can be in RAM, allowing fast switching
- part of processes can be load into RAM on demand.
- provides a mechanism to share memory between processes.
Address Size

- 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, all addresses are 32-bit.

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.
- Signed and unsigned integer representations covered later in course.

MIPS Load/Store Instructions

<table>
<thead>
<tr>
<th>Assembly</th>
<th>Meaning</th>
<th>Bit Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lb r_t, I(r_s)</code></td>
<td>( r_t = \text{mem}[r_s+I] )</td>
<td>( 100000ssssstttttIIIIIIIIIIIIIIII )</td>
</tr>
<tr>
<td><code>lh r_t, I(r_s)</code></td>
<td>( r_t = \text{mem}[r_s+I] ) ( \text{mem}[r_s+I+1] &lt;&lt; 8 )</td>
<td>( 100001ssssstttttIIIIIIIIIIIIIIII )</td>
</tr>
<tr>
<td><code>lw r_t, I(r_s)</code></td>
<td>( r_t = \text{mem}[r_s+I] ) ( \text{mem}[r_s+I+1] &lt;&lt; 8 ) ( \text{mem}[r_s+I+2] &lt;&lt; 16 ) ( \text{mem}[r_s+I+3] &lt;&lt; 24 )</td>
<td>( 100011ssssstttttIIIIIIIIIIIIIIII )</td>
</tr>
<tr>
<td><code>sb r_t, I(r_s)</code></td>
<td>( \text{mem}[r_s+I] = r_t ) &amp; 0xff</td>
<td>( 101000ssssstttttIIIIIIIIIIIIIIIIII )</td>
</tr>
<tr>
<td><code>sh r_t, I(r_s)</code></td>
<td>( \text{mem}[r_s+I] = r_t ) &amp; 0xff</td>
<td>( 101001ssssstttttIIIIIIIIIIIIIIIIII )</td>
</tr>
<tr>
<td><code>sw r_t, I(r_s)</code></td>
<td>( \text{mem}[r_s+I] = r_t ) ( r_t &gt;&gt; 8 ) &amp; 0xff</td>
<td>( 101011ssssstttttIIIIIIIIIIIIIIIIII )</td>
</tr>
</tbody>
</table>
# 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 $v0, 0       # return 0
    jr $ra

source code for load_store_no_label.s
https://www.cse.unsw.edu.au/~cs1521/22T3/ COMP1521 22T3 — MIPS Data 7 / 46

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};
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

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 $v0, 0       # return 0
    jr $ra

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/~cs1521/22T3/ COMP1521 22T3 — MIPS Data 9 / 46
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 syntactically 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 `lw` etc

Specifying Addresses: Some mipsy short-cuts

- `mipsy` allows the constant which is part of load & store instructions can be omitted in the common case it is 0.
  
  ```
  sb $t0, 0($t1)  # store $t0 in byte at address in $t1
  sb $t0, ($t1)  # same
  ```

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

MIPSY 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>.0x7fffffffef</td>
<td>this address and below; read/write</td>
</tr>
<tr>
<td>.ktext</td>
<td>0x80000000</td>
<td>kernel code; read-only; only accessible in kernel mode</td>
</tr>
<tr>
<td>.kdata</td>
<td>0x90000000</td>
<td>kernel data; only accessible in kernel mode</td>
</tr>
</tbody>
</table>
C data structures and their MIPS representations:

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

A char, int or double

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

Global/Static Variables

Global and static variables need an 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
```

Initialised to 0 by default ... other directives allow initialisation 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
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 (.text)

```assembly
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

https://www.cse.unsw.edu.au/~cs1521/22T3/ COMP1521 22T3 — MIPS Data

add variables in memory (initialized)

C

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

MIPS (.text)

```assembly
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)
```

source code for add_memory_initialized.s

https://www.cse.unsw.edu.au/~cs1521/22T3/ COMP1521 22T3 — MIPS Data

add variables in memory (array)

C

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

MIPS (.text)

```assembly
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_arrays.s

https://www.cse.unsw.edu.au/~cs1521/22T3/ COMP1521 22T3 — MIPS Data
double array[10];
for (int i = 0; i < 10; i++) {
    printf("&array[%d]=%p\n", i, &array[i]);
}
printf("\nExample computation for address of array element\n");
uintptr_t a = (uintptr_t)&array[0];
printf("&array[0] + 7 * sizeof (double) = 0x%lx", a + 7 * sizeof (double));
printf("&array[0] + 7 * %lx = 0x%lx", a, 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]);

this code uses types covered later in the course
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;
}

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;
}

Printing Array: MIPS
# print array of ints
# i in $t0
main:
    li $t0, 0
    # int i = 0;
loop:
    bge $t0, 5, 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

Reading and Printing 10 Numbers #1
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", &num.
    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:
    li $v0, 0  # return 0
    jr $ra
.data
numbers:
    .word 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
string0:
    .asciiz "Enter a number: 

source code for read10.s

Reading and Printing 10 Numbers #2
C
i = 0;
while (i < 10) {
    printf("Enter a number: %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 &num.
    la $t2, numbers #
    add $t3, $t1, $t2 #
    lw $a0, ($t3)  # load numbers[
    li $v0, 1
    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
.data
numbers:
    .word 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
string0:
    .asciiz "Enter a number: 

source code for read10.s
Address of C 2-d Array Elements - Code

```c
int array[X][Y];
printf("sizeof array[2][3] = %lu", sizeof array[2][3]);
printf("sizeof array[1] = %lu", sizeof array[1]);
printf("sizeof array = %lu", sizeof array);
for (int x = 0; x < X; x++) {
    for (int y = 0; y < Y; y++) {
        printf("&array[%d][%d] = %p\n", x, y, &array[x][y]);
    }
}
```

Source code for 2d_array_element_address.c

- This code uses types covered later in the course

Address of 2-d C Array Elements - Output

```
$ dcc 2d_array_element_address.c -o 2d_array_element_address
$ ./2d_array_element_address
sizeof array[2][3] = 4
sizeof array[1] = 16
sizeof array = 48
&array=0x7ffd93bb16c0
&array[0]=0x7ffd93bb16c0
&array[0][0]=0x7ffd93bb16c0
&array[0][1]=0x7ffd93bb16c4
&array[0][2]=0x7ffd93bb16c8
&array[0][3]=0x7ffd93bb16cc
&array[1]=0x7ffd93bb16d0
&array[1][0]=0x7ffd93bb16d0
&array[1][1]=0x7ffd93bb16d4
&array[1][2]=0x7ffd93bb16d8
&array[1][3]=0x7ffd93bb16dc
&array[2]=0x7ffd93bb16e0
&array[2][0]=0x7ffd93bb16e0
&array[2][1]=0x7ffd93bb16e4
&array[2][2]=0x7ffd93bb16e8
&array[2][3]=0x7ffd93bb16ec
```

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

```c
.data
matrix: .space 120 # 6 * 5 == 30 array elements each 4 bytes
```

https://www.cse.unsw.edu.au/~cs1521/22T3/ COMP1521 22T3 — MIPS Data 30 / 46
Computing sum of 2-d Array : MIPS

li $t0, 0  # sum = 0
li $t1, 0  # row = 0

loop1: bge $t1, 6, end1  # if (row >= 6) break
li $t2, 0  # col = 0

loop2: bge $t2, 5, end2  # if (col >= 5) break
la $t3, matrix
mul $t4, $t1, 20  # t1 = row*rowsize
mul $t5, $t2, 4  # t2 = col*intsize
add $t6, $t3, $t4  # offset = t0+t1
add $t7, $t6, $t5  # offset = t0+t1
lw $t5, 0($t7)  # t0 = *(matrix+offset)
add $t0, $t0, $t5  # sum += t0
addi $t2, $t2, 1  # col++

j loop2

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

j loop1

end1:

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;
}

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)

```asm
li $a0, ' '  # printf("%c", ' ');
li $v0, 11
syscall
addi $t1, $t1, 1  # j++;

.end2:
li $a0, '\n'  # printf("%c", '\n');
li $v0, 11
syscall
addi $t0, $t0, 1  # i++
!

.end1:
li $v0, 0  # return 0
jr $ra
```

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
- on other architectures, aligned access faster

Example C with unaligned accesses

```c
char bytes[32];
int *i = (int *)&bytes[1];
// illegal store - not aligned on a 4-byte boundary
*i = 42;
printf("%d\n", *i);
```
Example MIPS with unaligned accesses

```assembly
.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/~cs1521/22T3/
```

```
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 succeed 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
```

```
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.

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

// new type called student_t
```c
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:
    .space 4 # student_t *stu;
```

Accessing structure components is by offset, not name
```mips
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;
```

Testing Endian-ness

```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);
```

```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
```

```
.u: .space 4
```
### 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
```assembly
la $t0, answer # p = &answer;
lw $t1, ($t0) # i = *p;
move $a0, $t1 # printf("%d\n", i);
li $v0, 1
syscall
li $a0, '\n' # printf("\n");
li $v0, 11
syscall
li $t2, 27 # *p = 27;
sw $t2, ($t0) #
lw $a0, answer # printf("%d\n", answer);
li $v0, 1
syscall
li $a0, '\n' # printf("\n");
li $v0, 11
syscall
li $v0, 0 # return 0 from function
```

---

### Printing Array with Pointers: C to simplified C

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

#### Simplified C
```c
int main(void) {
    int *p = &numbers[0];
    int *q = &numbers[4];
    loop:
        if (p > q) goto end;
        int j = *p;
        printf("%d", j);
        printf("\n");
        p++;
        goto loop;
    end:
        return 0;
}
```

---

### Printing Array with Pointers: MIPS

#### MIPS
```assembly
# p in $t0, q in $t1
main:
    la $t0, numbers # int *p = &numbers[0];
    la $t0, numbers # int *q = &numbers[4];
    addi $t0, $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("\n");
    li $v0, 11
    syscall
    addi $t0, $t0, 4 # p++
    b loop # goto loop
end:
```

source code for pointer5.s
Printing Array with Pointers: MIPS - faster

# 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;

source code for pointer5.faster.s
https://www.cse.unsw.edu.au/~cs1521/22T3/