

The first step in design is to understand the problem.

What does your assembler have to do?

- for the data segment...?
- for the text segment...?

What information does your assembler have to possess?

- for the data segment...?
- for the text segment...?

How are you going to organize that information?

```
.data
message: .ascii "The sum of the numbers in the array is: "
array: .word 2, 3, 5, 7, 11, 13, 17, 19, 23, 29
array_size: .word 10
```

```
01010100011010000110010100100000
01110011011101010110110100100000
01101111011001100010000001110100
01101000011001010010000001101110
01110101011011010110001001100101
01110010011100110010000001101001
01101110001000000111010001101000
01100101001000000110000101110010
01110010011000010111100100100000
01101001011100110011101000100000
00000000000000000000000000000000
00000000000000000000000000000010
00000000000000000000000000000011
000000000000000000000000000000101
000000000000000000000000000000111
0000000000000000000000000000001011
0000000000000000000000000000001101
00000000000000000000000000000010001
00000000000000000000000000000010011
00000000000000000000000000000010111
00000000000000000000000000000011101
0000000000000000000000000000001010
```

message

array

array\_size

The variable declarations in the data segment must be parsed and translated into a binary representation.

```
.text
main:
    la    $a0, array
    la    $a1, array_size
    lw    $a1, 0($a1)

loop:
    sll   $t1, $t0, 2
    add   $t2, $a0, $t1
    sw    $t0, 0($t2)
    addi  $t0, $t0, 1
    add   $t4, $t4, $t0
    slt   $t3, $t0, $a1
    bne   $t3, $zero, loop

    li    $v0, 4
    la    $a0, message
    syscall
    li    $v0, 1
    or    $a0, $t4, $zero
    syscall
    li    $v0, 10
    syscall
```

The assembly instructions in the text segment must be parsed and translated into a binary representation.



```
00100000000001000010000000100100
001000000000001010010000001001100
10001100101001010000000000000000
00000000000010000100100010000000
00000000100010010101000000100000
10101101010010000000000000000000
00100001000010000000000000000001
00000001100010000110000000100000
00000001000001010101100000101010
00010101011000001111111111111001
0010010000000010000000000000100
00100000000001000010000000000000
00000000000000000000000000001100
00100100000000100000000000000001
00000001100000000010000000100101
00000000000000000000000000001100
00100100000000100000000000001010
00000000000000000000000000001100
```

Keep it simple.

How would YOU translate a particular MIPS assembly instruction to machine code?

Consider: `add $t0, $s5, $s3`

What's the machine code format? (R-type, I-type, J-type, special?)

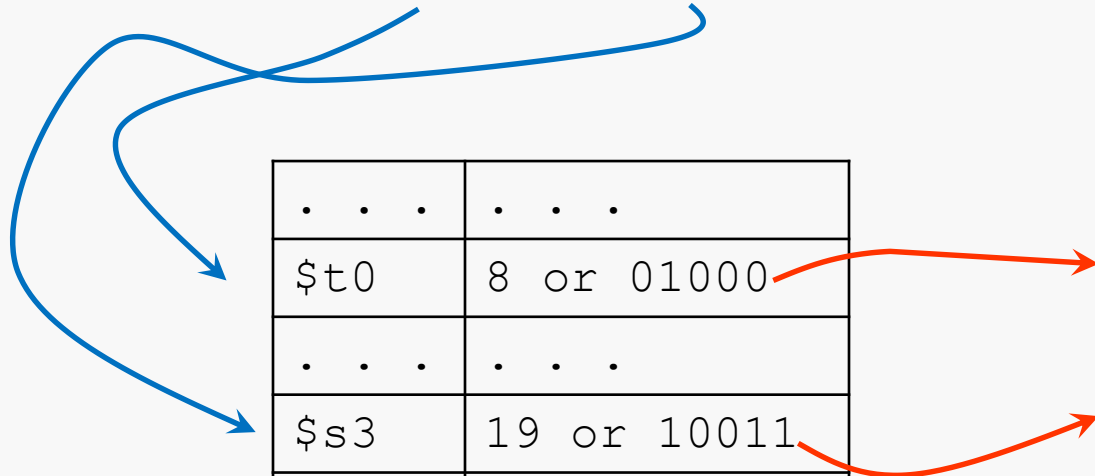
How do you know that?

What are the correct values for the various fields in the machine instruction?

How do you know that?

More to the point... how will your program "know" those things?

Consider: `add $t0, $s5, $s3`



|       |             |
|-------|-------------|
| . . . | . . .       |
| \$t0  | 8 or 01000  |
| . . . | . . .       |
| \$s3  | 19 or 10011 |
| . . . | . . .       |
| \$s5  | 21 or 10101 |
| . . . | . . .       |

Think of the table as defining a mapping from some sort key of value (e.g., symbolic register name) to another sort of value (e.g., register number, binary text string).

What are the key values?

What are the values we want to map the keys to?

Define a `struct` type that associates a particular key value with other values; for instance:

```
struct _RegMapping {    // register name to number
    char* regName;      // symbolic name as C-string
    char* regNumber;    // string for binary representation
};
typedef struct _RegMapping RegMapping;
```

Define an array of those, and initialize appropriately; for instance:

```
static RegMapping Table[...] = {
    {"$zero", "00000"},
    {"$at",   "00001"},
    . . .
    {"$t0",   "01000"},
    . . .
    {"$ra",   "11111"}
};
```

Define a function to manage the lookup you need and you're in business...

Put the initialization of your table at file scope in an appropriate .c file:

```
// Register lookup module .c file
. . .
struct _RegMapping {
    . . .
};
typedef struct _RegMapping RegMapping;
. . .
static RegMapping Table[...] = {
    . . .
};

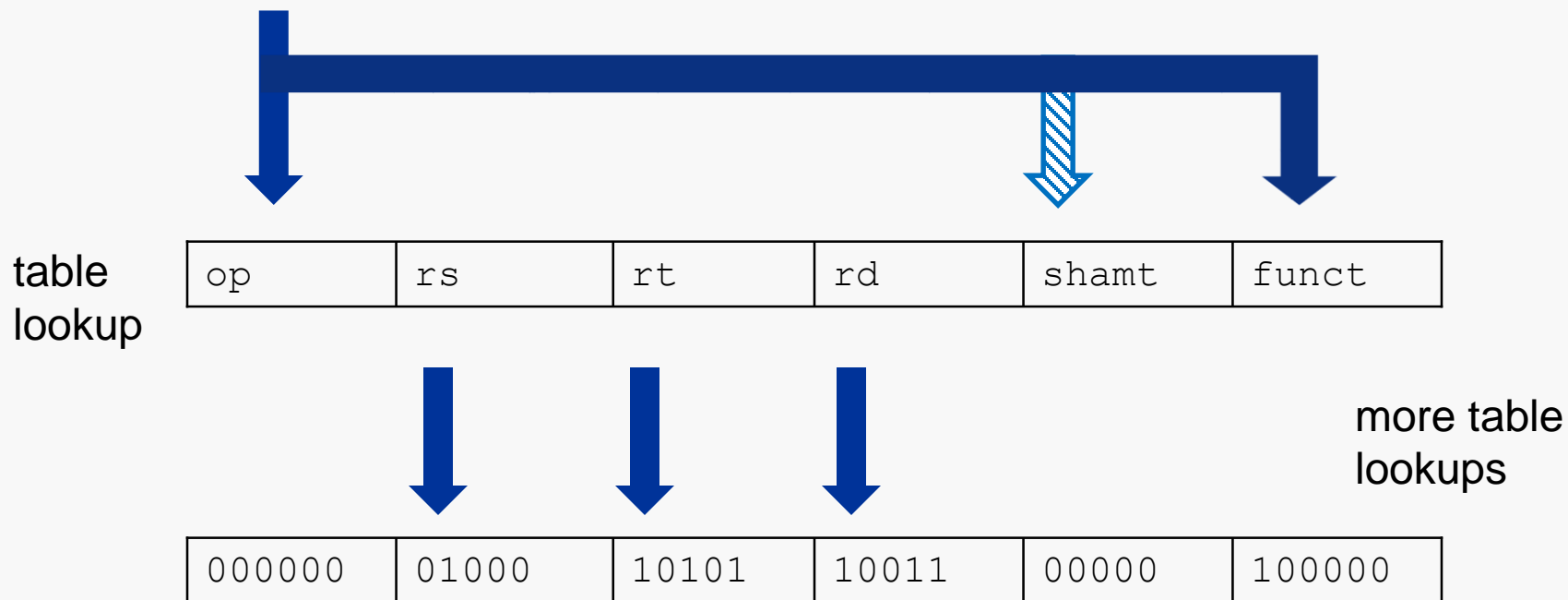
// definitions of lookup functions
. . .
```

Now:

- the table is created automatically when your program starts
- the table exists the whole time your program is running
- the table can only be accessed by calling the functions you "publish" in a .h file



Consider: `add $t0, $s5, $s3`



If we have the right tables and we break the assembly instruction into its parts, it's easy to generate the machine instruction...

One basic design decision is how to represent various things in the solution.

For the machine instruction, we have (at least) two options:

```
char    MI[. . .];    // array of chars '0' and '1'
```

```
uint32_t MI;          // sequence of actual bits
```

Either will work.

Each has advantages and disadvantages.

But the option you choose will affect things all throughout the design... so decide early!

Either way, you have to decide how to put the right bits at the right place in your representation of the machine instruction.

*memcpy()  
strncpy()  
snprintf()*

*vs*

*bit  
fiddling*

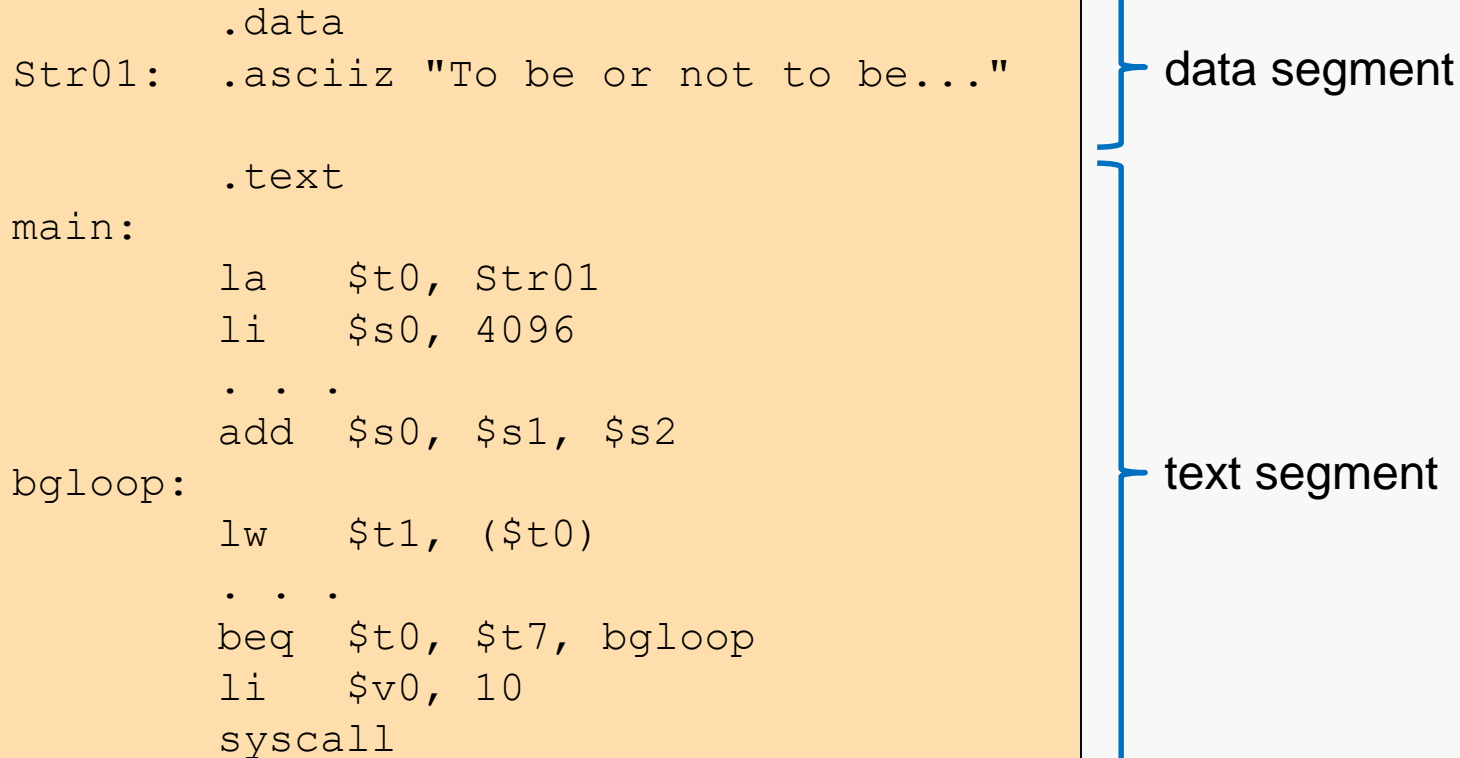
Alas, C does not provide any standard format specifiers (or some other feature) for displaying the bits of a value. But, we can always roll our own:

```
void printByte(FILE *fp, uint8_t Byte) {  
  
    uint8_t Mask = 0x80;    // 1000 0000  
  
    for (int bit = 8; bit > 0; bit--) {  
  
        fprintf(fp, "%c", ( (Byte & Mask) == 0 ? '0' : '1') );  
  
        Mask = Mask >> 1;    // move 1 to next bit down  
    }  
}
```

It would be fairly trivial to modify this to print the bits of "wider" C types.

It would also be easy to modify this to put the characters into an array...

But, execution of the assembler starts with an assembly program file, like:



The diagram shows an assembly program file with two segments. The first segment, labeled 'data segment', contains the following code:

```
.data
Str01: .asciiz "To be or not to be..."
```

The second segment, labeled 'text segment', contains the following code:

```
.text
main:
    la    $t0, Str01
    li    $s0, 4096
    . . .
    add   $s0, $s1, $s2
bgloop:
    lw    $t1, ($t0)
    . . .
    beq   $t0, $t7, bgloop
    li    $v0, 10
    syscall
```

Blue brackets on the right side of the code block group the first two lines under 'data segment' and the remaining lines under 'text segment'.

The logic of parsing is different for the data segment and the text segment.

So is the logic of translation to text-binary form.

How are you going to handle the high-level tasks of identifying instructions/variables?

Doing this by hand, you'd probably think of grabbing a line at a time and processing it.

```
    . . .  
    .text  
main:  
    la    $t0, Str01  
    li    $s0, 4096  
    . . .  
    add   $s0, $s1, $s2  
bgloop:  
    lw    $t1, ($t0)  
    . . .  
    li    $v0, 10  
    syscall
```

C provides a number of useful library functions:

*fgets()  
sscanf()  
strtok()*

safely read a  
line of text

read  
formatted  
values from  
a C-string

break a C-  
string into  
delimited  
pieces,  
destructively

Consider: `add $t0, $s5, $s3`

The specification says some things about the formatting of assembly instructions.

Those things will largely determine how you split an instruction into its parts.

And, don't forget that different instructions take different numbers and kinds of parameters:

*How would we figure out the number and kinds of parameters at runtime?*

```
    . . .  
    la    $t0, Str01  
    li    $s0, 4096  
    . . .  
    add   $s0, $s1, $s2  
bgloop:  
    lw    $t1, ($t0)  
    . . .  
    syscall
```

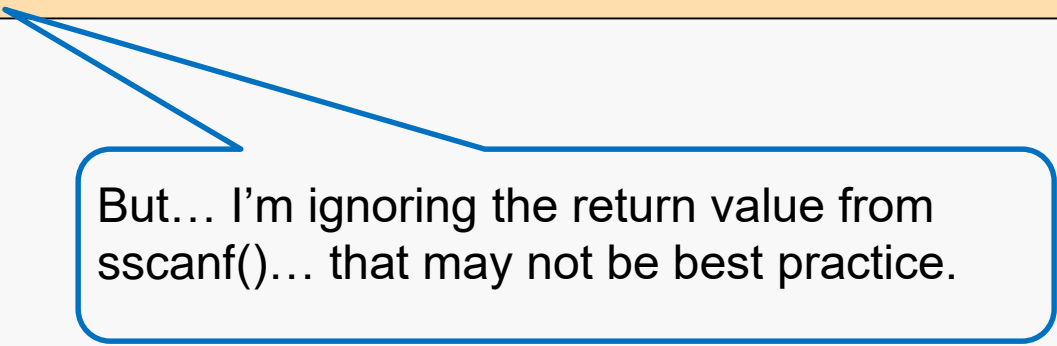
Consider: `add $t0, $s5, $s3`

C provides a number of useful functions here.

Do not ignore `strtok()` ... it's flexible and powerful.

But also, don't ignore the fact that C supports reading formatted I/O:

```
char* array = malloc(MAXLINELENGTH);  
...  
fgets(array, MAXLINELENGTH, source);  
...  
// determine that you read an instruction taking 3 reg's  
...  
sscanf(array, " %s %s, %s, %s", . . .);
```



But... I'm ignoring the return value from `sscanf()`... that may not be best practice.

We may find labels in the data segment and/or the text segment.

```
Str01: .data
       .asciiz "To be or not to be..."
```

data segment

```
main: .text
      la    $t0, Str01
      li    $s0, 4096
      . . .
      add   $s0, $s1, $s2
bgloop:
      lw    $t1, ($t0)
      . . .
      beq   $t0, $t7, bgloop
      . . .
      bne   $t7, $s4, done
      . . .
done:  li    $v0, 10
      syscall
```

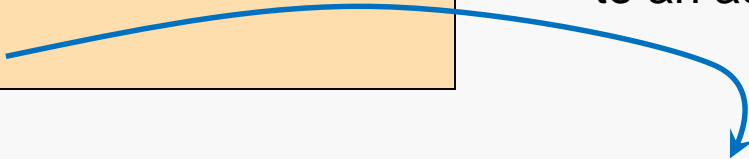
text segment



What's the deal?

```
.data
Str01: .asciiz "To be or not to be..."

.text
main:
    la    $t0, Str01
```



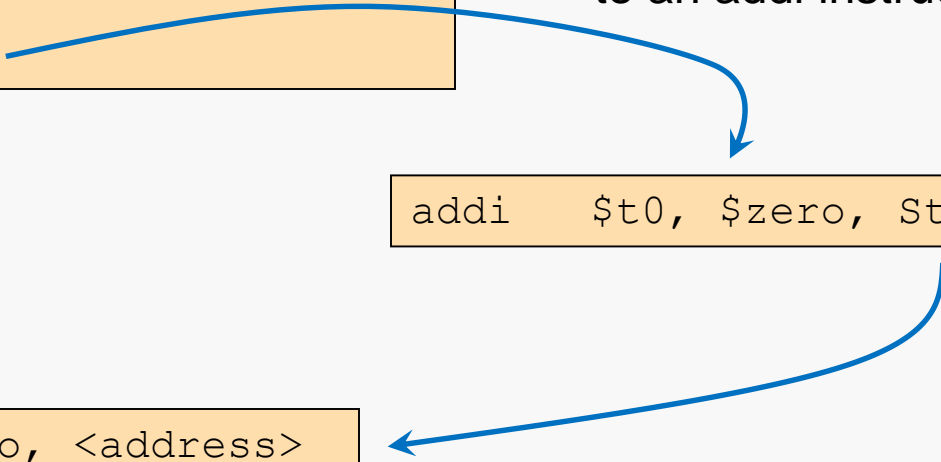
la actually translates  
to an addi instruction

```
addi    $t0, $zero, Str01
```

What's the deal?

```
.data
Str01: .asciiz "To be or not to be..."

.text
main:
    la    $t0, Str01
```



la actually translates  
to an addi instruction

```
addi    $t0, $zero, Str01
```

```
addi    $t0, $zero, <address>
```

Labels translate to 16-bit  
addresses... how?

```

.data
Str01: .asciiz "To be or not to be..."

.text
main:
    la    $t0, Str01
    li    $s0, 4096
    . . .
    
```

data segment

0000 2000

"To be or ..."

text segment

0000 0000

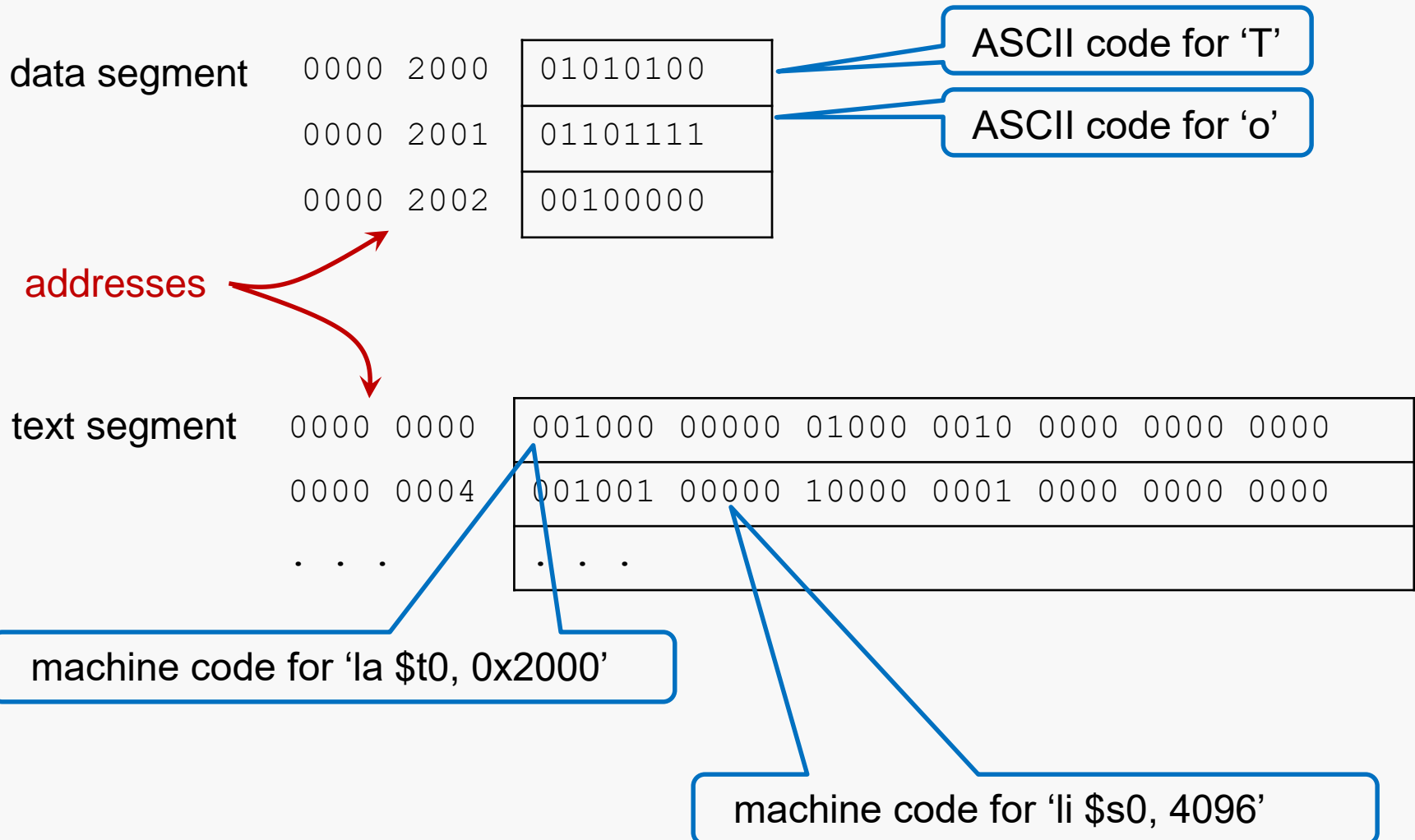
la \$t0, 0x2000

0000 0004

li \$s0, 4096

. . .

. . .



The assembler needs to build a *symbol table*, a table that maps symbolic names (labels) to memory addresses:

|           |        |
|-----------|--------|
| 0000 0000 | main   |
| 0000 001C | bgloop |
| 0000 2000 | Str01  |

Building the symbol table is a bit tricky:

- need to know where data/text segment starts in memory
- may see a label in an instruction before we actually see the label "defined"

One reason most assemblers/compilers make more than one pass.

We want the symbol table to be built before we start translating assembly instructions to machine code — or else we must do some fancy bookkeeping.

Plan your development so that you add features one by one.

This requires thinking about (at least) two things:

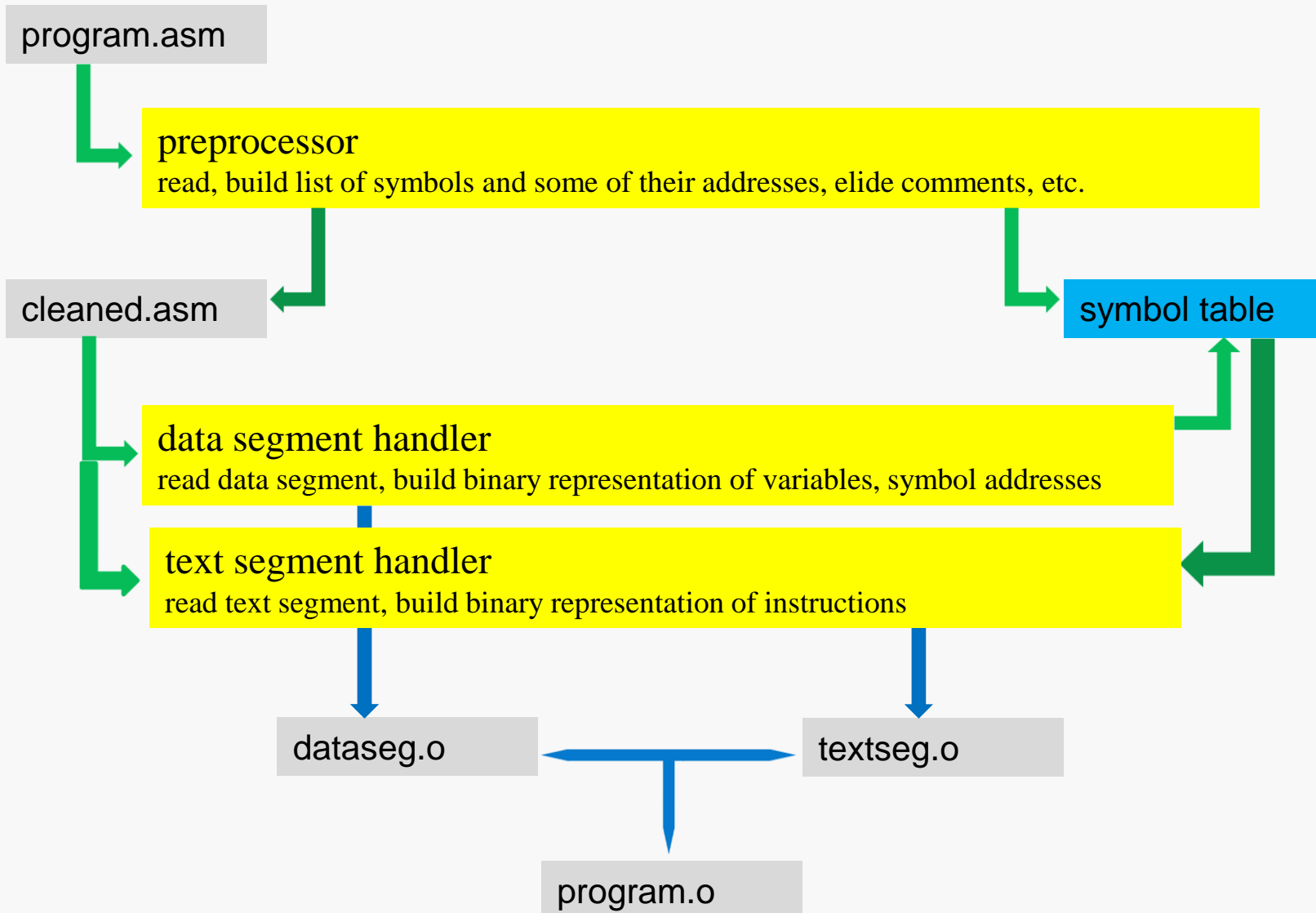
- How can I decompose the system into a sequence of "features" that make sense.

I often start by asking what's the minimum functionality I need to be able to actually do anything? Frequently, that's a matter of data acquisition... so I start with planning how to read my input file.

Now, what can I do next, once I know I can acquire data?

- In what order can I add those "features"?

This is usually not too difficult, provided I've given enough thought to the specification of the system and thought about how I might handle each process that needs to be carried out. But, if I get this wrong, I may have to perform a painful retrofit of something to my existing code.



But this analysis leads to more questions, including (in no particular order):

When/where do we deal with pseudo-instructions?

Some map to one basic instruction, some to two... is that an issue?

What "internal" objects and structures might be valuable in the design/implementation?

Instructions (assembly and machine)?

Build a list of instructions in memory at some point?

How should this be broken up into modules?

What focused, smaller parts might make up a text segment handler?



I will test features as I add them to the system.

NEVER implement a long list of changes and then begin testing. It's much harder to narrow down a logic error.

This may require creating some special test data, often partial versions of the full test data.

For example, I might hardwire a string holding a specific assembly instruction and pass it to my instruction parser/translator module.

Or, I might edit an assembly program so it only contains R-type instructions so I can focus on testing my handling of those.

Take advantage of the diagnostic tools:

`gdb`

- can show you what is really happening at runtime
- which may not be what you believe is happening at runtime
- breakpoints, watchpoints, viewing values of variables

`valgrind`

- can show you where you have memory-usage bugs
- finds memory leaks
- finds memory overruns where you exceed the bounds of a dynamic array

Use the right development environment: CentOS 7 with gcc 4.8.x.

Do that from the beginning; don't wait until the last few days to "port" your code.

Read *Maximizing Your Results* in the project specification.

Small things within your control can make a huge difference in your final score.

There are many of you and few of us, so you cannot expect special treatment.

Use the supplied test harness (shell scripts and test files).

If your packaged submission doesn't run properly with these for you, it won't do that for us either.

There are many of you and few of us, so you will not receive special treatment.

Use the resources...

The course staff is here to help.

Most of us have prior experience with this assignment.