Garcia, Nikolic
Fall 2020

CS 61C Machine Structures

Notes

Contact for corrections: Caroline Liu (carolinexwliu89@berkeley.edu)

Disclaimer: These notes are still in beta and haven't been thoroughly fact-checked. In any factual dispute, all other course material takes precedence. Any feedback is welcome.

Introduction to RISC-V

RISC-V is one of the languages with which we'll become familiar in CS61C —we'll be exploring its intricacies throughout project 2 (CS61Classify), the next few labs, and future modules as well, which include but are not limited to the single cycle datapath and pipelining sections. Although you will not see RISC-V explicitly in most of your upper div classes, the skills and topics covered will be expanded upon in common upper divs like CS161, CS162, and more. (For anyone interested more in architecture and have a forbidden love with RISC-V, EECS151 and CS152 explore this side of Berkeley EECS in depth.)

A Brief History of RISC-V

First coined (and started) by David Patterson (yes, one of the authors of your favourite textbook) and his team in the 1980s, RISC-V is an open standard instruction set architecture (ISA) that is based upon the idea of a "reduced instruction set computer" (RISC) principles. The V comes from the fact that the current version of RISC-V is the fifth iteration of the ISA. Part of what sets RISC architecture apart from other ISA designs is the fact that it's provided under open-source licenses (i.e. it's free to use). Not only that, but its design is both extremely flexible and inflexible —there are a base set of 6 instruction types upon which all 32-bit build up. This allows for a limited number of instructions to be made, but it also allows for simplicity of hardware. Part of RISC's appeal **is** the simplicity —the less variation the hardware needs to resolve, the better it is, both for the hardware designers as well as the hardware itself. This does force a constraint on the software side though —you've seen this happen in lecture and will continue to see it appear in labs and projects; tasks that require 1-2 lines in any other higher level programming language, whether it be C or Python can translate into multi-line processes (think about list comprehensions in python and how you would implement that).

All of this seems relatively counterintuitive, after all, why wouldn't you just want to create an instruction for everything you would want to do? Why **can't** we have subtraction instructions? Or comparisons for various values? Why can't we have a single instruction accomplish what would would have to do painstakingly one by one in RISC (or other low-level operations)? And those are totally vaoid questions!

RISC vs CISC

That's the entire premise behind CISC, or **complex instruction set computers**. The definition of CISC architecture is the "ability of a single instruction to execute several low-level operations (such as a load from memory, an arithmetic operation, and a memory store) or to be able to conduct multi-step operations or addressing modes within single instructions" [1]. These are generally more common in production (and probably is the basis of most electronics out there). Some microprocessor and microcontroller design families you might be more familiar with include the Intel 8080, x86 architecture (also Intel), and Intel MCS-51 (more commonly known as 8051).

In the past 40 or so years, there have been 5 iterations upon the original RISC Gold (otherwise known as RISC-I), amongst many other attempts at emulating and improving upon the original RISC family, the most famous of which is MIPS, developed by Stanford around the same time that Patterson and his team were originally pursuing the project. Nowadays, it's still in use, albeit not widely. Fun fact, CS61C used to be taught back in MIPS and only switched over to RISC-V around 2018!

There's probably one final question you're wondering, which is "if RISC-V is so good, why isn't there a RISC-VI"? The lowdown of it is, well simply put, there's no need for it (at least in the current world of microprocessors and microcontrollers). There are people who argue that "they're ready for a RISC-VI or RISC-W" but all in all, the core ISA plus the numerous extensions (standard extensions) explained here. We work with the very lowest base set of 32-bit instructions with the bare minimum because that's what satisfies our requirements in CS61C, but there are many more sets out there, which include, but are not limited to 64-bit and 128-bit instructions (imagine trying to parse through bits to translate those instructions to machine code... now you see why we stick with 32 bits?)

Assembly

Assembly in and of itself seems like a pretty nuts thing to learn, after all it seems very unlikely that you'd need to keep your assembly knowledge sharp. However, understanding how assembly works is fundamental to writing better code and understanding how the system works as a whole. We'll be exploring how assembly fits into the "compilation" process during the **CALL** (Call, **Assembly**, Linker, Loader) section of CS61C.

What's in a File?

RISC-V files (`<name>.S`) at first glance seem complicated, but they can be broken down into a few key segments, which we'll discuss at length in the following sections. But first, a few definitions:

[1] CISC Wikipedia

- **Define Statements:** we use them same as we would in C, `#define <MACRONAME> <VALUE>`, in order to help define macros to use within RISC-V code. Macros are snippets of code that have been labelled/given a name. They can range anywhere from simple values like ints to objects to even real function fragments. Because they're pre-processed (the C preprocessor transforms the code before actual compilation, i.e. C code to assembly code), the macro names can be anything you want, even C (or RISC-V keywords).
- **Directives:** in general, directives are essentially instructions that are marked in a specific way such that the compiler, or whatever other translator is being used knows how to treat a section of code. In RISC-V, we focus on 5 in particular: `.import`, `.globl`, `.data`, `.word`, and `.text`.
- **Functions:** these are the same in RISC-V as with any other language —they're marked with a label (equivalent to a function name in C), and the documentation specifies what arguments should be passed into what registers (`a0`—`a7`). The biggest difference in RISC-V functions is the fact that the system runs the instructions line by line, albeit the machine code translation. In other words, it reads the file linearly, from top to bottom unless instructions that force a change in position, which is tracked by the program counter (PC). We'll see this show up more once we start discussing the RISC-V datapath in the next few modules of the course.

Directives

There are more than three total directives that're used in RISC-V, but in CS61C, we're only concerned about 5: `.import`, `.globl`, `.data`, `.word`, and `.text`.

`.import`
The `.import` directive is essentially the same as the `#include` statements in C. For any assembly file's labels you want to use, you'd have to import the CWD's path to file as well as use the `.globl` directive as described next.

`.globl`
The `.globl` directive is to help reference labels created across multiple files. To use this directive, you must use the directive keyword, followed by a list of all function labels you want to be able to reference in other files (that has imported the file itself) delimited by commas. Note, this keyword is **not** the equivalent of creating a global variable, like in C, but rather to help "globalise" the function labels themselves, acting in complement to import statements, though they're declared in the file in which the labels are defined rather than the headers of the files in which they're referenced.

`.data`
The `.data` directive is to declare variables referring to data stored in the data (i.e. static) portion of general memory. This is essentially the equivalent of declaring global variables in C code, which as a refresher, is any variable declared outside of all functions, including main. This is **not** the same thing as the `.globl` directive as these do not refer to function

labels, but rather data values themselves. A great example of this is in lab03's `ex2.c` and `ex2.S` files —look through the two files and see what values correspond to each other!

`.word`
This directive is to help declare word-sized pieces of data corresponding to a variable name. This is particularly useful when you have larger data structures (e.g. structs, arrays, etc...) involved as well as constants you want to be able to refer to later on in your assembly code. For example, you might want to use this when you have a buffer you're trying to write to, or an error string you want to print later on, and other similar items. You can also see examples of this in lab03's `ex2.S` file. **Note: without a proper variable label, you won't be able to refer to the data later on. However, this won't result in a compilation error. This is the equivalent of dumping some data in static memory and never giving it a variable name —non-referential but also not a leak because it isn't in heap memory.**

`.text`
This directive is used before starting the functional part of RISC-V code. It divides up the pre-function code, which includes the `.import` statements, `.globl` declarations, `.data` segment, and `.word` definitions, and the function code which consists of the functional labels and instructions.

Memory (Review)

This section is very much not finished and will be added to later. It will probably be mostly in a separate set of notes.
There are 4 general portions of memory, which we'll briefly review here:

- **Stack**—also considered temporary or local memory, the stack can be seen as an (almost) unlimited buffer of memory blocks that we can use at our discretion. Starting from essentially the top of memory, every additional action we take in functions causes the stack to grown downwards, i.e. into lower memory. However, for each block we "allocate" space to, we write **upwards**. This is because naturally speaking, every additional item we add to a buffer of any kind starts at the lower end of memory and indexes upward. This is no different in the stack. The reason the initial stack grows downward is a configuration designed to not get the stack and heap memory (which we'll talk about next) meshed together. Most memory we'll be directly touching in RISC-V will be on the stack as you can't dynamically allocate any memory without system calls (which you won't need to directly make).
- **Heap**—this is considered more persistent memory. That means that without the user deliberately freeing it, the memory will not be "garbage collected". In RISC-V, just as in C, there's the potential for memory leaks and dangling pointers and whatnot. As we said earlier, there will never be a time in CS61C where you'll have to make a system call to get memory, so if the `malloc` (or similar) functions don't exist in helper files, it means you shouldn't need to dynamically allocate any memory. As in

C, the `malloc` function returns a pointer, which is just a memory address to a chunk of memory, and in order to use it, you need to index into it at the correct intervals. For arrays, this should be self-explanator—ensure you'll indexing in intervals the size of the array element. For structs, this means that you'll need to count the bytes that each field in a struct takes up to properly access it. (See padded vs packed structs for more information as to how to efficiently build a struct).

- **Static**—this memory is where data that's specified prior to running the program exists. It's where all the data declared under the `.data` directives at the top of RISC-V files are stored.
- **Code**—after the RISC-V section as well as instruction formatting sections, it should be a bit more clear as to how the program actually gets stored in the code/text part of memory. After the higher level programs get translated into assembly code, it gets further broken down into machine code which then gets ported to memory where it has to sit in order to actually be run later.

Functions in RISC-V

Functions in RISC-V are quite simple, almost dangerous in its simplicity. On one hand, there are a defined number of instructions which can be used, at least in the standard ISA. That severely limits the actual syntactical variety we're exposed to in the code, unlike the higher level programming languages you might be used to. However, this also introduces complications as each individual line in RISC-V is a singular operation. This seems simple enough but our brains aren't designed to think through things one at a time; instead, it's meant to consolidate which is why it's sometimes harder to break down tasks into bite-sized chunks than to build up. This allows us both to boil down code into its simplest form but also introduces the possibility of error due to erroneous calculations, whether it be too many steps taken or too few. There are several components to writing RISC-V functions, and following, RISC-V files, which we'll explore in the following sections.

Registers

There are 32 registers in the base RISC-V version we're working with. You'll notice in the RISC-V green sheet (as well as on various online resources) there are extensions, but we'll be ignoring them in CS61C for the time being as they fundamentally function the same way as the lower 32 registers. In this section, we'll explore the different characteristics of the various groups of registers and we'll be referring to the following chart in explanations. Some register groups have specific qualities for which the notes section do not suffice; they've been noted with a *. A few definitions before we begin:

- **Caller:** the function that's calling another function (this will be detailed more later); in the context of Calling Convention (CC) for the following chart, being marked "caller" indicates that it's the job of the caller to save that particular register/set of registers. We'll be discussion Calling Convention more later on.

- **Callee:** the function that's **being** called (ditto); same as caller, if it's indicated under CC in the chart, it means it's the job of the callee to save the register/set of registers to follow calling convention.
- **Volatile:** for registers, assumed /subject to change across a function call. For the state of the world: constantly. :(
- **Non-Volatile:** for registers, can be assumed to "not change" across a function call. For chemistry: noble gases!

Reg	ABI	Desc	CC	State	Notes
x0*	zero	Hard-wired zero	—	non-volatile	Used as 0, false, NULL, for throwing away results; equivalent to logically anding all bits by 0
x1*	ra	return address	caller	volatile (both)	stores return address by default if rd is not set for certain instructions; often saved by callee in examples
x2*	sp	stack pointer	callee	non-volatile	always stores pointer (memory address) of bottom of stack (i.e. what is the lowest position available); should **never** be manually outside of incrementing/decrementing
x3	gp	global pointer	—	—	disregard for CS61C
x4	tp	thread pointer	—	—	disregard for CS61C
x5	t0	first temporary register	caller	volatile	should almost always be first temporary register used
x6-x7	t1-t2	temporary registers	caller	volatile	ditto
x8	s0/fp	saved register/frame pointer	callee	non-volatile	we'll be using it as the first saved register in most processes
x9	s1	saved register	callee	non-volatile	ditto
x10-x11*	a0-a1	function args/return values	caller	volatile	used dually as the first argument registers and return values (RISC-V can only have 2 official returns; see notes for workarounds)
x12-x17	a2-a7	function args	caller	volatile	arguments! Should never be used before a0 and a1
x18-x27	s2-s1	saved registers	callee	non-volatile	
x28-x31	t3-t6	temporary registers	caller	volatile	

x0 → zero

The zero register is special in RISC-V as it falls into the category of being unchanging, as opposed to every other register who is subject to spontaneous change through and across functions. The zero register can be used as a comparison for the value 0 itself, if something is false, being NULL, amongst others. One of the key features is that x0 acts as the rubbish bin for all RISC-V operations. By setting the rd register in instructions to x0, you are effectively throwing out the value calculated/saved because it cannot change. Most times this

is counterintuitive, but for certain instructions and situations, it's extremely useful. We'll cover some of the special cases in the next section.

`x1` → `ra`
The `ra` register is called the **return address register** because it exclusively stores the return address when jumping to a function/memory address. The `ra := PC + 4` with PC being defined as the program counter. In the upcoming sections discussing the single cycle datapath and RISC-V datapath in particular, you'll learn more about what the program counter does. At this point, it'll suffice to know that the program counter is essentially the memory address of your current instruction. As we know, all code is essentially just bits in memory so in base RISC-V, because every instruction is 32-bits wide, we know that every instruction is **word-aligned**, that is, the instruction memory addresses are in multiples of 4. Thus, we add 4 (bytes) to where we currently are in order to move to the next step. As we'll see in the upcoming **Instructions** section, there are situations in which the translator knows the lack thereof a `rd` register in certain instructions, indicates an implicit `ra` register, much like the `x0` register.

`x2` → `sp`
The stack pointer register is unique in the sense that the user should never directly set the value —that is the job of the system. However, the user is free to set the values **in the memory each `sp` address points to with discretion**. This is because the stack memory should only be altered with regards to the previous location of the stack pointer.
If you're having trouble imagining how the stack pointer functions, you can view it as an array or a buffer. Every time the stack grows down by `-n` bytes, think of it as creating space in memory for a buffer of `n` bytes. We know that buffers almost always start from a lower memory address and index to higher up in memory, so for the buffer we just created, every time we access the space in memory, starting from `0` going up to `n - 1` indexing, we're going **up** the stack. It's kind of wonky to think about at first, like a Dali painting, but just imagine it as creating space going down but once that space is created, you index into it just as with any other array.
An example of this would be say for example our `sp` starts out at an address of `0xBFFF7380`. We want to allocate 16 bytes of memory on the stack in order to store information. We run the instruction `addi sp, sp, -16` and now our `sp` is at `0xBFFF7370`. Notice that our stack pointer is now lower in memory than before by exactly 16 bytes. Now we want to index into the "array" we just created to store, say, 4 4 byte integers. Our `sp` register is storing the **loweest** memory address in the stack we have access to, which is `0xBFFF7370`. To access the first element, we use `lw t0, 0(sp)` given that `t0` holds the first value that we want. This stores our first element at the lowest part in stack memory we have access to. The next position will be `4(sp)` which is 4 bytes physically higherin memory than our previous location. Notice that **we do not increment our stack pointer yet**. This reason for this is because by incrementing our stack pointer, we are effectively rendering the memory addresses lower than whatever the updated stack pointer value is useless and inaccessible. We then follow with `8(sp)` and `12(sp)` for our 3rd and 4th elements respectively. Notice here that we don't index all the way to `15(sp)`, or `n - 1`. This is because we're working in intervals of 4 as opposed to 1 as with that rule. After you're done using the buffer for

whatever calculations you need, you must make sure to restore the stack back to where it was previous to your **most recent buffer allocation. Look at the decrementing and incrementing of the stack as bookends —there must be a corresponding decrease for every increase starting from the inside out otherwise the values will end up mismatched and incorrectly interpreted (not to mention memory issues).**
One final note about accessing memory —we'll discuss this more in later sections but loading and storing values aren't limited to just the stack, that is just one of the most common default places where pointers to somewhere accessible in memory lie. You can just as easily pass in pointers (read: memory addresses) that have been allocated in other locations, whether it be stack memory or even heap memory. **Note: `malloc` is not an "inherent" feature of RISC-V; it requires `ecall` which is essentially making an ask to the system for memory, and is slow (as with any operation involving `ecall`) and should not be used frequently. For CS61C, `ecall`s should not be made without a wrapper function which starter code should provide.**

`x10-x11` → `a0-a1`
These two registers as you might've noticed dual as both argument registers and return registeres. The way in which this works is when calling a function, the registers will be prepared as arguments and loaded with the corresponding values. After a function is done computing the returns, the same registers (depending on how many values need to be returned) will be loaded with the return values. If there are more than 2 values to be returned, you have to sidestep this constraint by instead, passing in a pointer to a buffer in memory in which you can load your return values to your heart's content and then return the pointer in the return registers. You'll see this in action in project 2.

`a` Registers
Generally speaking, these are just argument (or return) registers. They're considered volatile because we first load the proper arguments for function A that'll call function B. Even if function B (the callee) doesn't call additional functions, there's the likelihood that there're return values which implies at the very least `a0` and possibly `a1` could be overwritten. We'll discuss this more in the Calling Convention section but it's bad practice and unsafe to assume there are no changes between function call and return.

`t` Registers
Similarly to argument/return registers, temporary registers are volatile. This shouldn't come as too much of a surprise as the word temporary is right in the name. So what separates these registers from saved registers (which we'll discuss next) and when should you use this over saved? A good way to differentiate between using temporary versus saved registers is considering, is a value an intermediate or a final value? An intermediate is just a temporary state between 2 destinations. When considering $1 + 2 + 4$, in RISC-V (and technically our brains) do $1 + 2 = 3$, then $3 + 4 = 7$. The 3 would be stored in a temporary register because it's not "permanent" and it's only used as a vesicle to the final answer, which is 7 and would thus be stored (theoretically) in a saved register. This isn't a hard rule though, and is just a guideline to help decide. Ultimately it's up to your best judgement.

`s` Registers
These registers are considered and are typically used to move around values that are considered more "permanent" and are depended on later in functions. You'll find that if you don't allocate these carefully, you'll actually run out of saved registers to use. It's not because there are too few due to the hardware but rather due to poor allocation in the code. Typically this happens when values that could be stored in `t` registers haven't been and thus take up room from more crucially saved values. If you find yourself truly running out of space however, you can always use the stack as a temporary buffer to store values. The way in which that would happen is detailed in the `sp` section with the key being to think of the stack as just a very long section of memory you can portion out into buffers.

Instructions

There are 50 unique instruction slots in the RISC-V green sheet that we can see and a total of and 59 total if we count the word-specific variations on certain instructions. Out of those, we only truly use a subset of them (you'll notice that certain ones we never mention). This is because they're just decently uncommon, has to do with the operating system (which we handle for you), or simply not within the scope of this course. The following instructions are the ones for which you should at least have some level of understanding of:

1. What the instruction does
2. How the instruction functions
3. Instruction type
4. Order of registers/parameters
5. Instructor formatting/translation to machine code

They include the following: `add`, `addi`, `and`, `andi`, `auipc*`, `beq`, `bge`, `bgeu`, `blt`, `bltu`, `bne`, `jal*`, `jalr*`, `lb`, `lbu`, `ld`, `lh`, `lhu`, `lui*`, `lw*`, `lwu`, `or`, `ori`, `sb`, `sd`, `sh`, `sll`, `slli*`, `slt`, `slti`, `sltiu`, `sltu`, `sra`, `srai*`, `srl`, `srli*`, `sub`, `sw*`, `xor`, `xori`
Most of these instructions are decently self-explanatory, with details being given on the green sheet, but a few of them warrant another looking at. Those will be marked with a `*`. With regards to the green sheet itself and how to read it, in particular the third column from the left on the first page, we'll discuss that more in the next section.

UJ Instructions
The two upper immediate instructions are `auipc` `lui`. Both instructions deal with formatting and transferring the upper (most significant) 20 bits of an immediate (including memory addresses) into `rd`.

- `auipc`—add upper immediate to program counter
  this can also be considered **PC-relative loading**; the instruction is adding the current PC to the immediate and loading the upper 20 bits into `rd`. You'll most commonly see

this being used in the **load address** pseudoinstruction, which can be used as follows: `la <label>`.

  - A note, it was mentioned in lecture that the majority of pseudoinstructions are formed because the combination of actual RISC-V instructions are used so commonly that it would be easier just to create a "fake" instruction that makes life easier for users. This is the case for instructions such as `mv <rd> <rs1>` and `neg <rd> <rs1>`. Whilst `la` does accomplish this by combining the `auipc <rd> <imm>` and subquently, `addi <rd> <rs1> <rs2>` instructions (upper 20 bits + lower 12 bits), it's out of necessity, not by choice.
  - Because we're working with 32 bit memory addresses (which is what gets loaded from the label and since the PC is also 32 bits, we're attempting to load 32 bits worth of immediate with an instruction that's also 32 bits. Logically speaking this does not work because it leaves us no space to specify the instruction or the destination. Hence why we must break it up into two pieces.

- `lui`—load upper immediate
  this is considered to be **0-relative loading**; the instruction is adding 0 to the immediate and loading the upper 20 bits into `rd`. This is different from `auipc` because there is no additional value being added to the immediate taken in. We also call thie **absolute addressing** as we're accessing the address exactly as it was loaded, or absolutely. This gives less flexibility than `auipc` does because similar to biased versus regular notation, even if we have access to the same number of bits, the bias (in this case PC), allows us to reach more values overall in a more flexible manner. This only really affects the accesses in a memory-range larger than the absolute reach of the two instructions though as anything "extra" the PC addition provides winds up wrapping across the top/bottom of memory to the other side (similar to overflow).

`lw`—load word
Used as `lw <rd> <offset>(<rs1>)`, the instruction accesses memory at the address specified in `rs1 + offset` and loads the word (4 bytes) stored there. This is the equivalent of dereferencing a void pointer in C—for a piece of data 4 bytes wide, you would call something along the lines of `*(ptr + 4)`. Note that we're using a `void*` pointer here because in RISC-V, we don't differentiate between types—to move places in memory, everything is in terms of bytes unless otherwise specified.

- `sw`—save word
  Used as `lw <rs1> <offset>(<rd>)`, save word works essentially the same way that load word does, except it goes the other direction, i.e. it saves the word in `rs1` (first register) into the memory pointed to by the offset specified plus the memory address stored in `rd`. You can think of loading a word as an arrow pointing from the right register to the left register and moving the value there whereas saving a word is an arrow from the left register to the right.

- NOTE: there are several other versions of loading on the RISC-V greensheet—these allow for loading/storing of different sized inputs as well as various types of bit-extension for values smaller than what our immediate field needs for loading/storing.

Jump Instructions
The family of jump instructions gets its own categories because there are so many ways in which we can look at the instructions, both the actual and pseudo.

- `jal`—jump-and-link
  - **General Format:** `jal <rd> <label>`
    The full-length instruction has 3 parts—the instruction name, the register destination, which is where the return address (`PC + 4`) would go, and the label to which the function would jump. The label can be a function label or one defined in the directives, but at the minimum it has to be a non-register argument.
  - **Abbreviated Format:** `jal <label>`
    Occasionally (more often than not), you'll see `jal` being used with just one argument, i.e. it's missing the register destination. This is like a half-pseudoinstruction—the fact that there is no `rd` doesn't mean the return address doesn't get saved, but rather there's an implicit `ra` declared there. The full written out version of this instruction would be `jal ra <label>`.
  - `j <label>`—jump
    This pseudoinstruction can be thought of as an extension of the above one. Instead of having an implicit `ra` as the register destination for the return address, it'll be `x0` instead. Because we know `x0`'s value can't be altered, this is equivalent throwing away the return address because we no longer need it. We would use this in the circumstance that we want to jump to a different function and not return to where we called it from because we were done entirely there.
- `jalr`—jump-and-link-register
  - **General Format:** `jalr <rd> <offset>(<rs1>)`
    The full-length instruction has 4 parts—the instruction name, the register destination (where the return address is stored), the offset, and the argument register. The instruction adds the offset to the memory address stored in `rs1`, the return address gets stored in `rd`, then the function jumps to `M[rs1] + offset`. **NOTE: the offset must be in a multiple of 4 for base RISC-V because since instructions are 32 bits each, AKA 4 bytes AKA 1 word, if we jump to an offset not a multiple of 4, we'd effectively be splicing different instructions together which don't get parsed into the correct instruction.**
  - **Abbreviated Format:** `jalr <rs1>`
    Like with `jal`, sometimes you'll see an abbreviated format of `jalr` used with only 2 components. The lack thereof of an offset creates an implicit assumption by the translator of `offset = 0` and as with `jal`, no `rd` implies `rd = ra`.
  - `jr <rs1>`—jump register
    This pseudoinstruction is similar to `j` in it being an extension of `jalr`, its main family of instruction. Because it only has 2 items specified, we know 2 are implicit, that being `rd = x0` and `offset = 0`. Most of the time, you'll see this being used

to jump to a return address or something similar, but you can use it to jump to any register's memory value you want to.

  * `jr ra`—`ret`
    This is a pseudoinstruction on top of the existing pseudo, like a meta-pseudo. It specifies that we're jumping to what's in our return address register without tracking our updated return address. This is the equivalent of calling `return` in C.

`slli` —shift left logical immediate
This instruction is used in the format `slli <rd> <rs1> <imm>` and is the equivalent of the `a << b` shifting operator in C (as well as other languages). It moves all the bits to the left by whatever value the immediate specifies. By doing so, the instruction is effectively multiplying the original value by 2^{imm}. This is because every additional bit position we're at that's a higher priority is two-times the bit before it. This works well if the goal is to multiply a value by a power of 2 because it's slightly quicker than the `mul` instruction. **NOTE: to shift by a value stored in a register instead of an immediate, use `slli <rd> <rs1> <rs2>` with `rs2` being the amount shifted.**

`srli` —shift right logical immediate
The format is `srli <rd> <rs1> <imm>` and it's essentially the same as `slli` except the shifting happens to the right. From number rep, we know that there are two ways we can extend a number if there are more bit positions to fill —sign extension and 0-extension (or unsigned extension). The logical shifts will always favour 0-extending values so it's particularly useful for moving unsigned numbers. Shifitng bits right is the equivalent of dividing a value by 2^{imm}, much like to the left is to multiply by that value. **NOTE: the register version of this instruction is `srl <rd> <rs1> <rs2>`.**

`srai` —shift right arithmetic immediate
The format for this one is instruction is `srai <rd> <rs1> <imm>`. This is essentially the same as `srli` except the missing bits are sign-extended instead. **NOTE: the register version of this instruction is `sra <rd> <rs1> <rs2>`.**

Verilog (Green Sheet)

CS61C doesn't go too deep into Verilog because it's not needed in order to understand this class, however, it's useful to understand at a basic level in order to read the green sheet quickly and easily. This section will just cover a few of the basic features you might see but if you're interested, definitely take EECS151.

Fundamentally, Verilog is just a hardware description language —it allows us to transcribe electronic components you see on the hardware into a readable format, much like the symbols for resistors, capacitors, transistors, etc... exist for EE. There are a few key components to pay attention to on the green sheet in particular:

- **R**—You'll see this appearing before the following 3 terms: **rd**, **rs1**, and **rs2**. It just

indicates that we're using that particular register as a register, meaning we use the value stored **in** the register. This seems redundant but we'll see later why it is not.

  - **rd**—register destination; it'll always appear as the first register in any instruction that needs a register destination, i.e. on the left-most side. This is where values get stored, whether it's from memory or calculated. **NOTE: not all instructions need an rd.**
  - **rs1**—register source 1; it's the second register to appear after **rd** should there be more than 1 register needed for the instruction. **NOTE: if there is a source register needed, rs1 will always appear.**
  - **rs2**—register source 2; it's the third register to appear after **rd** and **rs1** if there are 3 registers needed for an instruction. **NOTE: rs2 will never appear without rs1 because it'll only be used if there are 2 sources specified.**

- **M**—This is another symbol that'll appear before a register. This symbol indicates that a memory-access will be made and that the value inside of the register will be treated as a memory address instead of as a simple value.
- **PC**—program counter; indicates where in the memory your instruction sits. In the next section of CS61C, you'll see there is a distinct **instruction memory** and that is where the PC should be pointing.
- **imm**—immediate; should only appear in instructions that require an immediate input; can be thought of as a constant that you hardcode in
  - **{imm,*n*b'0}**—this is a way of representing which bits of the immediate in the instruction are represented. The n represents the least most significant bit that is represented by the immediate in the instruction. The two values of n that are most likely represented are 1 and 12.
    The former is used typically for jump instructions because we know that jumps can only happen in a word-aligned manner. By not including the 0th bit, we allot ourselves more bits to have a larger jump range, since we know the 0th bit must be 0. So the natural question begs, "why do we not start counting the bits starting from index 2 so we can have a larger range to jump to?" The reason for this is because we still need backwards compatibility with older RISC-V code and translators, back when program instructions were 2 bytes instead of 4 (this is a result of smaller memories). Thus, we need to still include the bit at index 1.

    The latter (*$n = 12$* is used almost exclusively in UJ instructions. This is because they deal with the upper 20 bits of immediates, so it has no need to consider anything below the bit at index 12.
- **Other Symbols**—these operate as usual, as their respective logical, arithmetic, and ternary operators. Any phrases you don't recognise are out of scope for this course.

C to RISC-V

This section is very much not finished and will be added to later.
In general, translation from C to RISC-V includes a few keys steps to follow.
General Steps

1. Break down line of C code into its fundamental components—this means that lines that have multiple operations, for example, `arr[i + 2] = arr[i + 1] + arr[i];` is not just one simple line in RISC-V. It would be broken down into:
    - Sum of `i + 1`
    - Sum of `i + 2`
    - Memory access `arr[i]`
    - Memory access `arr[i + 1]`
    - Sum of `arr[i] + arr[i + 1]`
    - Memory access loading into `arr[i + 2]`
2. Break down problem into subparts—which ones should have their own label? Some examples include different exit conditions (from loops, conditionals, the function itself, etc...), separate loops, different functions getting called, etc...
3. Decide which operations require temporary (read: intermediate values) registers and which ones should get saved registers.
4. Work through each line of C code and ensure that **you** know which registers are being manipulated and what they represent.
5. Run a test or two!

Writing Functions in RISC-V

This section is very much not finished and will be added to later.

Writing functions in RISC-V is very similar to translating them from C—you're essentially translating the more condensed version of the function from your mind/spec into broken-down step-by-step instruction calls. There is no real "formula" unfortunately to writing functions (much like there isn't any oracle you can consult for other languages) but there are certain things to keep in mind when doing so to make your (and the TA helping debug your code's) lives easier.
Some RISC-V Function Tips

- Comment your code! This sounds obvious but it's even more important than usual to comment your code in depth and clearly. There are two things to do:
    1. Break down your problem into components to consider and detail what each portion is meant to do.

2. For each line you write, on the side comment in what values are being calculated and what different registers hold; yes it'll take a bit longer than just writing straight code but your future you (and the TA helping you) will love you for it.

- Be deliberate in what instructions you're using—if you don't know why you're using it, don't just put it in.
- Remember calling convention! It exists for both the callee **AND** the caller (which often gets forgotten).
- Always check the argument registers needed for a function and the return register contents.
- If you can, create labels in static data for your values used frequently! It'll be more clear when you're referencing them later on.

Calling Convention

Calling convention can be defined as the general set of rules one should follow when writing proper RISC-V code; taking it one step further, as rules one should follow when writing any assembly code. Although this, true to its name, is just "convention", it's important to follow. Technically speaking, any register could be used to functionally hold any type of value —`a0` could be used as a saved register, `t0` could be used as an argument register, etc... When you write your RISC-V code, you'll notice that running it locally or on venus's web platform will often times produce the right results. That's due to the fact that RISC-V registers are merely 32-bit storage devices in the hardware and have been simulated through Venus. However, you'll also notice that running the autograder will produce all sorts of errors. This is because calling convention isn't followed.

One of the biggest reasons behind having a calling convention is due to the need for regularity and to provide an abstraction layer between various functions and files. Say in scenario 1 you merely wanted to create `functA` within `fileA.S` and `functB` within `fileB.S` with `functA` calling `functB`. Assuming no calling convention is followed, it's perfectly simple for you to know what registers need to be maintained across the function call `functB` as well as what registers are manipulated within the functions themselves. However, as soon as we introduce linked libraries and foreign files, that dependency knowledge collapses —you have no way of knowing what registers are necessarily stored and which are volatile. In the case of RISC-V files, you'll notice that there are no `#include`s, `import`s, or other various import statements. That's due to the fact that in assembly the keywords used are incredibly specific and are limited to a small subset. This falls in line with other RISC design principles —simplicity is key.

RISC accomplishes this through the `.globl` directive, as we previously discussed. Anything included under that directive is fair game to be used elsewhere in the files under the same directory. You might notice that in Project 2, the only external calls, i.e. the functions called that you didn't write, are located in `utils.s` and that it's quite simple to read through and to understand, given that every label governs approximately 3-5 lines. However, this is

not the case all the time. We can see clearly that in these function calls, we're only manipulating the arguments to syscalls (system call; out of scope for CS61C, if you're interested, check out CS162). However, as external labels govern more and more complicated functions, it's impossible (read: very inefficient) to track what registers are used and what are not. Not only that, but in order to make sure for all calls of the label functionally work, you'd have to guarantee whatever registers you rely on post-function call also get saved prior to the call. All in all, it would make for an extremely unpleasant experience.

We ameliorate this by providing a concrete set of rules to follow with regards to how registers are used, and which part in a function call takes care of the values, for every function declaration and call. We'll explore these rules in the upcoming sections.

The Bucket Analogy[2]

> ... Viewing registers as buckets; let's say we have 3 buckets, a T bucket, an S bucket, and an A bucket —jumping from label to label is like picking up bucket and moving it to a different part of the room; since it's just one person (file) working with it, the functions don't need to worry about saving the water inside [each bucket]. Jumping to a function is like me (caller) giving the bucket to my friend (callee). If my friend needs the bucket for orange juice instead of water, they can use it, but when they give me my bucket back, if my water's not still in the bucket, imma be mad so, it's up to THEM to save my water in a separate container before using the bucket and make sure when I get my bucket back, it's like nothing happened to it (remember to clean the bucket though!). T buckets (temporary registers, `t0|t6` have an issue where the moment I give it to a friend, all the water evaporates, i.e. we have to save our own water before passing it off. S buckets (saved registers, `s0|s11`) can be saved by our friend. A buckets (argument/return registers, `a|a7`, `a0, a1`) are if I want my friend to do something with the water, then return something else back to me. ...

```
.globl dan

dan:
    //use some registers, t0, t1
    //...
    addi sp, sp, -8
    sw t0, 0(sp)
    sw t1, 4(sp)
    jal bora
    lw t1, 4(sp)
    lw t0, 0(sp)
    addi sp, sp, 8
    //...
    //use registers t0, t1 again
```

[2] as created and described by Sambodh Mitra (Fa20 AI), edited by Caroline Liu (Fa20 TA

```
bora:
    addi sp, sp, -20
    sw s0, 0(sp)
    sw s1, 4(sp)
    sw s2, 8(sp)
    sw s3, 12(sp)
    sw ra, 16(sp)

    mv s0, a0
    addi s1, a1, 0

    //do some stuff

    lw ra, 16(sp)
    lw s3, 12(sp)
    lw s2, 8(sp)
    lw s1, 4(sp)
    lw s0, 0(sp)
    jr ra
```

Caller

This section is very much not finished and will be added to later. It will probably be mostly in a separate set of notes.

The caller (function) is the first half of what we deal with in calling convention —it can be considered as the "outside" function or the wrapper function. In the above example, `dan` would be the caller because it calls `bora` somewhere within the function.

Callee

This section is very much not finished and will be added to later. It will probably be mostly in a separate set of notes.

The callee (function) is the second half of what we deal with in calling convention —it is the inner function or the function that is called. In the above example, `bora` would be the caller because it is called by `dan`.

Note, a function can be both a caller and a callee —it just depends on what perspective you'll looking at it from. Say if `bora` called a function `cs61c` somewhere inside its body, it would be considered the caller of `cs61c` and the callee of `dan`.

Steps to Follow

Though it's the eventual goal that calling convention becomes second nature, it can sometimes be hard to keep in mind —it's like remembering to not return references to variables declared on the stack! You know what to do but yet somehow, you're like a moth drawn to flame —it just keeps happening. To mediate that, we have ourselves a handy dandy list of things to keep in mind when doing operations and writing functions to obey calling convention!

Callee (Prologue):

1. Allocate space on stack (`sp`)

$$-4 \times (num\ items\ saved)$$

2. Save the `s` registers and `ra` on the stack (`sw rs1, <offset>(sp)` command)
3. Either save the args on the stack (if you don't have enough saved registers and need them later OR move them into the previously "emptied" saved registers (`mv rd, rs1` OR `addi rd, rs1, 0` commands).

Callee (Epilogue):

1. Store return values (up to 2 without some fanagling) in `a0`, `a1` registers
2. Load the `s` registers and `ra` from off the stack (`lw rd, <offset>(sp)` command)
3. Restore the stack (`sp`) to where it was previously

$$+4 \times (number\ of\ items\ saved\ in\ prologue)$$

Bookends (Caller):

1. Allocate space on stack (`sp`) for how many volatile registers you need post-function call; remember: saving everything is not good coding practice! It's called hoarding.
2. Save any volatile registers you might need later on onto the stack; this might include temporary registers with counters, calculated values, addresses, etc... This should **not** include argument registers because those should've been saved in one way or another in your prologue.
    - **NOTE: `ra` is in a weird place when it comes to calling convention. On one hand, it's considered volatile and should not rely on being saved however, it is good habit to save the new return address after a function call in the prologue. This is to guarantee that the new return address is the one we're jumping to after the epilogue.**
3. Jump or branch to where you need to go! The function that you're jumping to would be the new callee (and should follow the callee's calling convention as detailed earlier).

4. Restore the volatile registers on whose values you count on post-function call and those **only** to the same registers previously used.
5. Restore the stack

It's important to note that functions are often **both caller and callee**—this is because anytime a function calls something inside the body, it becomes a caller and anytime it gets called, it becomes the callee. Thus, often times you'll see both aspects of calling convention within a single function body.

Common Questions