Compiled on 07/12/2022 at 1:17pm

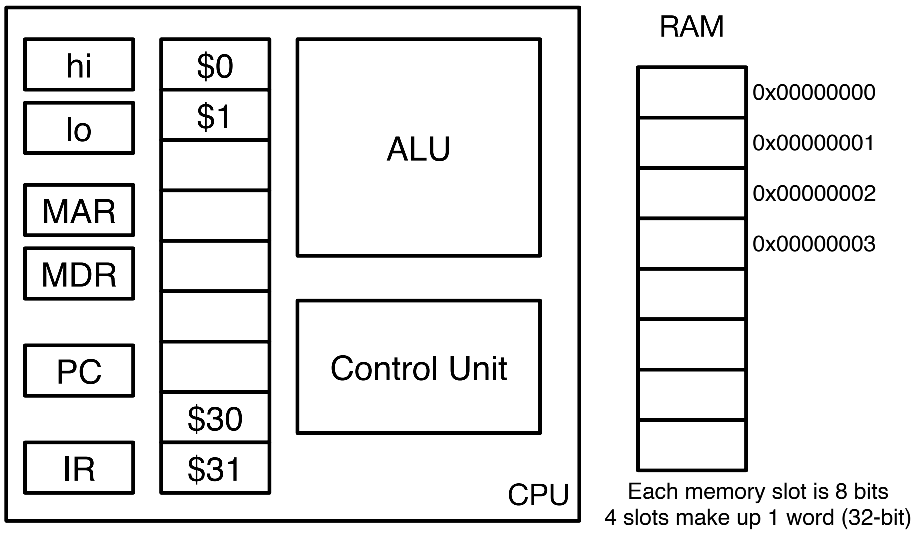
# Machine and Assembly Language

At some point, you have likely heard someone say that "computers only understand binary". While true, computers do not just understand any arbitrary sequence of 1s and 0s. Computer processors know how to interpret specific sequences of 1s and 0s. Different processors understand different patterns. Patterns that a particular processor understands are called a **machine language**, i.e., it is the language we must use to speak to the machine. A machine language is made up of multiple patterns. Each pattern is called a **machine instruction**.

There are many machine languages, some more popular than others. In CS241, we use a machine language named MIPS (Microprocessor Without Interlocked Pipeline Stages). At some point, a long time ago, MIPS was quite popular. Its uses included CISCO routers and game consoles such as Nintendo 64, PlayStation, PlayStation 2 and PSP. In recent years, MIPS' popularity has decreased and other languages, such as ARM and x86, have become more popular. Despite this, we continue to use MIPS in CS241 mostly because of its simplicity (while switching to ARM would be relatively straightforward there would be a lot more special cases to handle with no real benefit). While the particular instructions in MIPS are specific to MIPS, the concepts and general techniques are widely applicable; after learning MIPS, you will find it quite easy to learn, for instance, ARM.

## 1 MIPS Hardware

We use a 32-bit version of MIPS (while a 64-bit version does exist, the underlying theory is the same). This means that the machine instructions will all be 32 bits in length. The following is a pictorial representation of the simplified MIPS processor and RAM (Random Access Memory) that we will discuss in the course.



**Control Unit:** The Control Unit is the brain of the processor. It controls the sequence of instructions to execute (discussed later), interprets the instructions and the flow of data, contains the processor's timing unit, and controls signals to and from peripheral devices (not discussed in the course).

**Arithmetic Logic Unit (ALU):** The ALU performs mathematical operations such as addition, multiplication, logical comparison and decisions requested by the control unit.

**General Purpose Registers:** Our MIPS processor has 32 so-called "general purpose registers" (labelled \$0 through \$31 in the diagram). Each register holds 32 bits. While these registers are labeled for general use, we will use some of them to hold specific values at all times, and some of them have special uses dictated by MIPS. In particular, \$0 will always hold the value zero (attempts to modify \$0 are ignored by MIPS CPUs), \$29 will be used as a frame pointer (later in the course), \$30 as the stack pointer and \$31 will store the return address. The need for these will be discussed later.

**Program Counter (PC) and Instruction Register (IR):** PC and IR are two special 32-bit registers used by the Control Unit in fetching and decoding instructions.

**Memory Data Register (MDR) and Memory Address Register (MAR):** Data travels to and from registers and memory (RAM as a simplification) through the data bus. Very briefly, when the Control Unit wants to read from a memory address, it stores the address in the MAR. Data is fetched from that address and placed into the MDR from where it is loaded into the appropriate register.

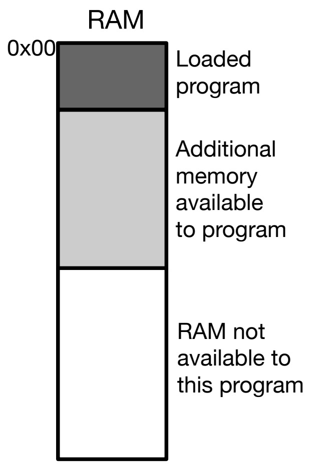
**hi and lo Registers:** These are two special registers used by the multiplication and division machine instructions (discussed later in this module).

**Off-chip memory (RAM):** The components discussed above are all physically part of our simplified MIPS processor. As you have likely noticed, the processor has access to very limited on-chip memory (in the form of registers). Since registers are physically on the processor, they are incredibly fast, but physical space on the processor comes at a premium. Since programs have bigger memory needs, a real-life processor must therefore also communicate with external memory modules that can hold more data but are slower to access. These include L1/L2 cache, RAM, hard drives, etc. Our simplified processor only communicates with the RAM (shown separately in the diagram above). We will view the RAM as an array of 8-bit (1 byte) slots with addresses starting from 0 at the top and increasing as we go down the array. Since our processor is 32-bits, 4 RAM addresses comprise a single word. We will use hexadecimal representation for memory addresses.

## 1.1 How MIPS Executes Programs

Programs operate on data but can also themselves be viewed as data. This is the classic **von Neumann architecture** that you might already be familiar with, at least at a high level from CS136; programs live in the same space within the RAM as the data they operate on. One question to ask is, how does the processor know which part of memory contains the program and which part contains the data? The answer is that it doesn't! This makes it important to ensure that when the processor wants to execute the next line of the program, it knows where that line of code is. The following discusses a simplified algorithm through which a MIPS processor executes a machine language program.

To run a program, the program (sitting on the hard drive) has to be loaded into memory. The operating system uses a special program called the `Loader` to do this. For simplicity, we assume that the program is loaded at the start of memory, i.e., starting address is `0x00`. We will remove this simplification (much) later. The diagram on the right shows what the memory looks like once a program has been loaded by our simplified `Loader`. Notice that the loaded program (shaded dark grey) sits at memory starting at address `0x00`. Some additional memory (shaded light grey) is also made available to the program for use. The rest of memory (not shaded) is not made available to the program. The loader must also do a number of other things to set up the other sections of memory for the program to use. One such task is that the loader communicates to the processor where it has loaded the program. It does so by setting the Program Counter register (PC for short) to the starting address of the program (`0x00` for now).


RAM
0x00
Loaded program
Additional memory available to program
RAM not available to this program

Once the loader is done, a hardware-based algorithm known as the **Fetch-Execute Cycle** begins to run. In pseudo-code, the algorithm behaves as follows:

**Algorithm 1** Fetch-Execute Cycle

```
1: PC = 0x00
2: while true do
3:   IR = MEM[PC]
4:   PC += 4
5:   Decode and execute instruction in IR
6: end while
```

The algorithm is a cycle which goes on forever unless an instruction is executed that breaks out of the loop. Within the loop, on line 3, the algorithm reads the machine instruction that starts at memory address PC and stores it into the Instruction Register (IR for short). Recall that we are using a 32-bit architecture and all registers and machine instructions are 32 bits. Recall also that each address of memory is just 8 bits. Therefore, when an instruction is read from an address (PC in this case), the four bytes for the instruction are actually at PC, PC+1, PC+2 and PC+3. The next instruction to read will be at PC+4. Line 4, therefore increments PC by 4. Once an instruction has been read into IR, the processor decodes the instruction and executes it. Details of how this happens are discussed in CS251.

The order in which these steps happen is quite important. Note in particular that PC is updated *before* the instruction is actually executed; this affects the meaning of instructions that themselves interact with PC!

## 2 MIPS Machine Language

Let us recap what we know about our machine instructions so far. They are all 32 bits and must follow a certain pattern so that the processor can decode and execute the instruction. We use the following simplified set of 18 instructions[1] in our MIPS machine language:

**Basic Instruction Formats**

| | | | | |
|---|---|---|---|---|
| | Register | `0000 00ss ssst tttt dddd d000 00ff ffff` | R | s, t, d are interpreted as unsigned |
| | Immediate | `oooo ooss ssst tttt iiii iiii iiii iiii` | I | i is interpretted as two's complement |

**Instructions**

| Name | Assembly Syntax | Machine Syntax | | Semantics/behaviour |
|---|---|---|---|---|
| Word | .word i | `iiii iiii iiii iiii iiii iiii iiii iiii` | | |
| Add | add $d, $s, $t | `0000 00ss ssst tttt dddd d000 0010 0000` | R | `$d = $s + $t` |
| Subtract | sub $d, $s, $t | `0000 00ss ssst tttt dddd d000 0010 0010` | R | `$d = $s - $t` |
| Multiply | mult $s, $t | `0000 00ss ssst tttt 0000 0000 0001 1000` | R | `hi:lo = $s * $t` |
| Multiply Unsigned | multu $s, $t | `0000 00ss ssst tttt 0000 0000 0001 1001` | R | `hi:lo = $s * $t` |
| Divide | div $s, $t | `0000 00ss ssst tttt 0000 0000 0001 1010` | R | `lo = $s / $t; hi = $s % $t` |
| Divide Unsigned | divu $s, $t | `0000 00ss ssst tttt 0000 0000 0001 1011` | R | `lo = $s / $t; hi = $s % $t` |
| Move From High/Remainder | mfhi $d | `0000 0000 0000 0000 dddd d000 0001 0000` | R | `$d = hi` |
| Move From Low/Quotient | mflo $d | `0000 0000 0000 0000 dddd d000 0001 0010` | R | `$d = lo` |
| Load Immediate And Skip | lis $d | `0000 0000 0000 0000 dddd d000 0001 0100` | R | `$d = MEM[pc]; pc = pc + 4` |
| Load Word | lw $t, i($s) | `1000 11ss ssst tttt iiii iiii iiii iiii` | I | `$t = MEM [$s + i]:4` |
| Store Word | sw $t, i($s) | `1010 11ss ssst tttt iiii iiii iiii iiii` | I | `MEM [$s + i]:4 = $t` |
| Set Less Than | slt $d, $s, $t | `0000 00ss ssst tttt dddd d000 0010 1010` | R | `$d = 1 if $s < $t; 0 otherwise` |
| Set Less Than Unsigned | sltu $d, $s, $t | `0000 00ss ssst tttt dddd d000 0010 1011` | R | `$d = 1 if $s < $t; 0 otherwise` |
| Branch On Equal | beq $s, $t, i | `0001 00ss ssst tttt iiii iiii iiii iiii` | I | `if ($s == $t) pc += i * 4` |
| Branch On Not Equal | bne $s, $t, i | `0001 01ss ssst tttt iiii iiii iiii iiii` | I | `if ($s != $t) pc += i * 4` |
| Jump Register | jr $s | `0000 00ss sss0 0000 0000 0000 0000 1000` | R | `pc = $s` |
| Jump And Link Register | jalr $s | `0000 00ss sss0 0000 0000 0000 0000 1001` | R | `temp = $s; $31 = pc; pc = temp` |

In the next sections we discuss each of these instructions in detail.

### 2.1 Add and Subtract Instructions

The add instruction takes the values stored in two registers (let's call them \$s and \$t) and stores the result in another register, \$d. In other words, the behaviour is \$d = \$s + \$t. We will use the following syntax: `add $d, $s, $t`. Referring to the MIPS reference sheet, we see that the 32-bit pattern for the add instruction looks as follows:

```
add $d, $s, $t:  000000 sssss ttttt ddddd 00000 100000
```

Notice that we need 5 bits to represent a register number, since there are 32 general purpose registers, numbered 0 to 31 ($2^5$ is 32). We formatted the above by adding space and colour in between the different components of the machine instruction for convenience. An alternate formatting we will often use is to separate each nibble, as this makes the relationship with hexadecimal clearer.

```
add $d, $s, $t:  0000 00ss ssst tttt dddd d000 0010 0000
```

**Example:** Write the MIPS machine instruction that adds the values in register \$5 and \$7 and stores the result in register \$3.

[1] This actually isn't *very* simplified from MIPS, which has only 32 basic instructions and a further 24 variants of those basic instructions. Our version of MIPS is simplified because you'll be implementing various facets of it, and having many similar instructions makes that process more tedious without being any more instructive.

We want to perform `$3 = $5 + $7`, which in our preferred syntax is `add $3, $5, $7`. Therefore, `d` is 3 (00011 in binary), `s` is 5 (00101 in binary) and `t` is 7 (00111 in binary).

Solution: `add $3, $5, $7`: `000000 00101 00111 00011 00000 100000`

Note the difference in the order in which we specify the registers in our preferred syntax and the order the corresponding bits appear in the machine instruction.

Spacing based on nibbles, the instruction is `0000 0000 1010 0111 0001 1000 0010 0000` which in the compact hexadecimal representation is `0x00A71820`.

Write, in hexadecimal, the machine instruction for `add $1, $2, $3`.[2]

Write, in hexadecimal, an instruction which copies the value stored in $3 to register $8. (Hint: recall that $0 always contains the value 0).[3]

Write, in hexadecimal, an instruction which clears register 4, i.e., set the value in $4 to 0.[4]

The subtract instruction is similar in syntax to the add instruction, so we cover it briefly. Given values in $s and $t, `sub $d, $s, $t`, will subtract the value in $t from the value in $s and store the result in $d, i.e., after the processor is done executing the instruction, $d = $s - $t.

From the reference sheet:
`sub $d, $s, $t:  000000 sssss ttttt ddddd 00000 100010`

Note that at the bit level, the only difference between an add and a subtract instruction is the second to last bit.

Remember that Two's complement is designed so that addition and subtraction behave the same with unsigned or signed numbers. As such, there is no need for separate versions of `add` or `sub` for signed and unsigned values; `add` and `sub` are sufficient for either.

How would you negate the value in $3, putting the result back in $3?[5]

[2]In binary `add $1, $2, $3` is `000000 00010 00011 00001 00000 100000` which, formatted in nibbles, gives us `0000 0000 0100 0011 0000 1000 0010 0000`. This is `0x00430820`.

[3]We can copy the value in $3 to $8 by adding the value in $3 to 0. We will encode the instruction `add $8, $3, $0`. This gives us `000000 00011 00000 01000 00000 100000` or formatted in nibbles `0000 0000 0110 0000 0100 0000 0010 0000`. In hexadecimal representation this is `0x00604020`.

[4]One way to clear the value in a register is by storing in it the result of adding 0 to 0. We will encode `add $4, $0, $0`. This gives us `000000 00000 00000 00100 00000 100000` which, in nibble format, is `0000 0000 0000 0000 0010 0000 0010 0000`. This is `0x00002020`.

[5]To negate a value simply subtract that value from 0: `sub $3, $0, $3`.

## 2.2 Jump Instruction

The jump instruction is one of a number of instructions in MIPS that can change control flow in a MIPS program. That is, the instruction executed after a jump will likely not be the instruction that appears sequentially after the instruction (unless the instruction happens to jump to that instruction). The jump instruction must be provided the jump address in register \$s. The semantics, i.e., the behaviour, is that it changes PC to the value in \$s. This means that the next time the fetch-execute cycle fetches an instruction from PC, it will fetch the instruction at the address that was stored in \$s, since the jump instruction changed PC to this value. We will write the jump instruction as `jr $s`.

An interesting observation is worth making: the bits in a register can be interpreted in different ways. The `add` and `sub` instruction interpret the bits in the specified registers as representing signed integers, whereas the `jr` instruction interprets the bits in \$s as an address. This is different from what we are used to in higher-level languages, where a variable has a specific type (e.g., `int` vs. `int *`) and only values of that type can be stored in the variable. Registers simply store 32 bits; it is up to the program to decide what the meaning of those bits is.

One particular use of the jump instruction is to exit a MIPS program. Recall that the fetch-execute cycle executes forever unless we break the cycle. The question becomes, how do we jump out of a program? And the answer, once again, is the loader (discussed earlier). During the loading process, the loader stores into \$31 a `return address`. This is the address a program must jump to, to exit the program and return control back to the loader. This is similar in nature to return addresses for C/C++ programs; the stack frame for each function stores a `return address` to the caller of that function. The stack frame for the `main` function stores the return address to exit the program. In MIPS programs, when a program begins, \$31 contains the return address to exit the MIPS program. This means that to exit a program, the program must execute `jr $31`.

This is an important point, so it is worth repeating: **unlike most high-level programming languages, where a program exits when the last instruction is executed (even if the last instruction is not a return), MIPS programs are expected to have an explicit jump to the return address stored by the loader to exit the program.**

Looking at the MIPS reference sheet:

`jr $s: 000000 sssss 00000 00000 00000 001000`

`jr $31: 0000 0011 1110 0000 0000 0000 0000 1000`. In hexadecimal: `0x03e00008`

Earlier, we mentioned that while programs operate on data, they themselves are data as well. Now that we have an intro to machine code, we can discuss this in more detail. Take a look at the video Code really is data!.

## 2.3 Load Immediate and Skip Instruction

**Example:** Write a program that adds integers 11 and 13 and stores the result in $3.

The example looks simple enough: we can use the `add` instruction to add the integers 11 and 13 with the destination register set to $3. The problem is, we haven't learned how to load constant values (often called `immediates`) into a register. Unlike some flavours of MIPS, our MIPS language does not have the ability to add an immediate to a value in a register. Instead, we use a non-standard instruction, load immediate and skip (`lis`). From the reference sheet:

```
lis $d: 000000 00000 00000 ddddd 00000 010100
```

The instruction treats the value appearing in memory immediately after this instruction as data (an `immediate`) and places this value into $d. It then increments PC by 4 (the `skip` part of the instruction). Recall from the Fetch-Execute cycle that after fetching an instruction (and before executing it), PC has already been updated. Therefore, PC contains the address of the next word in memory, which means loading the value that is sequentially right after the `lis` requires reading the word from MEM[PC]. In other words, the instruction does two things: $d = MEM[PC] and then PC += 4.

Per the above, we must place the immediate (the value to be loaded into $d) right after the `lis` instruction itself in memory. In our shorthand, when we want to include binary data that is not meant to represent an instruction, we will use the notation `.word i` (i for `immediate`). We specify i using decimal or hexadecimal, but it is important to remember that in the actual machine language program, all values are in binary. As per the reference sheet, i is encoded as a 32-bit value:

```
.word i: iiii iiii iiii iiii iiii iiii iiii iiii
```

**Example:** Write a program that adds integers 11 and 13 and stores the result in $3.

| Shorthand: | RAM address | Machine instructions (spaced for convenience) | Hexadecimal |
|---|---|---|---|
| `lis $8` | 0x00000000 | 0000 0000 0000 0000 0100 0000 0001 0100 | 0x00004014 |
| `.word 11` | 0x00000004 | 0000 0000 0000 0000 0000 0000 0000 1011 | 0x0000000B |
| `lis $9` | 0x00000008 | 0000 0000 0000 0000 0100 1000 0001 0100 | 0x00004814 |
| `.word 0xD` | 0x0000000c | 0000 0000 0000 0000 0000 0000 0000 1101 | 0x0000000D |
| `add $3, $8, $9` | 0x00000010 | 0000 0001 0000 1001 0001 1000 0010 0000 | 0x01091820 |
| `jr $31` | 0x00000014 | 0000 0011 1110 0000 0000 0000 0000 1000 | 0x03E00008 |

A short video describing how The Fetch-execute cycle operates on this code is available.

CS241 uses a large number of tools to give you practical experience with the course content. These tools work in the CSCF teaching environment so be sure to be logged in to your Linux account. To gain access to the tools, add the following line to your `.bashrc` or `.profile` file found in your home directory (or simply run it manually after logging in).

```
source /u/cs241/setup
```

**Tool: cs241.wordasm**
This is a program that reads the hexadecimal representation of 32-bit words from standard input and produces, on standard output, the **binary** representation of the 4 bytes for each word. Note that this is more than just a convenience. Even if we wanted to, we could not just open a text editor and start typing in 1s and 0s and expect this to become a binary file. Text editors are exactly that, they edit text. When you type in characters in a text editor and save it, the editor saves each keystroke using the ASCII value for that keystroke. So, when you type 1, the 8-bit ASCII value of 1 is stored. But we do not want that! We just want the single bit 1, not an 8-bit ASCII code.

Another interesting thing to note about `cs241.wordasm` is that it does not restrict you to machine instructions. It doesn't care what the 32-bit word is, it simply converts it to binary. However, our primary use for the program will be to convert our machine instructions (written in hexadecimal representation) into binary.

Input to `cs241.wordasm` has a very simple format. Each 32-bit hexadecimal word must appear on its own line and must be prefixed with the string `.word`, a space and then the hexadecimal value using the `0x` notation. You may use a semicolon to enter single line comments.

```
$cat cs241.hex
.word 0x43533234  ; C(43) S(53) 2(32) 4(34)
.word 0x3120726f  ; 1(31) space(20) r(72) o(6f)
.word 0x636b730a  ; c(63) k(6b) s(73) newline(0a)

$cs241.wordasm < cs241.hex
CS241 rocks
```

Recall that `cs241.wordasm` produces output on standard output. An interesting thing to note is that the shell produces the strings `CS241 rocks`. The reason being that the shell is interpreting the binary output generated by the tool as ASCII values and displaying the appropriate ASCII characters.

We can of course redirect the output produced into a binary file. Linux comes with a tool `xxd` (`man xxd` on your terminal for details) that can be used to inspect the binary file.

```
$cs241.wordasm < cs241.hex > cs241.bin
$xxd < cs241.bin
0000000: 4353 3234 3120 726f 636b 730a            CS241 rocks.

$xxd -b < cs241.bin  (the -b shows actual bits)
00000000: 01000011 01010011 00110010 00110100 00110001 00100000  CS241
00000006: 01110010 01101111 01100011 01101011 01110011 00001010  rocks.

$xxd -b -c4 < cs241.bin (-c4 to show 4 columns i.e. 4 bytes, a word, in one row)
00000000: 01000011 01010011 00110010 00110100  CS24
00000004: 00110001 00100000 01110010 01101111  1 ro
00000008: 01100011 01101011 01110011 00001010  cks.
```

**Tool: cs241.binview, a binary viewer**
While `xxd` is a useful tool that you can rely on in any Linux environment, its output format and command line options are not tailored to our needs. Therefore, we have our own tool, `cs241.binview`, for viewing binary files. The tool provides similar functionality to `xxd` but uses a different output format that is more tailored to this course.

The `cs241.binview` tool will produce binary output by default, with one 32-bit word per line. For example, consider the following MIPS program:

```
add $1, $2, $3
jr $31
```

Suppose you translate this into hexadecimal and assemble it with `cs241.wordasm`, producing a machine code file called `simple.bin`. You can view the binary form of this machine code as follows:

```
$ cs241.binview simple.bin
00000000 01000011 00001000 00100000
00000011 11100000 00000000 00001000
```

The tool also supports other command line options to display the bytes in the file as decimal, hexadecimal and ASCII. You can use the command `cs241.binview --help` to get a message describing all available options. Combinations of options are also possible. For example, the combination `-ba` will display the bytes in both binary and ASCII (b and a). Similarly, the combination `-bh` can be used to view the bytes in binary and hex:

```
$ cs241.binview -bh simple.bin
0x00     0x43     0x08     0x20
00000000 01000011 00001000 00100000

0x03     0xE0     0x00     0x08
00000011 11100000 00000000 00001000
```

You can use `--all` (or -A for short) to see the bytes as binary, decimal, hex and ASCII. Of course, the ASCII representation is mostly weird non-printable characters, since this is machine code. Decimal numbers are prefixed with # and hex numbers are prefixed with 0x.

```
$ cs241.binview --all simple.bin
#0       #67      #8       #32
0x00     0x43     0x08     0x20
^NUL     C        ^BS      (SP)
00000000 01000011 00001000 00100000

#3       #224     #0       #8
0x03     0xE0     0x00     0x08
^ETX     \xE0     ^NUL     ^BS
00000011 11100000 00000000 00001000
```

**Tool: mips.twoints, a MIPS processor simulator**
Once we write a MIPS machine language program, we can run this program on a MIPS processor. Now we could go out and buy a MIPS processor or we can use a simulator. `mips.twoints` is one of two simulators we use in CS241. It takes a filename as a **command line argument** that must contain a binary MIPS machine language program. The tool will simulate the execution of the program as if it was running on a MIPS processor and, once the program ends, print the output and values of all registers. Some important notes on running the tool:

- The tool simulates a loader that has loaded the program starting at address `0x00`.
- Before running the program, the tool requires input of two integers (from standard input) that it will store in $1 and $2 (hence the name `twoints`).

```
$ cat Add11And13.hex
.word 0x00004014 ; lis $8
.word 0x0000000B ; .word 11
.word 0x00004814 ; lis $9
.word 0x0000000D ; .word 13
.word 0x01091820 ; add $3, $8, $9
.word 0x03E00008 ; jr $31
$ cs241.wordasm < Add11And13.hex > output.mips
```

The file `Add11And13.hex` contains the program we discussed earlier in this section. We have written this in a format suitable for `cs241.wordasm`. The file `output.mips` contains this program in MIPS machine language. We can provide this file as input to `mips.twoints`.

```
$ mips.twoints output.mips
Enter value for register 1: 1
Enter value for register 2: 42
Running MIPS program.
MIPS program completed normally.
$01 = 0x00000001   $02 = 0x0000002a   $03 = 0x00000018   $04 = 0x00000000
$05 = 0x00000000   $06 = 0x00000000   $07 = 0x00000000   $08 = 0x0000000b
$09 = 0x0000000d   $10 = 0x00000000   $11 = 0x00000000   $12 = 0x00000000
$13 = 0x00000000   $14 = 0x00000000   $15 = 0x00000000   $16 = 0x00000000
$17 = 0x00000000   $18 = 0x00000000   $19 = 0x00000000   $20 = 0x00000000
$21 = 0x00000000   $22 = 0x00000000   $23 = 0x00000000   $24 = 0x00000000
$25 = 0x00000000   $26 = 0x00000000   $27 = 0x00000000   $28 = 0x00000000
$29 = 0x00000000   $30 = 0x01000000   $31 = 0x8123456c
```

- When we ran `mips.twoints` it requested input for register 1 and 2. We arbitrarily gave it the integer values 1 and 42, respectively.
- $1 contains 1 and $2 contains `0x2a` which is 42 in hexadecimal.
- $8 contains 11 and $9 contains 13 (since the program loaded these)
- $3 contains `0x18` which is 24. This is the result of adding 11 and 13.

Note that in this example our program did not actually need to use the values in registers 1 and 2. However, most useful programs will want some kind of input. The `mips.twoints` is hard-coded to always require two inputs.

We discuss two other tools `cs241.binasm` and `mips.array`, later in this module.

# 3 MIPS Assembly Language

Programmers quickly realized that programming in machine language is very tedious and prone to errors. The general trend in programming has been to develop abstractions over low-level programming languages and rely on software to convert a higher-level abstraction to a lower level.

While the MIPS processor can only execute MIPS Machine Language, that does not mean that we must write the 1s and 0s that the processor will eventually process. MIPS Assembly Language is an abstraction over MIPS Machine Language.

**Definition 1** *Assembly language: a textual representation of a machine language*

Unlike high-level languages (C or Java), assembly code is still very close to machine code. In fact, there is a one-to-one correspondence between an assembly instruction and a machine instruction. A tool called an `assembler` converts each assembly instruction to its corresponding machine instruction. We will discuss how an assembler works very soon, and in fact, you will implement a MIPS assembler in the assignments.

**Tool: cs241.binasm, the binary assembler**
`cs241.binasm` is an assembler that takes a MIPS assembly language program as input (from standard input) and produces a MIPS machine language program in binary as output (on standard output). Running the program is straightforward as seen in the following example:

```
$ cs241.binasm < input.asm > output.mips
```

In CS241, we use the convention to use the extension `.asm` for Assembly language files and the extension `.mips` for Machine language files.

We have already been using assembly code. Our shorthand notation (column two in the MIPS reference sheet) lists the Assembly instructions that correspond to the Machine instructions (column three). One exception is `.word`, which is not an assembly instruction but an `assembler directive`. An assembler directive is something that the assembler understands (other than an instruction). For `.word`, the assembler simply generates the binary word representing the immediate value. As each machine code instruction is also a single word, and an assembler thus generates a 32-bit word for each normal instruction, every instruction can be replaced by an equivalent `.word` (albeit this is a rather sociopathic thing to do), and many `.word` directives can be replaced by an equivalent instruction, if some instruction happens to have the same encoding as the value.

We will continue our discussion of MIPS Machine Language, but will start representing it using MIPS Assembly Language (and stop looking at the actual bit patterns).

## 3.1 Multiply and Divide

In MIPS assembly language, we can multiply the values in \$s and \$t using the syntax: `mult $s, $t`. One interesting thing to note is that a destination register is not specified. This is because the multiplication of two 32-bit integers can result in an integer that could be as large as 64 bits. To account for this, the MIPS processor stores the result in two special registers, `hi` and `lo`. The most significant word is stored in `hi`, and the least significant word in `lo`.

`mult` assumes that the numbers are in signed Two's complement representation. However, there is also the `multu` instruction, which assumes that the register values being multiplied are unsigned integers. For simplicity, in CS241, we will usually assume that multiplications do not overflow into the hi register. That is, unless explicitly stated, it is safe to assume the result of multiplying two registers is a 32-bit value that fits entirely in the `lo` register.

The `div` and `divu` instructions support signed and unsigned division. `div $s, $t` performs signed integer division and places the quotient `$s / $t` in `lo` and the remainder `$s % $t` in `hi`. The sign of the remainder matches the sign of the dividend stored in `$s`.

## 3.2 Move from hi and lo

As discussed, the results of multiplication and division are stored in the special registers `hi` and `lo`. Since these are not general-purpose registers, there are special instructions to access this data:

`mfhi $d`: Move from register `hi` into register `$d`
`mflo $d`: Move from register `lo` into register `$d`

**Example:** Write a MIPS assembly program that finds the remainder when \$1 is divided by \$2 and stores it in \$3. Assume \$1 and \$2 are unsigned integers and \$2 is nonzero.

```
divu $1, $2 ; important to use divu and not div
mfhi $3     ; recall remainder is placed in hi
jr $31      ; don't forget to exit the program
```

Write a MIPS assembly program that interprets \$1 as a unsigned base 7 number (zero or positive), finds the least significant base 7 digit of the number (which is between 0 and 6) and stores this digit in \$3. Hint: If you are not sure how to work with base 7, first think about how you would find the least significant decimal digit of a decimal number, or the least significant bit of a binary number. [6]

[6] The answer is found in Section 5.1 at the end of this module.

## 3.3 Branch on Equal and Not Equal

The assembly instruction `beq $s, $t, i`, compares the values in \$s and \$t and increments PC by `i` number of words if the values are equal. In other words, if `$s == $t` then `pc += i * 4`. The branch on not equal instruction, `bne`, will update PC similarly if the values in \$s and \$t are **not** equal. We have already discussed that PC always contains the address of the next instruction to execute. Hence, if the condition for `beq` or `bne` is not true, we simply move to the next instruction. If the value for `i` is positive, a successful branch will skip `i` instructions. However, if the value for `i` is negative, a successful branch will go "up" the code `|i|-1` instructions (where the bars represent absolute value), since PC was already at the next instruction.

**Example:** Write a MIPS assembly language program that updates register `$2` to the value 3 if the signed number in register `$1` is odd and the value 11 if the number is even.

```
lis $8
.word 2         ; $8 contains 2
lis $9
.word 3         ; $9 contains 3
lis $2
.word 11        ; $2 contains 11
div $1, $8      ; hi <-- $1 % $8
mfhi $3         ; $3 <-- hi
beq $3, $0, 1   ; if $3 == $0 skip 1 instruction
add $2, $9, $0  ; store 3 in $2
jr $31          ; exit the program
```

How do you write a branch on equal instruction that always (unconditionally) jumps by a fixed number of instructions (say 3)?
[7]

## 3.4 Set Less Than

The Set Less Than (`slt $d, $s, $t`) instruction sets the value of register `$d` to 1 if the value in register `$s` is less than the value in register `$t` and sets it to be 0 otherwise. Once again, there is also an unsigned version, `sltu`.

**Example:** Write a program that computes the absolute value of \$1 and stores the result in \$1. There are multiple ways to implement this. Two variations of equal runtime efficiency, are shown:

**Version 1:**

```
slt $2, $1, $0 ; $2 is 1 if $1 < 0
beq $2, $0, 1  ; skip 1 instruction if $2 is 0
               ; jump if $1 is NOT less than 0
sub $1, $0, $1 ; $1 = $0 - $1 (negate)
jr $31         ; exit
```

[7]In general, you can jump unconditionally by comparing a register value to itself. It is common to compare \$0 with itself, i.e., `beq $0, $0, 3`.

**Version 2:**

```
slt $2, $1, $0
bne $2, $0, 1  ; skip 1 if $2 is 1
               ; i.e. $1 was negative
jr $31         ; exit as $1 is positive
sub $1, $0, $1 ; $1 = $0 - $1 (negate)
jr $31         ; exit
```

**An aside on signed vs. unsigned interpretation**: Twice now, we have seen instructions that interpret data as either signed or unsigned values. `mult` performs signed multiplication, i.e., it assumes the values in the provided registers are in Two's complement form. `multu`, on the other hand, performs unsigned multiplication, i.e., it assumes that the bits represent an unsigned binary number. Similarly, `slt` does signed comparisons while `sltu` does unsigned comparisons. Using the wrong instruction can drastically change the results.

Type up the following program, assemble it using `cs241.binasm` and then run it using `mips.twoints`. Give register 1 the value 7 and give register 2 the value -7. When the program terminates, what are the values in \$3 and \$4?

```
slt $3, $1, $2
sltu $4, $1, $2
jr $31
```

[8]

To explain the results of the experiment, consider the 32 bits stored in registers \$1 and \$2:

```
$1   00000000 00000000 00000000 00000111
$2   11111111 11111111 11111111 11111001
```

The value in \$1 is 7 in decimal. What about the value in \$2? Is it 4294967289 or is it -7? Why do we ask? Well, if \$2 contains a big positive value, then comparing \$1 < \$2 will be true, but if \$2 contains -7, then \$1 < \$2 is false, since 7 is not less than -7. Notice that the bits in \$2 have not changed. What changes is how we decide to interpret the bits. If we decide to interpret the bits in \$1 and \$2 as Two's complement representation, which is what `slt` does, then the value in \$2 is indeed -7, and \$3 is set to 0 as 7 is not less than -7. However, if we choose to interpret the bits in \$2 as an unsigned value (which is what `sltu` allows us to do), then \$2 contains a big number and 7 is certainly less than that, therefore \$4 is set to 1. **A single piece of data can be interpreted in many ways, and which interpretation is correct or intended can only be known from context.**

Write a MIPS assembly program that interprets \$1 as a signed base 7 number, finds the least significant base 7 digit of the number (between 0 and 6) and stores this digit in \$3. Note that even if the number is negative, the digit you return should be positive. For an extra challenge, try to make sure your solution works for the minimum 32-bit two's complement integer (in decimal this is $-2^{31}$).

[9]

[8]Recall we placed 7 and -7 in \$1 and \$2 respectively. \$3 contains the value 0 (indicating \$1 is not less than \$2 as expected. However, \$4 will contain 1 (indicating that \$1 is less than \$2 which is not true!).

[9]The answer is found in Section 5.2 at the end of this module.

### 3.4.1 A conditional loop

**Example:** Calculate the value 13+12+11+10.....+1 and store it in \$3.

While we could write a naïve, hard-coded program that loads the values 13 to 1 into registers and sums over them, that would just be silly. A better approach would be to use a loop like the one shown in the pseudocode below:

```
$2 = 13
$3 = 0
repeat {
   $3 += $2
   --$2
} until ($2 == 0)
```

But we haven't studied for/while loops yet. And we won't! Because they don't exist as separate instructions in MIPS assembly language. However, we don't really need any special construct for loops: looping is simply branching based on a condition. Here is the MIPS assembly program for the pseudocode shown above:

```
lis $2
.word 13        ; $2 = 13
add $3, $0, $0 ; initialize $3 to 0
add $3, $3, $2 ; $3 += $2
lis $1
.word -1        ; $1 = -1
add $2, $2, $1 ; --$2
bne $2, $0, -5 ; if $2 != 0, go up 5 words
               ; (remember PC is already at the next word)
jr $31          ; exit
```

An interesting observation at this point is that this is not the most efficient program we could write. In particular, we load into \$1 the value -1 within the loop! Since \$1 is unchanged in each iteration, there is no reason for this to be within the loop. Let's pull the "load into \$1" out of the loop. However, if we do this, we must also make another change.

Can you think of what other change we might need to make if we are moving the two lines that load the value -1 into \$1 above the loop?

The problem is that within the branch instruction we hard coded a jump upwards by 5 instructions. We no longer need to jump up by 5 instructions. It should be just 3.

```
lis $2
.word 13       ; $2 = 13
lis $1
.word -1       ; $1 = -1
add $3, $0, $0 ; initialize $3 to 0
add $3, $3, $2 ; $3 += $2
add $2, $2, $1 ; --$2
bne $2, $0, -3 ; if $2 != 0, go up 3 instructions
               ; (remember PC is already at the next instruction)
jr $31         ; exit
```

We dodged a bullet there. Having to keep track of branch offsets when refactoring code (by adding or removing instructions from within a loop) is a dangerous task that can easily lead to errors. Fortunately for us, we are not working with machine language, but at a higher level of abstraction. Assembly language provides an extra feature: an `assembler directive` in the form of labels. Below is the same code as above, this time using labels:

```
0x00    lis $2
0x04    .word 13
0x08    lis $1
0x0c    .word -1
0x10    add $3, $0, $0
      loop:
0x14    add $3, $3, $2
0x18    add $2, $2, $1
0x1c    bne $2, $0, loop
0x20    jr $31
```

Aside from showing memory addresses (in hexadecimal) for convenience, we have made two actual changes to the program. We added a label to the start of the loop. In the branch instruction, instead of hardcoding the number of instructions to skip (often called the offset), we have used the label to refer to the place we want to jump to. Notice that while each instruction takes a word of memory, label definitions do not. We discuss the finer details below.

### 3.4.2 Using labels

It is important to remember that labels are assembler directives. Machine language has no concept of labels. This means that the assembler must reconcile the different definitions and uses of labels and produce numeric offsets that are expected in machine language. This task is straightforward: when the assembler encounters the definition of a label, it associates the name of the label with the address of the instruction at that point. In our example, the assembler associates `loop` with the address `0x14` (20 in decimal). When the assembler encounters a label being used (e.g., in a branch instruction), the assembler computes the number of instructions to skip using the formula (label − PC)/4, where label is the address associated with where the label was defined and PC is the location where PC *would* be when this instruction is executed, i.e., the next instruction.

In our specific case, the `loop` label is used in the branch instruction. At that point, PC has already been updated to the next instruction and is therefore `0x20` (32 in decimal). The offset (number of instructions to skip) is computed as: $(20 - 32)/4$ which is -3.

In MIPS Assembly language, a label can be defined at the start of any line. A label might be followed by another label, a MIPS assembly instruction or nothing. A label may only be defined once. The following code gives some examples of valid label uses (the program uses arbitrary instructions and places labels randomly but at valid locations).

```
0x00  sub $3, $0, $0
      sample:
0x04  add $1, $0, $0
      random:    ; is this the end
0x08  mylabel: done: jr $31
      end:
0x0c
```

The memory addresses we have been using indicate the memory address at which a MIPS machine instruction would occur if the program was loaded into memory at starting address 0. As discussed earlier, since labels disappear from the machine language version of the program, they do not occupy any memory in the machine language version (as seen in the code above). For this example, the assembler will associate the address `0x04` with `sample`; the address `0x08` with the labels `random`, `mylabel` and `done`; and the address `0x0c` with `end`.

Consider this MIPS Assembly code:

```
endless: beq $0, $0, endless
beq $0, $0, end
end:
```

What should this code look like after labels are removed and replaced with decimal offsets?
[10]

A video showing how to compute the factorial of a number using a loop is available.

Write a MIPS assembly language program that stores into register $3 the sum of all even numbers from 1 to 20 inclusive. Use labels instead of hardcoding offsets.
[11]

[10] The MIPS Program with labels replaced with hardcoded offsets is:
```
beq $0, $0, -1  ; (0-4)/4
beq $0, $0, 0   ; (8-8)/4
```
This might be unintuitive given the position of the labels! It is important to understand the rules for translating labels into offsets.

[11] The answer is found in Section 5.3 at the end of this module.

## 3.5 Store Word and Load Word

While discussing the MIPS processor, we mentioned that there is limited memory available to a processor in the form of registers. However, the processor has the ability to access a much larger bank of memory, RAM, though it is slower.

The Store Word instruction, `sw $t, i($s)`, takes the 32-bit value currently in register \$t and stores it at MEM[\$s + i], where i is a 16-bit Two's complement immediate. Notice how the value in \$s is treated as a 32-bit address. Since each address in memory can store 8 bits, storing a word (32 bits) at MEM[\$s +i] causes the most significant byte to be stored at MEM[\$s + i] while the least significant byte is stored at MEM[\$s + i + 3].

The MIPS processor can only perform operations (arithmetic or logical) on data that is in registers. Therefore, any value that has been stored into memory must first be loaded into a register before operating on it. The Load Word instruction `lw $t, i($s)`, loads a word starting at MEM[\$s + i] and places it into \$t. Recall that in our processor, each word is 4 bytes. As discussed earlier, when we access memory, MEM[\$s+i], it is accessing 4 bytes starting at the address \$s + i. Older versions of MIPS processors only supported `aligned` access, i.e., the starting memory address to read/write from, \$s+i in our example, had to be a multiple of 4. While some new processors support `unaligned memory access`, it is often slower, and requires the use of special instructions which our reduced version of MIPS doesn't implement. **In CS241, we assume that our hardware only supports aligned access. Therefore, memory addresses that access memory (load/store) must be multiples of 4.**

**Example:** Suppose \$1 and \$2 contain memory addresses (properly aligned, i.e., they are multiples of 4). Write a MIPS program that copies the value at the address in \$1 into the address in \$2.

The question tells us that \$1 contains an address. We are to copy the value at that address into the location whose address is present in \$2. The following program achieves this task:

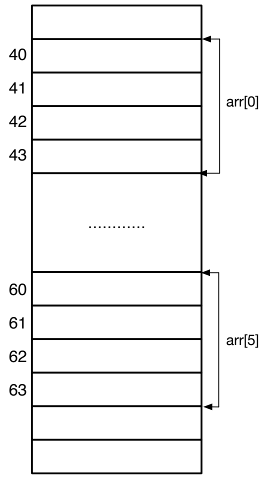
```
lw $3, 0($1)
sw $3, 0($2)
jr $31
```

We begin by loading the value stored at the address in \$1 into register 3. Then we store the value we just loaded, i.e., the value that is currently in \$3, at the address that is contained in \$2.

**Example:** \$1 contains the starting address of an array of 32-bit integers, `arr`, stored in memory. \$2 contains the number of elements in `arr`. Retrieve `arr[3]` into \$3.

Before we write the assembly program to solve the question, let's visualize the array in memory:

$1 contains the starting address of an array. For this example, let's assume the address is 40 (in decimal). Let's also assume that there are 6 elements in the array, i.e., $2 is 6.



Since the first element (index 0) starts at address 40, the word consumes the memory addresses 40 through 43 (inclusive). The next element (index 1) will consume addresses 44 to 47 and so on. The last element (index 5) will consume the addresses 60 through 63 (inclusive).

The question asks us to load the element at index 3. The offset into the array for index 3 will be 4 x 3 = 12. We can add this offset to the starting address of the array to get the starting address for `arr[3]`, 40+12 = 52.

Once we have understood how the array is situated in memory and how offsets into the array are calculated, writing the solution is straightforward:

```
lis $5
.word 3  ; index of element to retrieve
lis $4
.word 4
mult $4, $5    ; offset into array
mflo $5        ; assuming that the result fits in 32-bits
add $5, $5, $1 ; starting address of element at index 3
lw $3, 0($5)   ; $3 = MEM[$5 + 0]
jr $31
```

While the program above solves the question, it is not the simplest. We presented it to demonstrate using multiplication to compute offsets. A simpler solution is:

```
lw $3, 12($1)  ; $3 = MEM[$1 + 12]
jr $31
```

Since we know beforehand the element we need to access, we can hardcode the offset into a load instruction. In situations where we need an arbitrary element or if we need to iterate over all the elements, knowing how to compute offsets is essential.

**Tool: mips.array, another MIPS processor simulator**
`mips.array` is another simulator for running MIPS machine language programs. The simulator takes a filename as a **command line argument** that must contain a binary MIPS machine language program (either produced by writing machine instructions in hexadecimal notation and then using `cs241.wordasm` or writing an assembly language program and then using `cs241.binasm`). The tool will simulate the execution of the program as if it were running on a MIPS processor and, once the program ends, print the output and values of all registers.
Some important notes on running the tool:

- The tool simulates a loader that has loaded the program at address `0x00`.
- `mips.array` expects to read (from standard input) an integer n which will be treated as the length of an array. This value is stored in \$2.
- After reading this integer, the simulator expects to read $n$ more integers which will be stored as consecutive 32-bit integers in memory. The starting address of the array containing these $n$ integers is stored in \$1.

\$1 contains the address of an array and \$2 contains the number of elements in this array. Write an assembly language program that places the value 7 in the last possible spot in the array.
[12]

### 3.5.1 Input/Output from MIPS Programs

MIPS programs can read from standard input using a load instruction to load from the specially designated address `0xFFFF0004`[13]. Loading from this address will read **one** character from standard input and store it as the least significant byte within the specified register.

```
lis $1
.word 0xFFFF0004
lw $3, 0($1) ; load one character from standard input into $3
```

MIPS programs can also generate output (on standard output) using a store instruction. The special address to write to standard output is the address `0xFFFF000C`. Storing a value at this address will send the least significant byte in the specified register to standard output.

```
lis $1
.word 0xFFFF000c
lis $2
.word 67      ; ASCII for upper case C
sw $2, 0($1) ; outputs C to standard output.
```

[12]The answer is found in Section 5.4 at the end of this module.
[13]Making I/O devices accessible through special memory addresses is called *memory-mapped I/O*, and is how virtually all I/O devices are implemented on modern systems. You'll learn more about memory-mapped I/O in CS350.

## 3.6 Implementing Procedures in Assembly Language

Being able to rely on helper functions/procedures is always useful. While assembly language provides no special syntax for writing procedures, we can still write them by leveraging assembly instructions and assembler directives. That said, a number of issues have to be resolved:

- In high-level languages, variables defined within a function have local scope. In assembly language, registers have global scope. We will need a convention that ensures that calling a procedure does not overwrite data in registers that the caller might still need.
- How do we transfer control to the callee and how do we return to the caller?
- How will procedures calling other procedures work? What if a procedure calls itself recursively?
- How will we pass arguments and return values from a procedure?

We have created a video illustrating the final conventions we choose to follow in CS241. We suggest you **read through the following sections first** and then watch the Calling Procedures video.

We discuss below these issues in more details to arrive at a convention for writing and using procedures in assembly programs. It is worth noting that other approaches exist.

### 3.6.1 Protecting data in registers

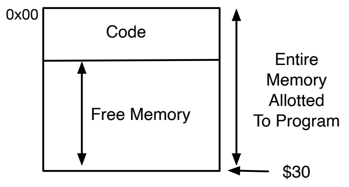
Consider the following scenario:

```
; $3 contains critical data
; call procedure foo
                            ; procedure foo
                            ; foo modifies value in $3
                            ; foo returns
; data in $3 lost!!
```

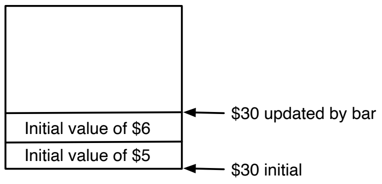
One option would be to reserve some registers for the caller and some others for `foo`. If you give this some thought, you will identify issues. Suppose we have deeper calls, e.g., `foo` now calls `bar`. Do we now reserve a subset of registers for `bar` as well? What if `foo` recursively calls itself?

A better approach is to ensure that a procedure leaves register values unchanged, i.e., the caller has the guarantee that when control returns back to it, registers contain the same values they did when the procedure was called. The question of course is: how does the callee guarantee this since it likely needs to use registers? The answer to this is that the callee can **store** within RAM the original values of registers, use the registers and then, just before returning to the caller, **restore** the values from RAM. One concern is that each procedure cannot arbitrarily decide which memory locations it will use (this could cause overwrites for example in recursive procedure calls). We must therefore use memory systematically by keeping track of which part of memory has already been used and which part is still free.

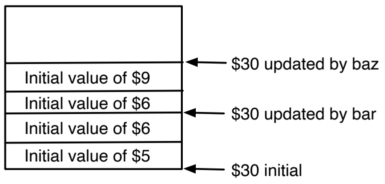
This approach will work but requires us to know which part of memory is free when the program starts executing. The MIPS loader comes to the rescue once again: it sets $30 to the address **just past** the last word of memory allotted to the program (recall that this includes memory to store the code for the program and additional memory for the program to use). This means we can use the address in $30 as a bookmark to separate used memory from unused memory.



Let's consider a point during the execution of a program when we are within procedure `foo` and it calls procedure `bar`. The first thing `bar` does is to store the current values of registers that it plans to use. The figure assumes `bar` uses $5 and $6. At the same time `bar` must update $30 to ensure that it accurately reflects that boundary between used and unused memory.



Now suppose that `bar` changes registers $5 and $6 and then calls `baz`. The first thing `baz` does is to store the original values of registers that it must use. Say $6 and $9. Obviously, `baz` must also update $30 to reflect the boundary between used and unused memory.



When `baz` is done, just before returning, it restores the values it overwrote ($6 and $9) and also restores $30. When control reaches back to `bar` it will find register contents to be exactly the way they were when it called `baz`. Once `bar` is done, just before returning, it will restore the values it overwrote ($5 and $6) and also $30.

Notice something interesting on how the procedures are using memory: procedures **push** registers they intend to use onto memory and when returning **pop** these registers from memory. That's right! Free memory is being used as a stack. $30 is the stack pointer as it always points to the top of the stack. Because our stack is at the *end* of available memory, the stack grows down in terms of memory addresses.

Pushing a register value on the memory stack is done with a store instruction and an update to the stack pointer.

```
sw $1, -4($30)   ; storing $1
lis $1
.word 4
sub $30, $30, $1 ; update stack pointer
```

Notice that the value is stored at address $30 - 4, since $30 represents the top of the stack (there is already data there).

Popping a value from the memory stack into a register is done with a load instruction and an update to the stack pointer.

```
lis $1
.word 4
add $30,$30,$1 ; update stack pointer
lw $1, -4($30) ; load into $1
```

An optimization that is possible, in case multiple registers are being pushed (or popped), is to batch the stores (or loads) and then update the stack pointer in one go.

### 3.6.2 Calling and returning from procedures

Assembly language has no special construct for defining a procedure. A procedure in assembly is simply a label in front of assembly instructions that represent the code for the procedure. Calling a procedure requires jumping to this label. We have looked at branch instructions, which can be used to jump to a label. Let's see why this will not work:

```
beq $0, $0, foo
                    foo:
                    ...............
                    ................
                    beq $0, $0, ???    ; how do you return?
```

The branch instruction on the left in the code above will indeed jump to the label `foo` (provided the offset computed is in range). However, how do we then return from `foo`? We could consider labeling the call site (place in the caller where the call was made) and then having `foo` jump back to the call site. But that will only work for one call site since the branch in `foo` must specify a unique label.

Another control flow instruction we have studied is the jump instruction. However, this does not work for the same reason as the one above.

```
lis $21
.word foo ; load address of foo
jr $21
                 foo:
                 ............
                 ............
                ; How do we return? Same problem as beq
```

The code above shows a new way of using the `.word` assembler directive. In our MIPS assembly language, we can follow the `.word` directive with a label. The assembler will place, in the assembled program, the memory address that it had associated with the label. The problem with both approaches is that the callee needs to know the return address so that once it is done, it can jump back to that address. The return address will be the current value of PC when the call is made. MIPS machine and assembly language has a special instruction named **jump and link register**. The instruction `jalr $s` stores into register $31 the current value of PC and then sets PC to $s.

```
lis $21
.word foo ; load address of foo
jalr $21 ; $31 = PC, PC = $21

                 foo:
                 ............
                 ............
                 jr $31 ; jump to address stored in $31
```

We are close! The only issue is that when we used `jalr` we overwrote the value currently in $31. Recall that the loader had set up $31 with the address that MIPS programs should jump to in order to exit the program. We have overwritten that address. To prevent this from happening, anytime we use `jalr`, we must first preserve the address currently stored in $31. That's easy! Let's use the same strategy we use for preserving register values; push $31 on the stack before using `jalr` and pop it off the stack once the call returns.

```
sw $31, -4($30) ; Push $31 to stack
lis $31         ; Using $31 since it was just saved
.word 4
sub $30, $30, $31

lis $21
.word foo
jalr $21
                  foo:
                  ............
                  jr $31

lis $31    ; Using $31 since we will pop from stack
.word 4
add $30, $30, $31
lw $31, -4($30) ; Pop $31 from stack
............
jr $31 ; return to loader
```

### 3.6.3 Procedures calling other procedures, recursion

The decisions we made regarding preserving registers when calling procedures and how we call and return from procedures mean that we have to do nothing special if a procedure calls another procedure. As the call chain gets deeper, the stack will grow (much like how we are accustomed to in our C/C++ memory model). This suffers from the same problem that uncontrolled recursive functions have; at some point the stack might grow so much that it might begin to overwrite our program! The simulators in CS241 have no special checks to prevent this from happening and will likely generate a generic error such as `MIPS emulator internal error` once the stack starts overwriting the program segment of the code.

### 3.6.4 Passing and returning values from procedures

There are two approaches to passing arguments to a procedure; we could use registers (often preferred when the number of parameters is relatively small) or we could use the stack (number of parameters bounded only by stack size!). A procedure must document where it expects its arguments to be.

Return values can be dealt with similarly; our convention is to use $3 to store the value being returned by a function although the top of the stack could also be used. Once again, documentation is crucial so that the caller knows where to look for the return value. Of course, unlike every other register, the value in the return register should not be preserved and restored on the stack, as the procedure is intended to overwrite it.

**Example:** Write a procedure to sum numbers 1 to N and store result in $3.

In Section 3.4.1, we wrote a small program that computed `13+12+11+10+.....+1`. This example generalizes that code as a procedure:

```
; Sum1ToN adds all numbers 1 to N
; Registers:
;    $1 scratch   (original value should be preserved)
;    $2 input number, N (original value should be preserved)
;    $3 output (don't preserve original value)
Sum1ToN:
sw $1, -4($30)     ; save previous value of $1
sw $2, -8($30)     ; save previous value of $2
lis $1
.word 8
sub $30, $30, $1   ; update stack pointer

add $3, $0, $0     ; initialize to 0
lis $1
.word -1

loop: add $3, $3, $2
add $2, $2, $1
bne $2, $0, loop

lis $1
.word 8
add $30, $30, $1   ; update stack pointer
lw $1, -4($30)     ; restore original value of $1
lw $2, -8($30)     ; restore original value of $2
jr $31  ; return to caller
```

One important point to notice is that we initialize $3 (our accumulator for the sum) to 0 in the procedure. If you have been using the simulators, you might have started to rely on the fact that the simulators begin with most of the unused registers already set to 0. Relying on this behaviour is a bad idea for the same reasons as relying on a local variable defaulting initialized to 0 in C/C++ (some compilers might not do it). Additionally, now that we are working with procedures, the assumption that a register begins with a value of 0 is often not true. For example, if the caller of `Sum1toN` had previously used $3, forgetting to clear $3 in the procedure will have catastrophic results.

If you haven't already, watch the video describing the process of Calling a Procedure.

A video showing how to compute the factorial of a number using a recursive function call is available.

A video tracing through recursion while computing factorials is also available.

Write a MIPS procedure named `SumDigits` that takes a positive integer in register \$1 and stores the sum of the digits in register \$3. For example, if \$1 contains 123, \$3 should contain 6.
[14]

Write a MIPS assembly language program that uses the procedure `SumDigits` to sum all the digits in \$1 and \$2. The result is stored in \$3. Values in \$1 and \$2 need not be preserved by the program. For example, if \$1 contains 123 and \$2 contains 4567, \$3 should contain 28.
[15]

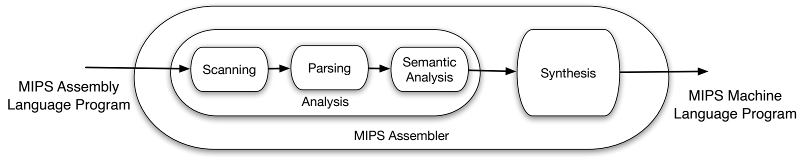
Write a **recursive** MIPS procedure named `Sum` that computes the sum of an array of integers. The procedure takes the starting address of the array in \$1, and the number of elements in \$2. The result is stored in \$3.
[16]

[14]The answer is found in Section 5.5 at the end of this module.
[15]The answer is found in Section 5.6 at the end of this module.
[16]The answer is found in Section 5.7 at the end of this module.

# 4 MIPS Assembler

When we introduced MIPS Assembly Language (Section 3), we discussed the MIPS assembler, a tool that converts each assembly instruction to its corresponding machine instruction. In this section, we discuss what it takes to implement such a tool.

An assembler must make sense of each line in the assembly language program and determine that there are no syntax errors or violations of Assembly Language rules. This is the **Analysis** stage. Once the assembler has determined that the input is a legal program, the **Synthesis** stage generates the output, the equivalent MIPS Machine Language program.



## 4.1 Analysis

The Analysis stage is itself a multi-stage process. The input to the assembler is a text file, the MIPS Assembly language program, which can be viewed as a string of characters. In the first Analysis stage, this string of characters is chopped up into meaningful **tokens**, such as labels, numbers, `.word`, MIPS instruction names, etc. This tokenization is often referred to as the **Scanning** stage, and the module that does it is called the **Scanner**. We will discuss writing Scanners and the supporting theory behind it in more detail in future modules.

The result of the scanning stage is a sequence of tokens. For example, the input line

```
loop:  add $1, $2, $3
```

could be converted into a stream of tokens LABEL ID REG COMMA REG COMMA REG. Each token contains additional attributes, e.g., LABEL would contain the string `loop`, ID would contain the string `add`, REG would contain the register number, etc.

The next Analysis stage is parsing. In general, parsing acts as a syntax checker and in doing so determines the structure of the program as well. In the case of Assembly Language, each line of MIPS code is independent of others, i.e., there are no nested structures such as conditional or nested blocks common in higher-level languages. Therefore, writing a **parser** for Assembly Language is straightforward; just check that the sequence of tokens for each line of input follows the syntax for some assembly instruction. For the `add` instruction from above, we can confirm that the ID token is followed by a REG token, followed by COMMA, then another REG, a COMMA then another REG and then followed by nothing other than perhaps a comment [17]. As discussed in Section 3.4.2, any assembly line can begin with one or more labels. Additionally, it might be necessary to check the attributes of certain tokens, e.g., for an ID token, we can confirm that the string is a valid instruction name, for a REG token the register value should be in range, and any token that represents an immediate has a value that is in range for that type of instruction.

[17]Most scanners don't even generate comment tokens, and just discard comments right away.

Once we have performed the syntax checks, the next stage is Semantic Analysis. In a higher-level language, this stage would involve things like checking variable names are not duplicated, types match, etc. In Assembly Language, this stage checks that there are no duplicate label names.

## 4.2 Synthesis

Once the Analysis stage is complete, and the input program is known to be a correct assembly program, the synthesis stage generates the output, the machine instruction equivalent for each assembly instruction. This is trivial to do for most instructions other than dealing with assembler directives (labels and `.word`) and perhaps nuances of the programming language you are writing your assembler in. We discuss both these concerns in the following sections.

### 4.2.1 Dealing with labels

Dealing with labels poses an unexpected problem when writing the assembler. We recommend reading Section 3.4.2 once again before continuing.

Since labels are assembler directives, and do not exist in machine language, the assembler must convert each use of a label into either an address (if used with `.word`) or to an offset (if used within a branch instruction). As discussed in Section 3.4.2, the assembler can do this by associating an address (the current value of PC) to each definition of a label and then use this address when a label is used. However, this presents a problem. Below, we have two valid uses of labels.

```
begin:                          beq ..... myLabel
...................             ............
beq $0, $0, begin               myLabel:
```

For the code on the left, the label `begin` is defined before it is used. During the parsing stage, the assembler will encounter this definition and can update a data structure, referred to as a **symbol table**, which maps the label's string to the current value of PC. This means that the assembler will need to track PC by maintaining a virtual count of instructions that it has seen so far. At this point, the assembler can also ensure the uniqueness of labels; if, when trying to insert a new entry into the symbol table, an entry for this label already exists, the assembler can generate an appropriate error indicating that a duplicate has been detected.

Later, during Synthesis, when the assembler encounters the branch instruction that uses `begin`, it can refer to the symbol table, obtain the address at which this label was defined and use it to compute the offset using the formula discussed in Section 3.4.2. Similarly, if `begin` was used with the `.word` directive, this is even simpler; the assembler replaces `.word begin` with the address associated with `begin` in the symbol table.

Now, let's consider the code snippet on the right. This code uses the label `myLabel` before defining it. This is legal assembly language, as there is no requirement that a label must be defined before it is used. This problem has plagued compiler writers since the beginning of higher-level languages, and you have likely seen it discussed in one form or the other. For example, in C you cannot use a function before it has been `declared`. Unfortunately, assembly language does not have special syntax for forward-declaring labels. This means that the assembler must somehow account for labels that are used before they are defined. The solution is to implement the assembler in two **passes**.

- Pass 1: Scanning, Parsing (including the generation of the symbol table along with duplicate label checks)
- Pass 2: Semantic Analysis (to check that labels that are used are in fact defined) and Synthesis

The following is a sample MIPS assembly program that demonstrates the symbol table that the assembler would create for this program.

```
0x00        main:  lis $2
0x04               .word beyond
0x08               lis $1
0x0c               .word 2
                   ;Ignore
0x10               add $3, $0, $0
            top:  begin:
0x14               add $3, $3, $2
0x18               sub $2, $2, $1
0x1c               bne $2, $0, top
0x20               jr $31
            beyond:
0x24
```

Symbol Table

| label | addr |
|---|---|
| main | 0x00 |
| top | 0x14 |
| begin | 0x14 |
| beyond | 0x24 |

In the example above, what value does the assembler substitute for `.word beyond`?
[18]

In the example above, the instruction `bne $2, $0, top` uses the label `top`. The assembler will compute the actual offset. What will it be?
[19]

### 4.2.2 Encoding the instruction

If an assembly program is valid, the assembler must produce the MIPS machine language equivalent on standard output. There are two steps to the process: computing the 32 bits that represent each instruction within the assembler, i.e., creating an encoding for the instruction and then sending this output to standard output. In this section, we focus on encoding the instruction and the next section shows how to generate the final output.

**Example:** Generate an encoding for the instruction: `add $3, $2, $4`

From the MIPS reference sheet, `add $d, $s, $t:  000000 sssss ttttt ddddd 00000 100000`

[18]The symbol table shows that `beyond` has the address `0x24`. The assembler will use this value.
[19]Offset Calculation:
From the symbol table, `top` is defined at address: `0x14`
PC at the time `top` is used: `0x20`
offset = (`defn` − `PC`)/4 = (`0x14` − `0x20`)/4 = (20 − 32)/4 = −3.

Therefore, `add $3, $2, $4` should produce the following (components are highlighted):

$$\underbrace{000000}_{\substack{6bits\\ opcode}}\underbrace{00010}_{\substack{5bits\\ \$s}}\underbrace{00100}_{\substack{5bits\\ \$t}}\underbrace{00011}_{\substack{5bits\\ \$d}}\underbrace{00000}_{\substack{5bits\\ padding}}\underbrace{100000}_{\substack{6bits\\ func}}$$

We will assume that the assembler has readied the following `int` values: `s`, `t` and `d` with the corresponding register values, `op` (short for operation code) with the value 0, and `func` (short for function code) with the value 32 (`100000` in binary). In C++, `int` values are stored as 32-bit values in Two's complement representation.

```
Value    Decimal  32-bit int in binary (space for emphasis)
op:      0        00000000000000000000000000 000000
s:       2        000000000000000000000000000 00010
t:       4        000000000000000000000000000 00100
d:       3        000000000000000000000000000 00011
func:   32        00000000000000000000000000 100000
```

Given these values, the first thing to do is to shift the bits into the position that component has in the final 32 bits that will represent the instruction. The 6 least significant bits of `op` need to be shifted left by 26 so that they occupy the 6 most significant bits (technically not needed as the bits are all 0 for `add`). The 5 least significant bits of `s` need to be shifted left by 21 so that they occupy the 5 bits reserved for $s. Similarly, the 5 bits of `t` need to be shifted by 16, `d` shifted by 11 and `func` bits do not need to be shifted since they are already in their designated spot. In C++, we can use the << operator and in Racket the `arithmetic-shift` function:

```
op:  (arithmetic-shift 0 26)  0 << 26     000000 00000 00000 00000 00000 000000
s:   (arithmetic-shift 2 21)  2 << 21     000000 00010 00000 00000 00000 000000
t:   (arithmetic-shift 4 16)  4 << 16     000000 00000 00100 00000 00000 000000
d:   (arithmetic-shift 3 11)  3 << 11     000000 00000 00000 00011 00000 000000
func:                    32         32     000000 00000 00000 00000 00000 100000
```

Once the useful bits are in the right spot within the corresponding values, we can use bitwise OR to produce the result:

```
(define instr (bitwise-ior (arithmetic-shift 0 26)
                           (arithmetic-shift 2 21)
                           (arithmetic-shift 4 16)
                           (arithmetic-shift 3 11)
                           32 ))
int instr = (0 << 26) | (2 << 21) | (4 << 16) | (3 << 11) | 32;
```

The variable `instr` contains the 32 bits that represent the instruction, we will discuss how to output this to standard output after another example on instruction encoding.

**Example:** Generate an encoding for the instruction: `beq $1, $2, -3`

From the MIPS reference sheet, `beq $s, $t, i:  000100 sssss ttttt iiiiiiiiiiiiiiii`

Therefore, `beq $1, $2, -3` should produce the following (components are highlighted):

$$\underbrace{000100}_{\substack{6bits\\opcode}}\underbrace{00001}_{\substack{5bits\\\$s}}\underbrace{00010}_{\substack{5bits\\\$t}}\underbrace{1111111111111101}_{\substack{16bits\\offset}}$$

As before, let's assume that the assembler has readied the values `op` (value 4, `000100` in binary), `s` and `t` with values 1 and 2 respectively and `offset` with a value of -3.

```
Value   Decimal    32-bit int in binary (space for emphasis)
op:        4       00000000000000000000000000 000100
s:         1       000000000000000000000000000 00001
t:         2       000000000000000000000000000 00010
offset:   -3       1111111111111111 1111111111111101 (2's complement)
```

We can now shift the different values so that the bits within are at the intended positions.

```
op: (arithmetic-shift 4 26)  4 << 26      000100 00000 00000 0000000000000000
s:  (arithmetic-shift 1 21)  1 << 21      000000 00001 00000 0000000000000000
t:  (arithmetic-shift 2 16)  2 << 16      000000 00000 00010 0000000000000000
offset:                     -3            111111 11111 11111 1111111111111101
```

Previously, once we had positioned the bits in their appropriate places, we used bitwise OR to produce the resulting instruction. But we have a problem! Notice that `offset` is a 32-bit negative value. Directly using bitwise OR on these four values will simply produce `offset` (due to the 1s in the two most significant bytes of `offset`). Since `offset` is supposed to be only 16 bits, let's first apply a **mask** to only get the last 16 bits by clearing the two most significant bytes:

```
  1111111111111111 1111111111111101
& 0000000000000000 1111111111111111 (0xFFFF in hexadecimal)
  ---------------------------------
  0000000000000000 1111111111111101

offset:  (bitwise-and -3 #xFFFF)  -3 & 0xFFFF  000000 00000 00000 1111111111111101
```

Now, we can apply bitwise OR on the values:

```
(define instr (bitwise-ior (arithmetic-shift 4 26)
                           (arithmetic-shift 1 21)
                           (arithmetic-shift 2 16)
                           (bitwise-and -3 #xFFFF)))
int instr = (4 << 26) | (1 << 21) | (2 << 16) | (-3 & 0xFFFF);
```

The variable `instr` contains the 32 bits that represent the instruction.

## 4.3 Generating output

Now that the assembler has the 32-bit encoding in a variable, it seems like there wouldn't be much left to do; just use the print functionality. Unfortunately, it is not that simple.

```
int instr = (0<<26) | (2<<21) | (4<<16) | (3<<11) | 32;   // add $3, $2, $4
cout << instr;
```

When this code runs, it will produce on standard output the characters: 4462624. This might be surprising at first, but it really isn't. We output an `int` using `cout` all the time and expect to see the decimal value produced. This is exactly what happened; the output operator for integers outputs the ASCII values for each of the digits in the decimal representation of the value which the shell interprets to display the appropriate characters.

This is obviously not what we want; remember, the value for the ASCII glyph 0 is actually 0x30 (48), 1 is 0x31 (49), etc., so by outputting ASCII, the actual bits that we're outputting are nothing like what we intended. We want the actual 32 bits stored in the variable to be sent to standard output. We can use the `unsigned char` type to do this. Consider the following code:

```
int x = 65;
unsigned char c = 65;
cout << x << c;
```

When this code is executed, it will print 65A. Both x and c have the same bits in the least significant byte. The integer is meant to be displayed as a number, so its decimal digits are extracted and converted to ASCII and then sent to standard output. The char value is already assumed to be in ASCII, so the bits are sent to standard output as is without any additional conversions. Note that while char stands for character, it is really just an 8-bit numeric type, and it is useful for more than just ASCII data. Above we stored 65 in an `unsigned char`. This caused C++ to output the binary representation of 65 to standard output. Since 65 happens to be the ASCII code of uppercase A, and terminals normally interpret the output of programs as ASCII, an uppercase A was printed.

However, what if we stored the number 241 (binary 11110001) in the `unsigned char`? While the value is in the numeric range for an `unsigned char`, it is not a valid ASCII character code. However, C++ would still write the binary representation of 241 to standard output. So how would your terminal display this? It depends on the terminal since this is not standard ASCII. Some terminals might display some kind of "extended ASCII" character, some might display a question mark symbol to indicate an unknown character, and some might display nothing at all.

There are two lessons here. One, that our use of `unsigned char` has nothing to do with "characters" and is just a way to output raw binary data in C++. Second, that when you view the binary data on your terminal, it will not be human-readable. Most parts of binary-encoded MIPS instructions correspond to non-printable or invalid ASCII characters!

Using the output operator for `unsigned char` types, we can use the following code to output MIPS instructions in binary:

```
int instr = (0<<26) | (2<<21) | (4<<16) | (3<<11) | 32;   // add $3, $2, $4
unsigned char c = instr >> 24;
cout << c;
c = instr >> 16;
cout << c;
c = instr >> 8;
cout << c;
c = instr;
cout << c;
```

If you try running this example code, on your terminal, you will likely see an uppercase D and a space, possibly accompanied by some other strange characters, but maybe not. This is because the second byte of the MIPS instruction happens to correspond to the ASCII code for D, and the fourth byte corresponds to a space, but the other bytes correspond to non-printable ASCII characters, and how these are displayed varies between terminals! You should use tools such as `xxd` or the CS241 tool `cs241.binview` to view the data in a readable form.

Racket has the same issue. We can use the `write-byte` function to output the raw bytes.

```
(define instr (bitwise-ior (arithmetic-shift 4 26)
                           (arithmetic-shift 1 21)
                           (arithmetic-shift 2 16)
                           (bitwise-and -3 #xFFFF)))  ; encoding for beq $1, $2, -3
(write-byte (bitwise-and (arithmetic-shift instr -24) #xFF))
(write-byte (bitwise-and (arithmetic-shift instr -16) #xFF))
(write-byte (bitwise-and (arithmetic-shift instr -8) #xFF))
(write-byte (bitwise-and instr  #xFF))
```

# 5 Answers to longer self-practice questions

## 5.1 Self-practice answer 1

```
lis $7      ; use $7 to hold 7 as it is easy to remember
.word 7
divu $1, $7 ; unsigned division since $1 is an unsigned number
mfhi $3
jr $31
```

## 5.2 Self-practice answer 2

Write a MIPS assembly program that interprets \$1 as a signed base 7 number, finds the least significant base 7 digit of the number (between 0 and 6) and stores this digit in \$3. Note that even if the number is negative, the digit you return should be positive. For an extra challenge, try to make sure your solution works for the minimum 32-bit two's complement integer (in decimal this is $-2^{31}$).

We already looked at a similar question when we introduced the `mfhi` and `mflo` instructions. However, in that example \$1 was an unsigned number whereas now \$1 contains a signed number. Therefore, we must now use `div` instead of `divu`, and after finding the remainder we take its absolute value to get a non-negative digit.

```
lis $7
.word 7
div $1, $7
mfhi $3
slt $5, $3, $0  ; $5 == 1 if $3 < 0, $5 == 0 if $3 >= 0
beq $5, $0, 1   ; skip next instruction as $3 is not negative
sub $3, $0, $3  ; $3 is negative, so negate it
jr $31
```

This program handles the case of the minimum two's complement integer correctly. Note that if we tried to take the absolute value of the original number, instead of the absolute value of the remainder, this case would not be handled correctly! This is because the minimum 32-bit two's complement integer is $-2^{31}$, but the maximum 32-bit two's complement integer is $2^{31} - 1$. The negation of the minimum is greater than the maximum, so negation does not work! If you attempt to do the "flip the bits and add one" trick with $-2^{31}$ you will actually find that it does not change the bit pattern at all.

## 5.3 Self-practice answer 3

Write a MIPS assembly language program that stores into register $3 the sum of all even numbers from 1 to 20 inclusive. Use labels instead of hardcoding offsets.

The solution is below. We have added addresses and offset computation for illustration purposes.

```
0x00  lis $2
0x04  .word 20
0x08  lis $1
0x0c  .word 2
0x10  add $3, $0, $0
      top:
0x14  add $3, $3, $2
0x18  sub $2, $2, $1
0x1c  bne $2, $0, top
0x20  jr $31
```

The assembler will compute the offset for the branch instruction as (top-PC)/4 = (0x14-0x20)/4 = (20-32)/4 = -3.

## 5.4 Self-practice answer 4

$1 contains the address of an array and $2 contains the number of elements in this array. Write an assembly language program that places the value 7 in the last possible spot in the array.

```
lis $4
.word 4
mult $2, $4
mflo $3
add $3, $3, $1 ; address just past end of array
lis $8
.word 7
sw $8, -4($3) ; $3 - 4 is address of last element
jr $31
```

### 5.5 Self-practice answer 5

Write a MIPS procedure named `SumDigits` that takes a positive integer in register $1 and stores the sum of the digits in register $3. For example, if $1 contains 123, $3 should contain 6.

```
; SumDigits computes sum of digits of integer in $1
; I/O Registers:
;     $1 positive integer (preserved)
;     $3 output (not preserved)

; strategy: repeatedly divide $1 and sum the remainders
;   $1 will always contain remaining number
;   $4 is used as a temp variable
;   $10 holds 10

SumDigits:
; prologue
sw $1, -4($30)  ; push $1
sw $4, -8($30)  ; push $4
sw $10,-12($30) ; push $10
lis $4
.word 12
sub $30, $30, $4   ; update stack pointer

lis $10
.word 10
add $3, $0, $0     ; clear $3 the result
loop:
  div $1,$10
  mfhi $4          ; mfhi has remainder
  add $3, $3, $4   ; add to sum
  mflo $1          ; quotient
  bne $1, $0, loop ; if not zero divide again

; epilogue
lis $4
.word 12
add $30, $30, $4 ; update stack pointer
lw $1, -4($30)   ; pop $1
lw $4, -8($30)   ; pop $4
lw $10, -12($30) ; pop $10
jr $31
```

## 5.6 Self-practice answer 6

Write a MIPS assembly language program that uses the procedure `SumDigits` to sum all the digits in $1 and $2. The result is stored in $3. Values in $1 and $2 need not be preserved by the program. For example, if $1 contains 123 and $2 contains 4567, $3 should contain 28.

Solution: we will call `SumDigits` twice, once for $1 and once for $2. Then sum the results. One reason for using this example is to demonstrate that we do not need to preserve the value in $31 between successive calls to procedures. Once we have pushed $31 on the stack, we can call as many procedures as we want without re-preserving $31. Once we are done making all the procedures calls, we restore $31 once.

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

; $1 already contains first input from user
lis $3
.word SumDigits
jalr $3        ; call procedure

add $4, $3, $0 ; result of call is in $3, save it to a different register before
               ; calling SumDigits again as it uses $3 to return the result

; need to call SumDigits on $2
add $1, $2, $0 ; SumDigits expects argument in $1
lis $3
.word SumDigits
jalr $3        ; call procedure, no need to preserve $31 again

add $3, $4, $3 ; $3 contains result of SumDigits on original $2
               ; $4 contains result from first call, SumDigits on original $1.
               ; After the instruction, $3 contains the result we wanted

lis $31        ; pop $31 from stack
.word 4
add $30, $30, $31
lw $31, -4($30)
jr $31

; code for SumDigits from previous example pasted here
```

## 5.7 Self-practice answer 7

Write a **recursive** MIPS procedure named `Sum` that computes the sum of an array of integers. The procedure takes the starting address of the array in \$1, and the number of elements in \$2. The result is stored in \$3.

There are numerous ways of implementing such a function. One interesting implementation is to use an accumulator:

```
int accSum(int *start, int *end, int acc){
  if (start>=end) return acc;
  return accSum(start+1, end, acc + *start);
}
```

We leave the above as an exercise for the curious.

Our MIPS assembly solution to the problem is essentially a translation of the following C++ program:

```
int sum(int *a, int N){
  if (N <= 0) return 0;
  return sum(a, N-1) + a[N-1];
}
```

```
Sum:
; $1 starting address of array (never changed)
; $2 number of elements in the array (preserved)
; $3 result (not preserved)

; $4 and $5 are used as temporary registers

; push all registers we change
; could have been more efficient and only pushed as needed
sw $2, -4($30)
sw $4, -8($30)
sw $5, -12($30)
sw $31,-16($30)
lis $4
.word 16
sub $30, $30, $4

add $3,$0,$0  ; clear $3 the result

beq $2, $0, end ; N = 0 so done
slt $4, $2, $0  ; $4 is 1 if N < 0
lis $5
.word 1
beq $5, $4, end ; N < 0 so done
```

```
; if we get here we need a recursive call
sub $2, $2, $5 ; N = N - 1
lis $5
.word Sum
jalr $5        ; result of call will be in $3

lis $4         ; retrieve a[N-1], recall $2 is already N-1
.word 4
mult $2, $4
mflo $5        ; 4(N-1)
add $5, $1, $5 ; address of a[N-1]
lw $5, 0($5)   ; a[N-1]

add $3, $3, $5 ; Sum(a,N-1) + a[N-1]

end:
; pop all registers we pushed
lis $4
.word 16
add $30, $30, $4
lw $2, -4($30)
lw $4, -8($30)
lw $5, -12($30)
lw $31,-16($30)
jr $31
```