Getting Started

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 Segment

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

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?
Table Lookup

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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$t0</td>
<td>8 or 01000</td>
</tr>
<tr>
<td>$s3</td>
<td>19 or 10011</td>
</tr>
</tbody>
</table>

Designing a Table

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, string).

What are the key values?

What are the values we want to map the keys to?
### Implementing a Table

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

```c
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:

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

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

### Mapping Fields to Bits

Consider: add $t0, $s5, $s3

```
add  $t0, $s5, $s3
```

```
<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>01000</td>
<td>10101</td>
<td>10011</td>
<td>00000</td>
<td>100000</td>
</tr>
</tbody>
</table>
```

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

```c
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.

Bits to Characters: Bit Fiddling

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

```c
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:

```
data
Str01: .asciiz "To be or not to be..."
.text
main:
    la $t0, Str01
    li $s0, 4096
    ...
    add $s0, $s1, $s2
bgloop:
    lw $t1, ($t0)
    ...
    beq $t0, $t7, bgloop
    li $v0, 10
    syscall
data segment
text segment
```

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

So is the logic of translation to 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.
Parsing an Assembly Instruction

Consider: \texttt{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:

\begin{verbatim}
    ... 
    la \$t0, Str01 
    li \$s0, 4096 
    ... 
    add \$s0, \$s1, \$s2 
    bgloop: 
    lw \$t1, (\$t0) 
    ... 
    syscall
\end{verbatim}

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

Parsing an Assembly Instruction

Consider: \texttt{add \$t0, \$s5, \$s3}

C provides a number of useful functions here. Do not ignore \texttt{strtok()}... it's flexible and powerful.

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

\begin{verbatim}
    char* array = malloc(MAXLINELENGTH); 
    ... 
    fgets(array, MAXLINELENGTH, source); 
    ... 
    // determine that you read an instruction 
    ... 
    sscanf(array, " %s %s, %s, %s", ...);
\end{verbatim}
Broader Perspective: Labels

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

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

Broader Perspective: Labels

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
```

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

Assembler Warmup 17

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

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

.text segment
0000 2000 "To be or ..."

.data segment
0000 0000 main
0000 001C bgloop
0000 2000 Str01

Symbol Table

Assembler Warmup 18

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000</td>
<td>main</td>
</tr>
<tr>
<td>0000 001C</td>
<td>bgloop</td>
</tr>
<tr>
<td>0000 2000</td>
<td>Str01</td>
</tr>
</tbody>
</table>

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.
Incremental Development

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.

One View

program.asm

preprocessor
read, build list of symbols and some of their addresses, elide comments, etc.

cleaned.asm

data segment handler
read data segment, build binary representation of variables, symbol addresses

text segment handler
read data segment, build binary representation of instructions

dataseg.o
textseg.o

symbol table

program.o
More Questions

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?

Testing

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.
Tools

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

Pragmatics

Use the right development environment: CentOS 6 with gcc 4.4.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 cannot expect special treatment.

Use the resources…

- The course staff is here to help.
- Most of us have prior experience with this assignment.