3 Variables

In machine language, programs can store data in the registers and memory locations provided by the hardware. When writing programs to solve real-world problems, we may not care where a particular piece of data is physically stored; we just want an abstraction of a piece of memory sufficiently large to store some value. *Variables* are such an abstraction, which we will implement in Assignment 3 as we start moving from assembly language towards a high-level language.

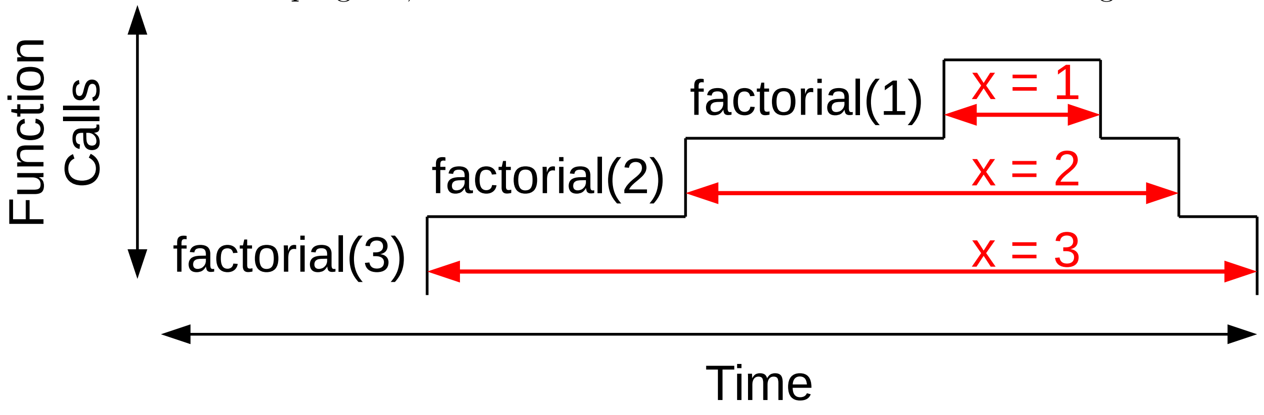
3.1 Variable instances

To begin thinking about how to implement variables, consider the following short Scala procedure that computes the factorial of a number:

```scala
def factorial(x: Int): Int =
  if(x < 2) 1 else x * factorial(x - 1)
```

This procedure defines one variable named `x`. When we call the procedure on an argument, for example 3, the variable `x` takes on the values 3, 2, 1. But notice that this procedure does not contain any assignment statements (`x = ...`). How can the variable `x` take on three different values when the program runs, when it is never assigned a new value?

During the execution of this procedure on the argument 3, there are three *instances* of the variable `x`. When `factorial` is first called, the first instance comes into existence, and has the value 3. When `factorial` makes a recursive call to itself, a second instance of `x` comes into existence, with the value 2. Finally, at the second recursive call, a third instance of `x` comes into existence, with the value 1. This is illustrated in the diagram below. There is *one* variable `x` in the program, but there are *three* instances of that variable during execution.



Consider again that each of the three instances of `x` has a different value. This means that we need to use a *different* memory location or register to implement each *instance* of a variable; otherwise, the values of different instances would overwrite each other.

3.2 Extents

We say that the *extent* of a variable instance is the time interval during program execution when the variable can be accessed. In the diagram above, the red arrows indicate the extent of each of the instances of variable `x`. How we implement a variable depends on its extent:

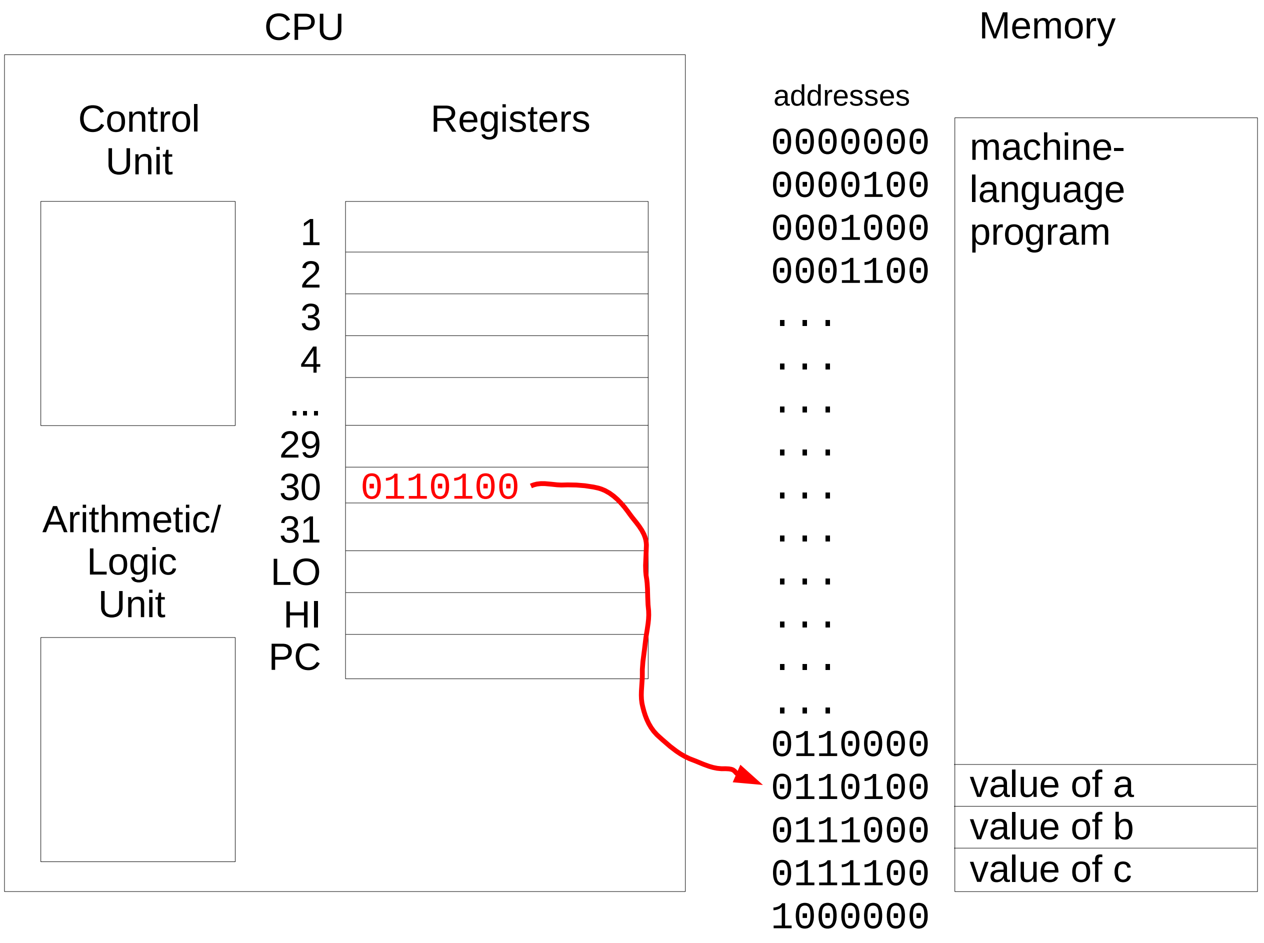
- When a variable is *local* to a procedure, like `x` in the `factorial` procedure, the extent of an instance of the variable ranges from when the procedure is called to when it returns. Since procedure calls and returns follow a last-in-first-out order (the last called procedure is the first to return), it is natural to use a *stack* data structure to implement local variables.
- A *global* variable has only one instance and its extent is the entire duration of the execution of the program. It is natural to implement a global variable with a *fixed memory address* or register.
- The field of an object or record can also be considered as a kind of variable. The extent of such a variable begins when the object or record is created and ends when it is freed, either explicitly by the program, or automatically when a garbage collector determines that the variable will no longer be used in the future. Since the extent of such a variable is arbitrary and determined at runtime, it is natural to implement such a variable using a *heap* data structure, which can allocate and recycle areas of memory in an arbitrary order.

For now, we will implement stack-allocated procedure-local variables and leave heap-allocated variables until later in the course. To support variables, we need to implement a stack data structure in the computer.

3.3 Implementing a stack

Recall that the hardware provides memory as an unstructured array of bits grouped into words. A stack is a simple data structure that can be implemented easily in terms of such an array. Specifically, we lay out the words on the stack in consecutive memory locations, and keep track of one word, the address of the top word on the stack. This address is called the *stack pointer*. To push a word w onto the stack, we subtract 4 from the stack pointer so that it contains the address of an unused word, and store w at the memory address in the new stack pointer. To pop a word from the stack, we load the word from the memory address given by the stack pointer (this is the word at the top of the stack), and add 4 to the stack pointer so that it contains the address of the next word on the stack. By convention, we will use register 30 to store the stack pointer.

This is illustrated in the following diagram:



The stack at the end of memory contains three words, the values of variables a, b, and c, with a at the top of the stack. The stack pointer, register 30, contains `0110100`, the memory address of the top of the stack. (Although addresses and register contents are 32 bits wide, they have been shortened to 7 bits to fit in the diagram.)

Under this convention, a word that is in register 1 can be pushed onto the stack using the assembly language code:

```
    LIS(Reg.result)
    Word(encodeSigned(4))
    SUB(Reg.stackPointer, Reg.stackPointer, Reg.result)
    SW(Reg(1), 0, Reg.stackPointer)
```

A word can be popped from the top of the stack into register 1 using the assembly language code:

```
    LW(Reg(1), 0, Reg.stackPointer)
    LIS(Reg.result)
    Word(encodeSigned(4))
    ADD(Reg.stackPointer, Reg.stackPointer, Reg.result)
```

Conveniently, our MIPS computer initializes register 30 to the size of memory, one word past the address of the last word of memory, when it begins executing a program. This means that we do not need to initialize the stack pointer with a starting address if we want the stack to be stored at the end of memory.

Remark. The assembly language code above uses symbolic constant names for registers, such as `Reg.result` for register 3 and `Reg.stackPointer` for register 30. These names document the purpose for which we will use each register throughout the course. The full list of names is defined in the file `Reg.scala` in the handout code. These names are just Scala shorthands for the register numbers.

Remark. In our convention, the stack grows towards lower memory addresses: we subtract from the stack pointer to push and add to it to pop. It would also be possible to implement the stack in the opposite direction. Growing the stack from the end of memory towards lower memory addresses is convenient because the code of our program grows from the beginning of memory towards higher memory addresses: this way, we do not have to think about the address at which the program should end and the stack should start.

3.4 Stack frames and the frame pointer

We will generally push enough space onto the stack to store all of the local variables of a procedure. This area of the stack reserved for the variable instances of an invocation of a procedure is called a *stack frame* or an *activation record*.

When the compiled program is running, it will often need to access variables in the stack frame, so the location of the stack frame in memory needs to be known at run time. It is tempting to look for the memory address of the stack frame in the stack pointer: after all, it holds the address of the last thing that was pushed on the stack. This would prevent us, however, from using the stack for any other purpose during the execution of the procedure, since any such use of the stack would modify the stack pointer, losing the address of the stack frame. We therefore make a copy of the address of the stack frame at the beginning of a procedure, copying the stack pointer into another register called the *frame pointer*. The frame pointer holds the address of the stack frame throughout the execution of the procedure. This frees up the stack pointer so that the stack can be used for other purposes within the procedure, such as storing temporary values. By convention, we will use register 29 as the frame pointer throughout the course.

The layout of the stack frame of a procedure is generally fixed at compile time. A compiler keeps track of all of the local variables in a procedure and assigns to each variable a unique offset from the frame pointer. The compiler maintains a symbol table that maps each variable to its offset. The memory address of each variable is calculated by adding the offset of the variable to the address in the frame pointer. To generate code that reads or writes the value of a variable, the compiler outputs code that performs this address calculation. The form of the `LW` and `SW` instructions in MIPS machine language makes this convenient: these instructions access memory at an address that is computed by adding a constant offset to the address in a specified register. For example, we can read and write a word at offset 8 from the address in the frame pointer using the instructions:

```
    LW(Reg(1), 8, Reg.framePointer)
    SW(Reg(1), 8, Reg.framePointer)
```

To summarize:

- The *stack pointer* is a register that, at *run time*, holds the address of the top word on the stack.
- The *frame pointer* is a register that, at *run time*, indicates the location in memory of the frame of the currently executing procedure. Variables are accessed at constant offsets from the address in the frame pointer.
- The *symbol table* of each procedure is a data structure that exists at *compile time*. It maps each local variable in the procedure to a constant offset from the frame pointer.

3.5 Chunks

In the assignments in this course, we will never push or pop an individual word on the stack at a time. Instead, we will organize all memory in *chunks*, collections of some known number of words stored at consecutive memory locations. The reasons for this choice will be explained later in the course. For now, each chunk will store one frame – the local variables of some procedure. Later in the course, we will also store data other than frames in chunks.

The following diagram shows the layout of an example chunk in memory at run time:

offset	
0	size
4	reserved for A11
8	a
12	b
16	c

The first word of each chunk always stores the total size of the chunk in bytes. The second word of each chunk is reserved for now: it will be used for Assignment 11. The remaining words contain arbitrary data to be stored in the chunk, such as the values of the local variables when the chunk implements the frame of a procedure. In this example, the chunk holds the values of three variables, a, b, and c, of some procedure.

The compile time code for keeping track of the layout of a chunk is in the `Chunk` class in the `MemoryManagement.scala` file in the handout code. In particular, each Scala `Chunk` object contains a symbol table that maps variables (instances of the Scala `Variable` class) to their offsets from the beginning of the chunk. The compiler can ask the Scala `Chunk` object to generate assembly language code to load or store the value of the word in the chunk associated with the variable, given a *base register* that will, at run time, hold the address of the beginning of the chunk.

Remark. Since local variables are stored at offsets from the frame pointer, when generating code to access a local variable, the base register holding the address of the beginning of the frame is the frame pointer.

Remark. The offset and the identity of the base register are known at *compile time*, and are hard coded in the generated `LW` or `SW` instruction. The value of the base register (i.e., the address of the beginning of the chunk) is known only at *run time*.

In Assignment 3, you will implement the missing parts of the `Chunk` class, add support to the `Stack` object in `MemoryManagement.scala` to push and pop a chunk of words on the stack, and compile reads and writes of variables to assembly language code.