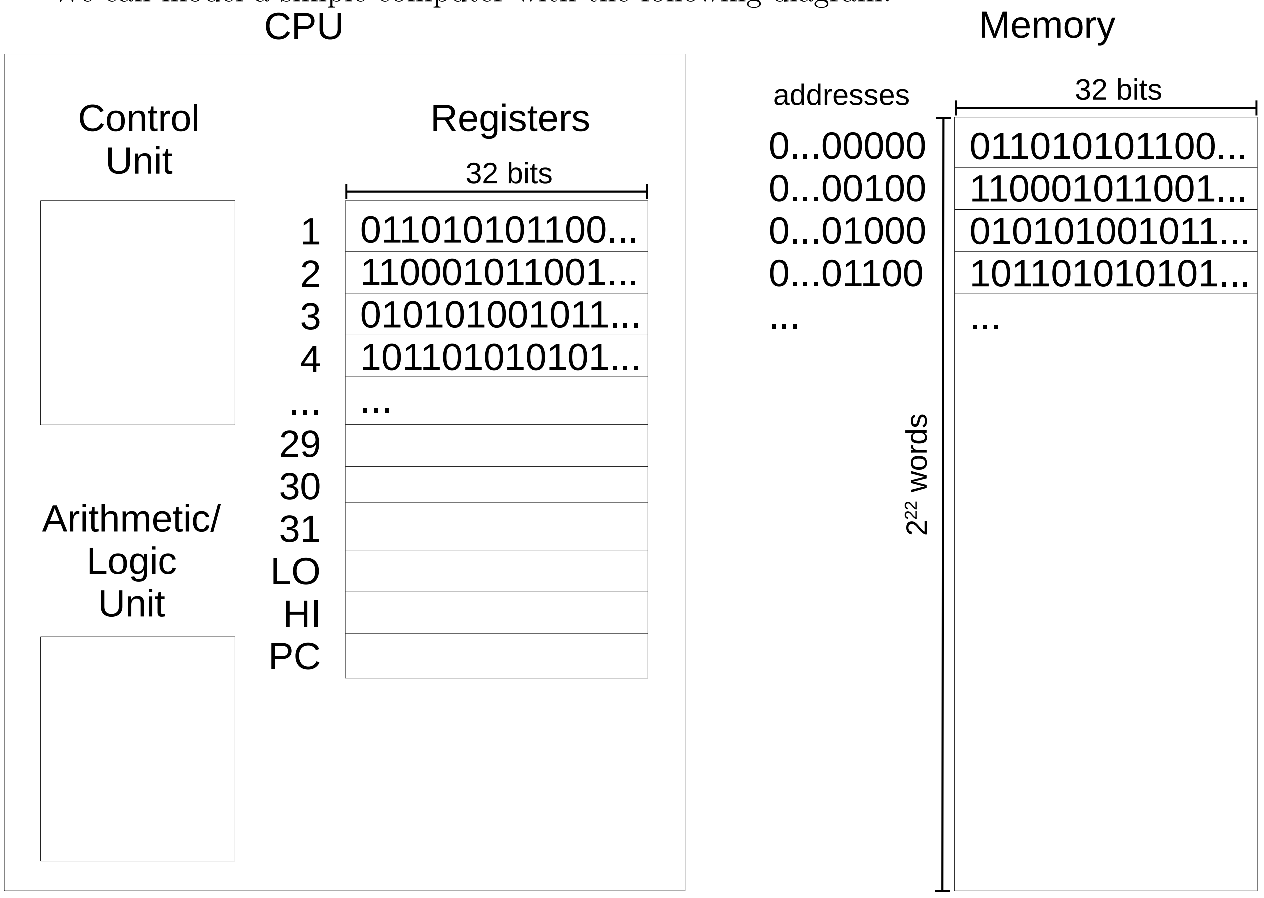
2 The MIPS computer

Recall from the previous module that a computer is a machine for manipulating bits. We will start the course with such a computer and gradually build up abstractions to represent higher-level concepts.

We can model a simple computer with the following diagram.



On the right is the *memory*: in our case, an array of 2^{27} bits, zeros and ones. For convenience, we will group the bits into 2^{22} *words* of 32 bits each. We will also label each word in memory with an *address*, a sequence of 32 bits. In our case, every valid address starts with 8 zero bits, followed by 22 bits that identify one of the 2^{22} words in memory, followed by 2 more zero bits. Therefore, each address, when interpreted as a binary number, is a multiple of four between 0 and $2^{24} - 4$ inclusive, and we will often write addresses as numbers, because writing out the 32 zeros and ones of an address is tedious. The reasons for these conventions are historical: early CPUs has words of 8 bits numbered sequentially; since one of our words is four times as wide, our addresses count by fours.

On the left is the Central Processing Unit (*CPU*) (or processor). The CPU also contains a smaller amount of memory called registers, in our case 34 registers of 32 bits each. In addition, the CPU contains a control unit and an arithmetic logic unit (ALU).

At any given time, the *state* of the computer are the values of all the bits in the registers and the memory. That is, a state is a sequence of (in our case) $2^{27} + 34 \times 32$ ones or zeros.

Together, the control unit and the ALU implement a function (in the mathematical sense of a function) that maps each possible state to some new state. We will call this function *step*. Over time, this function is applied repeatedly, many times per second, to update the value of the memory and registers. For example, if the vector of bits m represents the state of the computer right now, then in three clock cycles, after three state updates, the state of the computer will be $step(step(step(m)))$.

2.1 Stored program architecture

When designing a computer, how should we design the step function that it will implement? We would like the step function to solve whichever high-level problem we have decided to solve with the computer, such as performing calculations to predict the weather or rendering a website. If we encode the input to the problem using bits (according to some convention we have chosen), we want the step function to be such that if we apply it many times to those bits, decoding the resulting bits will yield the desired output, a solution to the problem. Thus, the step function is an encoding of the *program* that we want the computer to run.

What if we want to use the computer to solve a different problem, to run a different program? This means changing the step function and therefore changing the hardware. Indeed, early computers had to be physically rewired to run each new program, which is obviously tedious and costly. A *stored program computer* is one that implements a step function that is *general* enough to implement arbitrary programs. In such a computer, the program to be executed is also encoded as a sequence of bits, which are stored in the memory together with the bits that represent the data being processed. The step function looks at the bits representing the program to decide how to change the bits representing the data. Thus, the program can be changed without changing the hardware. The convention used to encode the instructions of a program into a sequence of bits stored in memory to be executed by a CPU is called *machine language*. The stored program architecture is also commonly known as the *von Neumann* architecture.

Exercise 9. *Read online about John von Neumann, the history of the stored program computer, and the other people involved in its invention.*

The stored program design can be summarized in the catchphrase *code is data*, since both code and data are encoded as sequences of bits. This fundamental idea enables code to be manipulated as data; in particular, we can write programs whose inputs or outputs are other programs. Such programs that manipulate other programs are a recurring theme throughout this course.

2.2 A general-purpose step function

Let's try to design such a general-purpose step function for a stored program computer. We will write the program to be executed as a sequence of instructions. We will encode each instruction as a word, 32 bits, so the sequence of instructions can be encoded as a sequence of words, stored at successive locations in memory. The step function will then read each word/instruction from memory, execute it, then move on to the next word. To keep track of our current place in the program, the instruction that we are currently executing, we will reserve one register, called the program counter (PC). The PC stores the address of the word in memory that represents the next instruction to be executed. Following this design, we can define the step function that the processor should implement as follows, in a Scala-like pseudocode:

```
1 def step(state: State): State = {
2   val instruction = state.memory( state.register(PC) )
3   val state2 = state.setRegister( PC, state.register(PC) + 4 )
4   instruction match {
5     ...
6   }
7 }
```

The function takes a state as parameter and returns a state. In line 2, the function reads the address of the instruction to be executed from the program counter (PC) register and then *fetches* (reads) the word representing the instruction from that address in memory. In line 3, the function *increments* the program counter by 4, so that it refers to the address of the next word in memory, representing the instruction to be executed next. Finally, the match expression in lines 4 to 6 looks at the contents of the current instruction word, and evaluates to the new state of the computer that results from *executing* the instruction. These three operations, which the CPU performs repeatedly, are called the *fetch-increment-execute cycle*. This design of the step function enables the CPU to run different programs by interpreting sequences of instructions that have been encoded as sequences of words stored in memory.

2.3 MIPS machine language instructions

Of the many different kinds of CPUs that have been produced, each kind provides a different set of instructions and encodes them differently in binary. In this course, we will use an instruction set from the MIPS family of CPUs. Nowadays, CPUs from the MIPS family are used mainly in embedded systems such as networking equipment and home appliances. In earlier years, they were used in high-end desktop workstations and servers, video game consoles, and supercomputers. We use the MIPS family in the course due to the relatively simple and regular design of its instruction set, but the general concepts that we will learn carry over to other families of processors.

An emulator of a MIPS computer written in Scala is provided in the handout code for the course. Have a look at the handout code now. The file `State.scala` defines the state of the computer, with operations to read and write a word into a particular register or memory address. The file `CPU.scala` defines the functioning of the CPU, in particular the `step` function and all of the instructions that it implements. This definition will be our canonical specification of the instruction set and the meaning of each instruction; it is also the implementation that Marmoset uses to actually run your machine language programs. As a secondary reference, the course website also provides a MIPS Reference Sheetthat briefly and less precisely summarizes the available instructions. Look at `CPU.scala` and the MIPS Reference Sheetnow as you read the summary of the instructions.

The instruction to be executed is identified by the first six bits of the instruction word: for example, for the Branch On Equal (beq) instruction, these first six bits are 000100. When these first six bits are all zeros, the instruction is further identified by the last six bits: for example, for the Add instruction, the first six bits are zeros and the last six bits are 100000.

Instructions that start with six zeros are in the Register format: after the six zeros, the next five bits of the instruction identify a register number s, the following five bits identify a second register number t, and the five bits after that identify a third register number d. Generally, these instructions read the values from the two registers numbered s and t, interpret them as binary numbers, perform some arithmetic operation on these, then store the resulting value in the register numbered d.

Other instructions are in the Immediate format: after the first six bits and the numbers s and t of two registers, the remaining 16 bits of the instruction specify a constant number i encoded in two's complement. For example, the Branch On Equal (beq) instruction uses this constant to determine by how many instructions to skip ahead or back to find the instruction that will execute next: it compares the values of registers s and t, and if they are equal, it adds the constant $i \times 4$ to the program counter register.

The **Add** and **Subtract** instructions add/subtract the numbers in registers s and t and place the result in register d. Arithmetic is performed modulo 2^{32}, so the same instruction can be used regardless of whether the bits in these registers represent an unsigned number in binary or a signed number in two's complement, because if $a \equiv a' \pmod{2^{32}}$ and $b \equiv b' \pmod{2^{32}}$, then $a + b \equiv a' + b' \pmod{2^{32}}$.

You may have noticed that the registers are numbered from 1 to 31 and there is no register 0. This is because 0 is such a commonly used number that the MIPS instruction set has special support for it: if an instruction refers to register 0, execution of the instruction will not attempt to read from the non-existent *register* 0, but will instead provide the *constant value* 0. This enables us, for example, to copy a value from register s to register d by specifying register t to be 0; in this case, the CPU will read the value of register s, add the constant 0 to it, and write the resulting value to register d. When the destination register d in an instruction is specified to be 0, the instruction does not write to any register.

You might expect to need only a single **Multiply** instruction to perform multiplication modulo 2^{32} and be usable for multiplying both unsigned binary and signed two's complement numbers, because if $a \equiv a' \pmod{2^{32}}$ and $b \equiv b' \pmod{2^{32}}$, then $a \times b \equiv a' \times b' \pmod{2^{32}}$. However, the MIPS CPU performs multiplication modulo 2^{64}, not 2^{32}, and it is not true that if $a \equiv a' \pmod{2^{32}}$ and $b \equiv b' \pmod{2^{32}}$, then $a \times b \equiv a' \times b' \pmod{\mathbf{2^{64}}}$. The **Multiply** instruction interprets the values in registers s and t as two's complement numbers, multiplies them, encodes the result as a 64-bit two's complement number, then stores the first (most-significant) 32 bits in the register named HI, and the remaining (least-significant) 32 bits in the register named LO. The **Multiply Unsigned** instruction does the same, except that it interprets the bits in registers s and t as unsigned binary numbers rather than signed two's complement numbers. The values of registers HI and LO can be copied into one of the numbered registers using the **Move From High/Remainder** and **Move From Low/Quotient** instructions. In both cases, the 32 least-significant bits that are written into the LO register represent the product of the multiplication modulo 2^{32}. Therefore, when multiplying numbers that are small enough that their product fits in 32 bits, if we read only the LO register and ignore the value of the HI register, the **Multiply** and **Multiply Unsigned** instructions have the same effect.

Exercise 10. *Find a counterexample to show that it is not true that if $a \equiv a' \pmod{2^{32}}$ and $b \equiv b' \pmod{2^{32}}$, then $a \times b \equiv a' \times b' \pmod{2^{64}}$.*

The **Divide** and **Divide Unsigned** instructions perform integer division, again interpreting the values in registers s and t as encoding integers in signed two's complement or unsigned binary, respectively. The integer part of the quotient is stored in the LO register and the remainder is stored in the HI register. Two separate instructions are definitely needed here, because it is generally not true that if $a \equiv a' \pmod{m}$ and $b \equiv b' \pmod{m}$, then $\frac{a}{b} \equiv \frac{a'}{b'} \pmod{m}$.

The **Load Immediate And Skip** instruction, operationally, reads the value at the memory address given by the program counter (PC) register into register d, then increments the program counter by 4. The practical application of this instruction is to load a 32-bit constant word into a register. To do so, we include in the program the word representing the Load Immediate And Skip instruction, followed immediately by the 32-bit constant word to be loaded into a register. Recall the fetch-increment-execute cycle, which increments the program counter by 4 immediately before every instruction is executed. When the Load Immediate And Skip instruction executes, the program counter will have already been incremented to the next memory address, the address of the constant word to be loaded into a register. After loading the constant word into the specified register, the Load Immediate And Skip instruction increments the program counter by 4 again, to ensure that execution continues with the word after the constant word that was loaded.

Exercise 11. *What would go wrong if the Load Immediate And Skip instruction did not increment the program counter by 4 after loading the constant word into a register?*

The **Load Word** instruction copies a word from an address in memory into register t; the **Store Word** instruction does the opposite, in that it copies a word from register t into an address in memory. In both instructions, the memory address is computed by taking the value in register s and adding to it the constant i that is included as part of the instruction. To set the address to exactly the value of register s, the constant i can be set to 0. The inclusion of the constant in the address calculation makes it easy to access memory addresses at fixed constant offsets from some address held in register s.

The **Set Less Than** and **Set Less Than Unsigned** instructions compare the values in registers s and t, and set register d to 1 if the value of register s is less than the value of register t, and to 0 otherwise. Two instructions are needed because it is generally not true that if $a \equiv a' \pmod{2^{32}}$ and $b \equiv b' \pmod{2^{32}}$, then $a < b \iff a' < b'$.

The **Branch On Equal** and **Branch On Not Equal** instructions compare the values of registers s and t. If the values are equal (respectively not equal), the instruction adds the constant $i \times 4$ to the program counter, so that execution of the program proceeds i words (instructions) after the next instruction (or before the next instruction if i is negative).

The **Jump Register** copies the value of register s into the program counter (PC) register. The next instruction that will execute after the Jump Register instruction will be the instruction whose address was in register s. The **Jump And Link Register** instruction does the same thing, but in addition, it saves the previous value of the program counter into register 31. Therefore, after performing a Jump And Link Register instruction, executing a Jump Register 31 instruction will restore the program counter back to the word immediately after the Jump And Link Register instruction. This instruction will later become our basic building block for implementing procedure calls, since a procedure must return to the point that it was called from when it finishes executing.

2.4 Writing machine language programs

In the assignments in this course, starting with Assignment 1, we will execute MIPS machine language code on a simulated computer. This is done by creating a sequence of words (`Seq[Word]`) in a Scala program, loading it into memory using the methods of the `State` class, and calling the run method of the simulated CPU on that memory state. An example of how this is done is in the `loadAndRun` method in `A1.scala` in the handout code.

As you will quickly discover while implementing Assignment 1, writing machine language code by hand is tedious. Since code is data, instead of writing machine language code directly, we could instead write a program that will write machine language code for us automatically based on a higher-level description of the program to be implemented – a compiler. It will take us until the end of the course to finish the compiler, but we want to write programs right away, so we will add abstractions *gradually* as we build up to a full compiler.

2.5 Assembly language

Our first simple but practical abstraction are *opcodes*: symbolic names for machine language instructions. In Assignment 1, you will write Scala functions such as:

```scala
def ADD(...) = Word(...)
```

After you write a full set of such functions for all the machine language instructions, you will be able to generate machine language programs automatically by writing a sequence of calls to these functions. For example, instead of having to write:

```scala
Seq(
  Word("0000 0000 0010 0010 0001 1000 0010 0000"),
  Word("0000 0011 1110 0000 0000 0000 0000 1000")
)
```

you will be able to write:

```scala
Seq(
  ADD(Reg(3), Reg(1), Reg(2)),
  JR(Reg(31))
)
```

Although both pieces of Scala code result in the same sequence of words of bits, and therefore the same machine language program, the latter is easier to read and write. A language that uses opcodes to designate the bit sequences of machine language instructions is called *assembly language*, and a program that translates from assembly language to machine language is called an *assembler*. Despite the convenience of using opcodes, each instruction in an assembly language program corresponds directly to an instruction in the resulting machine language program.

Opcodes are our first abstraction.

2.6 Labels

Most assembly languages provide a second abstraction: labels. To motivate labels, consider the following exercise:

Exercise 12. *Write an assembly language program that takes as input an integer encoded in two's complement in register 1, finds the absolute value of the integer and places it in register 1, and then jumps to the address in register 31.*

A solution is revealed on the next page. Try to do the exercise yourself before turning the page.

We might solve the exercise as follows:

```
17 Seq(
18   SLT(Reg(2), Reg(1), Reg(0)),
19   BEQ(Reg(2), Reg(0), 1),
20   SUB(Reg(1), Reg(0), Reg(1)),
21   JR(Reg(31))
22 )
```

The SLT instruction tests whether the number is less than zero. If it is, the SUB instruction negates it by subtracting it from 0. The BEQ instruction skips executing the SUB instruction when the number is non-negative (if the SLT instruction caused register 2 to become 0). In the BEQ instruction, we have to manually specify the offset 1: to skip one instruction when the number is non-negative. In this small example, it is easy to see that we want to skip one instruction, but in a larger program, we may wish to skip ahead or back by many instructions, and it would be tedious to have to count them all, and to count them again every time we change the program to add or remove instructions.

Just as we used opcodes to give names to bit strings representing instructions, we should give names to locations in the program. These location names are called *labels*. More precisely, a label is a name that represents a memory address. Typically, the memory address contains some designated instruction of the program. Labels are associated with two operations:

- *defining* a label in an assembly language program associates the name with the memory address at which the assembly language instruction after the label will be placed;
- *using* a label inserts the memory address corresponding to the label in the machine language code.

Once our assembly language has labels, we can write the previous program as follows:

```
23 val label = new Label("label")
24 Seq(
25   SLT(Reg(2), Reg(1), Reg(0)),
26   beq(Reg(2), Reg(0), label),
27   SUB(Reg(1), Reg(0), Reg(1)),
28   Define(label),
29   JR(Reg(31))
30 )
```

The line `Define(label)` associates the label `label` with the memory address of the next instruction, the memory address at which the machine language representation of `JR(Reg(31))` will appear. We *use* the label in the `beq` line: instead of writing the offset 1, we write the label `label` to say that when the number is non-negative, execution should skip to the `JR(Reg(31))` instruction. The lowercase `beq` instead of `BEQ` is used only to distinguish the Scala function that takes a numeric offset from the one that takes a label as argument.

The `new Label` expression in Line 23 is a Scala expression that creates a new label that can be used in the assembly language program. Each Scala execution of `new Label` creates a new and unique label. The textual name `"label"` is used only for debugging purposes and does not influence the identity of the label. That is, `new Label("label") != new Label("label")`; these two expressions create two distinct labels.

2.7 Implementing labels

Like opcodes, labels are a feature of only the assembly language program, and are eliminated when machine language is generated. The assembler must keep track of the memory address associated with each label. When the assembler generates machine language, it must translate each label to the associated address, and output the address into the machine language code that it is generating. In the case of a branch instruction such as BEQ, which specifies a branch *offset* to be added to the program counter, the assembler must consider the memory address associated with the label and the memory address at which the branch instruction will appear, compute the correct offset from these two addresses, and output the offset as part of the machine language instruction. You will implement this calculation to add support for labels in Assignment 2.

In Assignment 1, a machine language program was represented by the Scala data type `Seq[Word]`, a sequence of words of 32 bits each. To represent assembly language programs, we need more than just words: we additionally need label definitions and uses. Later throughout the course, we will add even more abstractions to our programs. From now on, we will represent programs using the Scala data type `Seq[Code]`. When representing assembly language, a `Code` can be a `Word`, but it can also be a `Define`, a `BeqBneLabel`, or a `Use`:

- A `Define` represents the definition of a label. Since labels are not present in machine language code, a `Define` in the assembly language program does not generate any words in the machine language code.
- A `Use` represents the use of a label. When generating machine language code, the assembler should output a word containing the address associated with the label in the `Use`.
- A `BeqBneLabel` represents a BEQ or BNE instruction with a label rather than an explicit offset. When generating machine language code, the assembler should compute the correct offset based on the label and output it as part of a BEQ or BNE instruction.

We will add more kinds of `Code` as the course progresses. Within Assignment 2, we already add `Comment` and `Block` as new kinds of `Code`. We will progressively add more abstractions to our program representation, so that it gradually moves from an assembly language to a high-level programming language. Our assembler will gradually become a compiler. For each new kind of `Code` that we add, we will implement a function in the assembler/compiler to eliminate that `Code` by rewriting it in terms of the more primitive kinds of `Code` introduced earlier. For example, in Assignment 2, you will implement `eliminateLabels`, `eliminateComments`, and `eliminateBlocks`. After running all three of these functions on a `Code`, all of the kinds of `Code` other than `Words` are eliminated, and the assembly language program has turned into a machine language program.

How does an assembler eliminate labels from an assembly language program, translating them to memory addresses and offsets? Two passes over the code are necessary because the use of a label may occur before or after its definition:

1. In the first pass, the assembler determines the memory address of each instruction in the assembly language program and determines the memory address of each label definition. The assembler builds a *symbol table*. In general, a symbol table is a map from names (in this case labels) to their meanings (in this case memory addresses). We will see other symbol tables for other kinds of names later in the course.

2. In the second pass, the assembler translates each assembly language instruction into the corresponding machine language word and translates each use of a label into the corresponding memory address or offset. In this pass, the assembler uses the symbol table to look up the address denoted by each label.

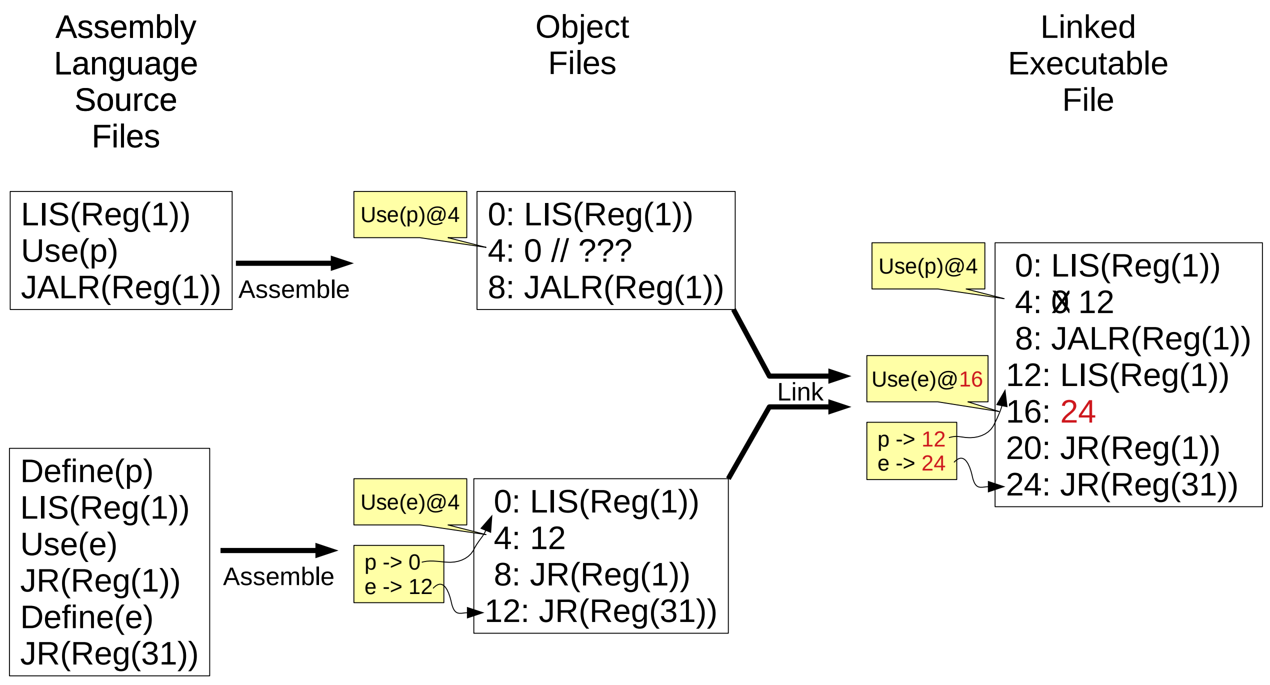
You will implement these two passes of an assembler in the `eliminateLabels` function in Assignment 2.

Exercise 13. *In the regular CS 241, students write many significant assembly language programs by hand. This is good practice to familiarize yourself with the assembly language. In CS 241E, we will mostly write Scala code that writes such assembly language programs for us. This more abstract approach is less tedious but requires you to think about the assembly language programs that your code generator is generating. Throughout the course, if you find this challenging, write some assembly language programs by hand for practice.*

2.8 Linking and Relocation

Recall that labels are a feature of assembly language and are translated to explicit memory addresses when the program is assembled to machine language. Often, programs grow large enough that it is inconvenient to assemble (or compile) the whole program at once. Think about a large company working on software written by hundreds or thousands of developers. In practice, we often want to assemble and compile small parts of a program, typically one source file at a time.

In this section, we will work through a running example illustrated in the following diagram.



At the left of the diagram are two assembly language source files. The code in these files is for illustrative purposes; it is not intended to do anything useful. The code in the first file loads the address represented by a label `p` into register 1, then uses the `JALR` instruction to transfer control to that address, retaining the old value of the program counter in register 31. The code in the second file defines label `p`, loads the address represented by label `e` into register 1, transfers control to that address, then defines label `e`, and finally transfers control to the address in register 31. We can think of this code as a simple example of a procedure call: the first file contains a call to a procedure starting at label `p`; the second file defines the procedure `p`, which executes some instructions and returns back to the calling code using the `JR(Reg(31))` instruction.

Ignoring the middle of the diagram for a moment, if we were to concatenate these two assembly language files into a single source file and pass it through an assembler, we would get the machine language code in the file on the left. Note that the diagram is slightly inaccurate here for readability: machine language code is binary, zeros and ones, but is hard to read; instead, the diagram shows the assembly language instruction corresponding to each word that is actually in the machine language file. The number before the colon on each line is the position of that word from the beginning of the file, and corresponds to the memory address of each word when the machine language program is loaded into memory.

Observe that the assembler has replaced the uses of the labels `p` and `e` with their values 12 and 24, the memory addresses at which those labels are defined.

Now consider what happens if we assemble the two assembly language files separately, without first concatenating them into a single source file. The assembler will generate the two machine language files in the middle of the diagram.

When assembling the first file, the assembler will not have any value for `p` in its symbol table, since `p` is not defined in the first file. Even if we suppress the undefined label error that the assembler would normally fail with, the assembler cannot know the constant address to which it should translate the line `Use(p)`. Therefore, we just make the assembler output the address 0, to be filled in later when the correct address of label p becomes known. The assembler adds a note (shown in yellow) to record that there was a use of label p at offset 4 in the output file. Such notes are called *metadata*. An *object file* contains machine language code together with an encoding of such metadata.

The only label used in the second file is e, and it is also defined there, so the assembler can translate it to the constant address 12. As part of the metadata in the second object file, the assembler records the symbol table, which says that label p is defined at offset 0 and label e is defined at offset 12. The metadata for the first object file should also include a symbol table, but the symbol table is empty in this example because the first assembly language file does not define any labels. The metadata in the second file also records that label e was used at offset 4. This seems redundant, since the assembler knows that the value of label e is 12 and replaces the use of the label with its value, but we will see later that this metadata is necessary for relocation.

The overall purpose of the metadata in the object files is to enable the two object files to later be *linked* together into the machine language file on the right. In other words, by assembling two source files into two object files and linking the object files, we wish to obtain the same machine language code as if we had concatenated the two source files and assembled them together as a single file. Assembling to machine language loses some information, such as the values of labels and knowledge of where they were used, so this information is recorded in the metadata that accompanies the machine language code in the object file. During linking, the symbol tables from the object file metadata are used to fix up the uses of labels. In our example, the assembler did not know the value of label p when assembling the first file, so it translated it to 0; the linker will find the value of p in the symbol table encoded in the metadata in the second file and use it to replace the 0 with the correct value.

If the linker just concatenates the machine language code in the two object files, the memory addresses will be incorrect. The words in both object files are numbered starting at offset 0, but in the resulting linked file, one object file must come before the other, so only one of the object files can start at offset 0. In the example in the diagram, the second file comes after the first file, so in the executable file on the right of the diagram, the words from the second object file start at offset 12, not offset 0. This means that any addresses encoded as constants in the machine language code are incorrect. For example, the Use(e) in the assembly language code of the second file is encoded as the constant 12 in the object file, because the label e is defined at offset 12 in that file, the location of JR(Reg(31)). But in the executable file on the right, the JR(Reg(31)) instruction is at offset 24!

In order for the resulting machine language code to work correctly, the constant 12 corresponding to the use of label e needs to be adjusted to 24. This process of adjusting memory addresses to account for machine language code starting at an address other than 0 is called *relocation*, and it is a necessary part of linking. In the example, all addresses in the second object file need to be relocated by adding 12 to them, because the machine language code of the second object file will start at address 12 rather than 0.

The need to relocate the use of e is the reason why the assembler has to include the seemingly redundant metadata that the constant 12 at offset 4 in the second object file came from a use of e. Without this metadata, there would be no way for the linker to distinguish a 12 that is an address that came from the use of a label, which needs to be relocated, and a 12 that is just the number 12 appearing as a constant in the source program, which should not be relocated. The metadata tells the linker which words represent addresses and therefore should be relocated.

The linker needs to relocate addresses not only in the machine language code, but also in the metadata. For example, the symbol table in the second object file maps the label `p` to the value 0. However, in the linked executable file, the word from offset 0 of the second object file is at offset 12. The use of `p` at offset 4 in the first object file should therefore be translated to the constant address 12, not the constant address 0. This is achieved by relocating the addresses in the metadata. Relocated addresses are shown as red numbers in the diagram. After we add 12 to all addresses in the second object file, including those in the metadata, all addresses refer to the correct offsets in the linked executable file: label `p` has value 12 there, and label `e` has value 24. Therefore, the placeholder 0 for the use of label `p` at offset 4 in the first object file is replaced by the correct address 12, not the incorrect address 0 from the symbol table of the second object file.

In summary, *linking* is the process of combining multiple object files into one. It involves two main activities:

- *relocating* addresses in object files to account for the code from those files starting at an offset other than 0 in the resulting linked file, and
- *resolving* labels that are used in a different file than the one in which they are defined, by finding the value of the label in the symbol table metadata of the defining file and writing the value in all the places where the symbol is used.