CS 241: Foundations of Sequential Programs

Chris Thomson

Winter 2013, University of Waterloo

Notes written from Gordon Cormack's lectures.

1 Introduction & Character Encodings

← January 7, 2013

1.1 Course Structure

The grading scheme is 50% final, 25% midterm, and 25% assignments. There are eleven assignments. Don't worry about any textbook. See the course syllabus for more information.

1.2 Abstraction

Abstraction is the process of removing or hiding irrelevant details. Everything is just a sequence of bits (binary digits). There are two possible values for a bit, and those values can have arbitrary labels such as:

- Up / down.
- Yes / no.
- 1 / 0.
- On / off.
- Pass / fail.

Let's say we have four projector screens, each representing a bit of up/down, depending on if the screen has been pulled down or left up (ignoring states between up and down). These screens are up or down independently. There are sixteen possible combinations:

Screen 1	Screen 2	Screen 3	Screen 4
Up (1)	Down (0)	Up (1)	Down (0)
Down (0)	Down (0)	Down (0)	Up (1)
⋮	⋮	⋮	⋮

Note that there are sixteen combinations because $k = 4$, and there are always 2^k combinations since there are two possible values for each of k screens.

1.3 Endianness

Let's consider the sequence 1010. This sequence of bits has a different interpretation when following different conventions.

- **Unsigned, little-endian**: $(1 \times 2^0) + (0 \times 2^1) + (1 \times 2^2) + (0 \times 2^3) = 1 + 4 = 5$.
- **Unsigned, big-endian**: $(0 \times 2^0) + (1 \times 2^1) + (0 \times 2^2) + (1 \times 2^3) = 2 + 8 = 10$.
- **Two's complement, little-endian**: $5 - 16 = -10$.
- **Two's complement, big-endian**: $10 - 16 = -6$.
- **Computer terminal**: LF (line feed).

Note that a two's complement number n will satisfy $-2^{k-1} \leq n < 2^{k-1}$.

1.4 ASCII

ASCII is a set of meanings for 7-bit sequences.

Bits	ASCII Interpretation
0001010	LF (line feed)
1000111	G
1100111	g
0111000	8

In the latter case, 0111000 represents the character '8', not the unsigned big- or little-endian number 8.

ASCII was invented to communicate text. ASCII can represent characters such as A-Z, a-z, 0-9, and control characters like ();!. Since ASCII uses 7 bits, $2^7 = 128$ characters can be represented with ASCII. As a consequence of that, ASCII is basically only for Roman, unaccented characters, although many people have created their own variations of ASCII with different characters.

1.5 Unicode

Unicode was created to represent more characters. Unicode is represented as a 32-bit binary number, although representing it using 20 bits would also be sufficient. The ASCII characters are the first 128, followed by additional symbols.

A 16-bit representation of Unicode is called **UTF-16**. However, there's a problem: we have *many* symbols ($> 1M$) but only $2^16 = 65,536$ possibilities to represent them. Common characters are represented directly, and there is also a 'see attachment' bit for handling the many other symbols that didn't make the cut to be part of the $65,536$. Similarly, there is an 8-bit representation of Unicode called **UTF-8**, with the ASCII characters followed by additional characters and a 'see attachment' bit.

The bits themselves do not have meaning. Their meaning is in your head – everything is up for interpretation.

1.6 A Message for Aliens

← January 9, 2013

In a computer, meaning is in the eye of the beholder. We must agree on a common interpretation – a convention. However, the English language and numbers also have their meaning determined by a set of conventions.

NASA wanted to be able to leave a message for aliens on a plaque on their spacecraft, however it was clear that aliens would not understand our language or even 0s and 1s. NASA wanted their message to be a list of prime numbers. They decided they would use binary to represent the numbers, but since 0s and 1s would be ambiguous to aliens, they used a dash (-) instead of 0, and 1 for 1. It's only a convention, but it's one that NASA determined aliens would have a higher chance of understanding.

1.7 Hexadecimal

Hexadecimal (hex) is base 16. It has sixteen case-insensitive digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, and f.

Why is hex useful? It makes conversions easy. We group bits into sequences of four:

$$\underbrace{\underbrace{0011}_{3}\underbrace{1010}_{A}}_{3A}$$

Conversions are made especially easy when the sequences of bits are lengthy:

$$\underbrace{\underbrace{10}_{2}\underbrace{1110}_{E}\underbrace{0111}_{7}\underbrace{0011}_{3}\underbrace{1011}_{B}\underbrace{1001}_{9}\underbrace{1000}_{8}\underbrace{0011}_{3}}_{2E73B983}$$

2 Stored Program Computers

Stored program computers are also known as the **Von Neumann architecture**. They group bits into standard-sized sequences.

In modern times, standard-sized sequences of bits are:

- **Bytes**. A byte is 8-bits (256 possible values). Example: 00111010.
- **Words**. A word is only guaranteed to be "more than a byte." Words are often 16-bit ($2^{16} = 65,536$ possible values), 32-bit ($2^{32} \approx 4 \times 10^9$), or 64-bit ($2^{64} \approx 10^{19}$).

2.1 Storage Devices

2.1.1 Registers

There are typically a finite number of fixed-sized sequence of bits, called **registers**. You can put bits in, peek at them, and modify them. A "64-bit CPU" just means it's a CPU that uses 64-bit words.

Calculators typically have 2-3 registers for recalling numbers and maintaining state.

There are a couple of downsides to registers. They're expensive to build, which is why there is a finite number of them. They're also difficult to keep track of.

2.1.2 RAM (Random Access Memory)

RAM is essentially a physical array that has **address lines**, **data lines**, and **control lines**. Data is fed into RAM using electrical lines. Data will remain in RAM until overwritten.

If you want to place a happy face character at address 100, you set the address lines to 100, the data lines to 10001110 (which is the Unicode representation of a happy face), and give the control lines a kick.

RAM could be implemented in several different ways. It could even be created with a **cathode ray tube**. The **core** method is synonymous with RAM, however. It involves a magnetic core, and the data remains magnetized after the magnet is removed. Bits are read by toggling the state (toggling the magnetic poles) and seeing if it was easier to toggle than expected (similar to unlocking an already-unlocked door), and then toggling back after. No one really uses magnetic cores anymore.

Capacitive memory (also known as dynamic RAM or **DRAM**) is still used today. It involves an insulator, and two conductive plates, one more negatively-charged than the other. The electrons will remain in their state even when the poles are removed. There is a problem, however. Insulators are not perfect – electrons will eventually make their way through the insulator. In order to alleviate this, we have to refresh the charge fairly often (every second, for instance).

Switches are typically used only for registers and cache. They produce more heat, but are much faster.

2.2 Control Unit Algorithm

The CPU contains a **control unit**, several **registers**, PC (**program counter**), and IR (**instruction register**), and is connected to RAM with electrical lines.

```
PC <- some fixed value (e.g. 0)
loop
  fetch the word of RAM whose address is in PC, put it in IR
  increment PC
  decode and execute the machine instructon that's in IR
end loop
```

IR would contain an instruction like "add register 1 to register 2, and put the result into register 7."

3 First Steps with MIPS

← January 11, 2013

3.1 Unix

You'll need Unix to use MIPS in this course. Unix was originally created in the 1970s at AT&T Bell Labs. Unix is still popular today, especially for servers. Linux is a Unix dialect, and Mac OS X is also based on Unix.

Unix has three types of files:

- **Binary files**. A sequence of arbitrary bytes.
- **Text files**. A sequence of ASCII characters, with lines terminated by a LF / newline.
- **Tools**. These are programs, which are technically binary files.

3.1.1 Getting Access to a Unix System

If you use Linux or Mac OS X, you're in good shape. However, Windows is not Unix-based, so you'll have to pursue one of these alternative options:

- Install Linux. You can dual-boot it alongside Windows if you'd like, or you could install it inside a virtual machine.
- Install Cygwin. When installing it, choose to install everything.
- Login to the `student.cs` servers remotely. You can use PuTTY for that.

3.1.2 Commands You Should Know

- `ssh username@linux.student.cs.uwaterloo.ca` – logs you into the student.cs systems remotely through SSH.
- `cat unix_text_file.txt` – copies the contents of the file to the current terminal. If a non-text file is given to `cat`, incorrect output will result.
- `xxd -b unix_file.txt` – prints the binary representation of the file to the terminal. The numbers in the left column are the location in the file. If it's a Unix text file, the ASCII representation is presented on the right, with all non-printable characters printed as dots. `xxd` is not aware of newline characters – it arbitrarily splits the file into 16 bytes per line.
- `xxd unix_file.txt` – prints the hex representation of the file to the terminal. Identical to the previous command (`-b`) in every other way.
- `ls -l` – lists all files in the current directory in the long-listing form, which shows the number of bytes in the file, permissions, and more.

3.2 Getting Started with MIPS

The MIPS CPU uses 32-bit words since it's a 32-bit machine, and it's big-endian. You can use `xxd` to inspect MIPS files. MIPS has 32 registers (numbered 0 to 31).

At the end of our MIPS programs, we will copy the contents of register $31 to the program counter (PC) to "return".

3.2.1 Running MIPS Programs

Upon logging in to the `student.cs` servers, run `source ~cs241/setup` in order to add the required executables to your `PATH`. Then, when given a MIPS executable called `eg0.mips`, you can run `java mips.twoints eg0.mips` in order to run the program.

`mips.twoints` is a Java program that requests values for registers $1 and $2 and then runs the given MIPS program.

3.2.2 Creating MIPS Programs

Start with `vi thing.asm` (or use your favorite editor). Inside this file, you'll create an **assembly language file**, which is a textual representation of the binary file you want to create. Each line in this file should be in the form `.word 0xabcdef12` (that is, each line should start with `.word 0x` – the `0x` is a convention that indicates that hex follows). You can add comments onto the end of lines, starting with a semi-colon (Scheme style).

Next, you'll need to convert your assembly language file into a binary file. You can do that by running `java cs241.wordasm < thing.asm > thing.bin`. You can then inspect `thing.bin` with `xxd` in hex, or in binary if you're masochistic.

A few important things you should know for developing MIPS programs:

- $0 is a register that will always contain 0. It's special like that.
- $30 points to memory that could be used as a stack.
- $31 will be copied to the program counter at the end of execution in order to "return".
- You can specify register values using base 10 values or as hex values (if prefixed by `0x`).
- It takes 5-bits to specify a register, since $2^5 = 32$.
- It's convention to call S and T (as indicated in various documentation) **source registers**, and D is the **destination register**.
- MIPS uses two's complement numbers by default, unless specified otherwise.
- Loops and conditionals are accomplished by adding or subtracting from the program counter.

There is a MIPS reference sheet available on the course website that you'll find to be quite useful. It contains the binary representations for all MIPS instructions. Convert the binary into hex and put them into an assembly language file.

3.2.3 A Few Important MIPS Instructions

1. **Load Immediate & Skip** (`lis`): loads word from the program counter. Loads the next word of memory into the D register. You specify a `lis` instruction followed by an arbitrary word next. You need to also skip the appropriate number of bytes by incrementing the program counter.
2. **Set Less Than [Unsigned]** (`slt`): compares S to T. If S $<$ T, 1 is put into the D register, otherwise 0 is put into the D register.
3. **Jump Register** (`jr`): copies S to the program counter.
4. **Jump and Link Register** (`jalr`): assigns the program counter to register 31, then jumps to it.
5. **Branch on Equal** (`beq`): if S is equal to T, it adds the specified number to the program counter (times 4). There is also **Branch on Unequal** (`bne`) which does the opposite.

3.2.4 MIPS Program Workflow

← January 14, 2013

The MIPS CPU understands **binary machine language programs**, however we cannot write them directly. Instead, we write **assembly language programs** in text files. By convention, we name these text files with the extension `.asm`. Assembly language contains instructions like `.word 0x00221820`. We feed the assembly language program into `cs241.wordasm`, which is an **assembler**. An assembler translates assembly language into binary machine code.

Assembly language can also look like this: `add $3, $1, $2`. Assembly language in this form has to be fed into a different assembler (`cs241.binasm`) that understands that flavor of assembly syntax.

There is a MIPS reference manual available on the course website. It might be useful in situations such as:

- When you want to be an assembler yourself. You'll need to lookup the mapping between assembly instructions like `add $3, $1, $2` and their binary equivalents.
- When you need to know what's valid assembly code that an assembler will accept.
- When you want to write your own assembler you'll need a specification of which instructions to handle.

3.2.5 The Format of MIPS Assembly Language

MIPS assembly code is placed into a Unix text file with this general format:

```
labels instruction comment
```

Labels are any identifier followed by a colon. For example, `fred:`, `wilma:`, and `x123:` are some examples of valid labels.

Instructions are in the form `add $3, $1, $2`. Consult the MIPS reference sheet for the syntax of each MIPS instruction.

Comments are placed at the end of lines and must be prefixed by a semicolon. Lines with only comments (still prefixed with a semicolon) are acceptable as well. For example: `; hello world`.

It's important to note that there is a **one-to-one correspondence** between instructions in assembly and instructions in machine code. The same MIPS instructions will always produce the same machine code.

3.2.6 More MIPS Instructions

Here's a more comprehensive overview of the instructions available to you in the CS 241 dialect of MIPS. Note that for all of these instructions, $0 \leq d, s, t \leq 31$, since there are 32 registers in MIPS numbered from 0 to 31.

- `.word`. This isn't really a MIPS instruction in and of itself. Words can be in several different forms. For example:

  - `.word 0x12345678` (hex)
  - `.word 123` (decimal)
  - `.word -1` (negative decimals whose representation will eventually be represented in two's complement)

- `add $d, $s, $t`. Adds $s to $t and stores the result in $d.
- `sub $d, $s, $t`. Subtracts $t from $s and stores the result in $d ($d = $s - $t).
- `mult $s, $t`. Multiplies $s and $t and stores the result in the `HI` and `LO` registers. Uses two's complement.
- `multu $s, $t`. Provides the same functionality as `mult`, but uses unsigned numbers.
- `div $s, $t`. Divides $s by $t. The remainder is stored in `HI` and the quotient is stored in `LO`.
- `divu $s, $t`. Provides the same functionality as `div`, but uses unsigned numbers.
- `mflo $d`. Copies the contents of the `LO` register to $d.
- `mfhi $d`. Copies the contents of the `HI` register to $d.
- `lis $d` (load immediate and skip). Copies the word from the program counter (PC), adds 4 to PC in order to skip the word you just loaded.
- `lw $t, i($s)` (load word, $-32,768 \leq i \leq 32,767$). For example: `lw $3, 100($5)` will get the contents of $5, add 100, treat the result as an address, fetch a word from RAM at that address, and put the result into $3.
- `sw $t, i($s)` (store word, $-32,768 \leq i \leq 32,767$). This works in a similar way to `lw`, except it stores the contents of $t at RAM at this address.
- `slt $d, $s, $t` (set less than). Sets $d to 1 if $s < $t, or to 0 otherwise.
- `sltu $d, $s, $t` (set less than unsigned). Sets $d to 1 if $s < $t, or to 0 otherwise. Interprets the numbers as unsigned numbers.
- `beq $s, $t, i` (branch if equal, $-32,768 \leq i \leq 32,767$). Adds $4i$ to the program counter if $s is equal to $t. Note that 4 is still added (in addition to adding the $4i$ for this specific command) as you move to the next instruction, as with all instructions.
- `bne $s, $t, i` (branch if not equal, $-32,768 \leq i \leq 32,767$). Works the same way as `beq`, except it branches if $s is not equal to $t.
- `jr $s` (jump register). Copies $s to the program counter.
- `jalr $s` (jump and link register). Copies $s to the program counter and copies the previous value of the program counter to $31.

3.2.7 Example Program: Sum from 1 to N

We want a program that sums the numbers from 1 to n, where n is the contents of \$1, and we want the result to be placed in \$3. Aside: it's only a convention that we reserve \$1 and \$2 as registers for input parameters and \$3 as the register for the result – the MIPS system itself does not treat these registers in a special way.

```
; $1 is N.
; $3 is the sum.
; $2 is temporary.

add $3, $0, $0  ; zero accumulator

; beginning of loop
add $3, $3, $1  ; add $1 to $3
lis $2          ; decrement $2
.word -1
add $1, $1, $2
bne $1, $0, -5  ; n = 0? If not, branch to beginning of loop

jr $31          ; return
```

If we enter 10 for \$1 (to get the sum of the numbers from 1 to 10), we should get 55. But the actual result is `0x00000037`. Note that $37_{16} = 55_{10}$, so the program works as expected. The end result is \$1 being `0x00000000` (0_{10}), \$2 being `0xffffffff` (-1_{10}), and \$3 being `0x00000037` (55_{10}).

3.2.8 Housekeeping Notes

- `cs241.binasm` will be available on Thursday after the assignment 1 deadline has passed. You can use this for future assignments as necessary.
- You don't need to memorize the binary representation of MIPS commands for exams, or the ASCII representation of characters. You'll be provided with the MIPS reference sheet and an ASCII conversion chart for the exams.

3.2.9 Labels

← January 16, 2013

Part of the assembler's job is to count instructions and keep track of their locations (`0x00000004`, `0x00000008`, `0x0000000c`, etc.). The assembler can also simplify the programmer's job at with **labels**.

Labels are identifiers in the form `foo:` (a string followed by a colon). A label `foo:` is equated to the **location** of the ine on which it is defined.

Some instructions like `beq` and `bne` rely on relative locations of lines. Counting these yourself is tedious, and can be troublesome in some situations. The locations you specify, both in places where they're specified relatively and in places where they're specified absolutely (`jr`), may become invalid if you add or remove any lines to your codebase.

Labels can be used in place of integer constants. If you have an instruction like `bne $1, $2, -5`, you can replace it with `bne $1, $2, foo`. The assembler will compute:

$$\frac{\text{location(label) - location(next instruction)}}{4}$$

The third argument of `bne` is always a number. It can an integer literal, or it can be a label which will be converted to an integer by the assembler. MIPS itself has no knowledge of labels – only the assembler does.

4 Accessing RAM in MIPS

4.1 RAM vs. Registers

There are some key differences between RAM and registers:

- There is lots of RAM available, but there are a finite number of registers available (usually not very many).
- You can compute addresses with RAM, but registers have fixed names that cannot be computed (i.e. you can compute memory address `0x00000008 = 0x00000004 + 0x0000004`, but you can't compute $2.
- You can create large, rich data structures in RAM. Registers provide small, fixed, fast storage mechanisms.

4.2 Storing in RAM

```
lis $5
.word 100000
sw $1, 0($5)
lw $3, 0($5)
jr $31
```

The example above uses memory address 100000. But how do we know that we have that much RAM? How do we know it's not already being used by someone else? This is clearly a bad practice.

We really shouldn't just use an arbitrary memory address without any type of safety checking. So, we'll reserve some memory ourselves. We can add a word after the last `jr` instruction, which means memory will be allocated for the word instruction, however it'll never be executed.

MIPS requires that we actually specify a word. The contents of it don't matter, so we'll just use `.word 28234`, which is entirely arbitrary. We can then replace 100000 in the above example with 20. For now, we can assume that our MIPS program will always run in memory starting at memory address 0, so memory addresses and locations in our code can be treated as being the same.

But wait! Hard-coding 20 is a bad idea, in case the program changes, and it's tedious to calculate the proper location (20). We should use a label instead.

4.2.1 Stack

$30 is the conventional register to place the **stack pointer** in (sometimes abbreviated as $sp). The stack pointer points to the first address of RAM that's reserved for use by other people. Here's an example of storing and fetching something in the stack:

```
sw $1, -4($30)
lw $3, -4($30)
jr $31
```

All memory with an address less than the value of $30 could be used by your program. You can use this method to create 100,000+ storage locations, and that wouldn't have been possible with registers without having 100,000 registers, and without hard-coding $1, $2, ... $100000.

The stack pointer isn't magical. It doesn't change on its own, but you can change it yourself if you'd like. Just make sure to change the stack pointer back to its original state before you return (before `jr $31`).

Here's another example of a program which sums the numbers from 1 to n without modifying anything except $3. Actually, it's okay to modify $1 and $2, so long as they are returned to their original state before returning.

```
sw $1, -4($30)   ; save on stack
sw $2, -8($30)   ; save on stack

lis $2
.word 8
sub $30, $30, $2 ; push two words

add $3, $0, $0

; beginning of loop
foo: add $3, $3, $1
  lis $2
  .word -1
  add $1, $1, $2
  bne $1, $0, foo

lis $2
.word 8
add $30, $30, $2 ; restore stack pointer

lw $1, -4($30)   ; restore from stack
lw $2, -8($30)

jr $31
```

`mips.array` is a MIPS runner that passes an array A of size N into your MIPS program. The address of A will be in $1, and the size of A (which is N) will be in $2.

To access array elements, you would execute instructions such as these:

```
lw $3, 0($1)
sw $4, 4($1)
```

Note that each array index increases by 4.

You can also compute the array index. In C/C++, you might have an expression A[i]. A is in \$1 and i is in \$3. How can we fetch A[i] into x (let's say, into \$7)?

1. Multiply i by 4.
2. Add to A.
3. Fetch RAM at the resulting address.

```
add $3, $3, $3
add $3, $3, $3 ; these two lines give i * 4

add $3, $3, $1 ; A + i * 4
lw $7, 0($3)
```

Note that the two first lines each double the value in \$3, so the two lines together effectively multiplied i by 4.

Here's an example program to sum the integers in an array A of length N. \$1 contains the address of A, \$2 contains N, and \$3 will contain the output (the sum). \$4 is used temporarily.

```
add $3, $0, $0

loop:
  lw $5, 0($1)     ; fetch A[i]
  add $3, $3, $5   ; add A[i] to sum
  lis $4           ; load -1 into $4
  .word -1
  add $2, $2, $4   ; decrement $2
  lis $4
  .word 4
  add $1, $1, $4
  bne $2, $0, loop ; loop if not done.

jr $31
```

5 Procedures in MIPS

← January 18, 2013

Recall the `sum.asm` program from earlier, which sums the numbers from 1 to N (\$1), and puts the result in \$3:

```
sum:              ; only needed for next example, sum10.asm
  add $3, $0, $0 ; $3 is the accumulator A.

loop:
  add $3, $3, $1 ; A = A + N
  lis $2
  .word -1
  add $1, $1, $2 ; A = A + (-1)
  bne $1, $0, loop

jr $31
```

Now, let's create a program `sum10.asm` which sums the numbers from 1 to 10, and puts the result in $3. We'll add the `sum:` label to the top of our `sum.asm` file, as indicated, so we have a way to jump to the `sum.asm` line (which is part of how procedures are called in MIPS).

```
; PROLOGUE
sw $31, -4($30)  ; push word onto stack
lis $2
.word 4
sub $30, $30, $2

; PROCEDURE CALL
lis $1
.word 10
lis $4
.word sum        ; address of sum procedure is in $4
jalr $4          ; puts old PC value into $31, jumps to $4

; EPILOGUE
lis $2
.word 4
add $30, $30, $2 ; restore stack pointer
lw $31, -4($30)  ; restore $31
jr $31
```

Note that if you ever get into an infinite loop while executing a MIPS program, you can push CTRL-C to forcefully end the process immediately.

We use `jalr` instead of `jr` so the `sum` routine knows how to get back. `jalr` is the only instruction that can access the contents of the `PC`.

How do we actually run this program? We `cat` together the two programs! It really is that simple. You execute `cat sum10.asm sum.asm | java cs241.binasm > foo.mips` to get a MIPS program in binary.

5.1 Recursion

Recursion is nothing special. You need to save any local variables (which are stored in registers), including given parameters and the return address, onto the stack so we can change

them back when we're done. We don't want subroutines (recursive calls) to mess with those values, so subroutines must preserve their own values. "It's always good hygiene to save your registers."

Let's build `gcd.asm`, where $1 = a, $2 = b, and $3 will hold the result. We will use the following algorithm:

$$gcd(a,b) = \begin{cases} b & a = 0 \\ gcd(b\%a, a) & a \neq 0 \end{cases}$$

Here's `gcd.asm`:

```
gcd:
  sw $31, -4($30)  ; save return address
  sw $1, -8($30)   ; and parameters
  sw $2, -12($30)
  lis $4
  .word 12
  sub $30, $30, $4

  add $3, $2, $0   ; tentatively, result = b
  beq $1, $0, done ; quit if a = 0
  div $2, $1       ; stores quotient in LO, remainder in HI
  add $2, $1, $0   ; copy a to $2
  mfhi $1          ; $1 <- b % a
  lis $4
  .word gcd
  jalr $4

done:
  lis $4
  .word 12
  add $30, $30, $4
  lw $31, -4($30)
  lw $31, -8($30)
  lw $2, -12($30)
  jr $31
```

An **invariant** means if something is true as a pre-condition then it is always true as a post-condition.

Notice in the `gcd.asm` example, you aren't actually erasing the stack contents. If you're storing secret data, you should overwrite it with zeroes, or (ideally) garbage data. For assignment 2, at least, we can just leave our garbage lying around.

5.2 Input and Output

`getchar` and `putchar` simulate RAM, however they actually send the data to/from the user's keyboard/monitor. `getchar` is located at memory address `0xffff0004` and `putchar`

is at address `0xffff000c`. If you store or load a byte at either of these addresses, you will send or retrieve the byte to/from standard input (STDIN) or standard output (STDOUT).

We will create an example program, `cat.asm`, to copy input to output:

```
lis $1
.word 0xffff0004   ; address of setchar()
lis $3
.word -1           ; EOF signal

loop:
  lw $2, 0($1)     ; $2 = getchar()
  beq $2, $3, quit ; if $2 == EOF, then quit
  sw $2, 8($1)     ; putchar() since getchar() and putchar() are 8 apart
  beq $0, $0, loop

quit: jr $31
```