x86_64 Assembly Tips & Tricks

EN 601.229: CSF @ JHU
Version 0.1.0

The CSF Course Staff (Max Hahn)

September 22, 2022

Contents

1 Notice

We politely request that if you make improvements to this document, that you let the CSF course staff know so that we can incorporate your changes. The jhucsf Github organization web page has contact information.

2 Introduction

This document summaries the key takeaways of the hundreds of discussion posts and questions we have encountered during the assembly portion of the class in the years we have offered this course. It is not intended to be read cover-to-cover (though you are free to do so); instead it is meant to serve as a reference to help you out as you encounter difficulties in the course of completing the assignments. The document is indexed, and should have PDF bookmarks to each section. We hope this document will be helpful!

3 Basics

3.1 Conventions used in this text

1. `text` will denote a snippet from a program such as a register name, instruction, or other text from a programming language.
2. `[thing]` will indicate that `[thing]` is a placeholder meant to be replaced in actual usage.
3. longer form examples will be in monospaced block listings.
4. some_link will denote a link and should be clickable in any competent PDF viewer.

3.2 Terminology

Here is a brief glossary of the unfamiliar terms used in this document:

1. *ABI* - Application Binary Interface, refers to the set of contract between applications and the kernel and includes things such as the calling convention, syscall convention, stack frame layout, and how memory spaces are provisioned for each process. Differs by operating system and underlying hardware. In this class, we are targeting x86_64 Linux, so we will be following the x86_64 System-V ABI.
2. *Calling Convention* - Rules that functions must follow on how they call other functions, and how they will handle global state such as registers and the stack.
3. *Stack frame* - A collection of all the information (non-register local variables) needed for a function call's execution. Created by the combination of the `call` instruction and any *function preamble* (the part of the function before the main body where rsp is adjusted and rbp is saved).

4 Mechanics

We'll briefly cover some important topics that don't have very good coverage in the usual resources here.

4.1 Reference Resources

Here are some basic assembly references that students have found to be helpful in the past:

1. The classic BrownU reference docs. They are using AT&T syntax, so the examples are completely compatible with the assembler we want you to use in this class.
2. UVA's intro guide. (Beware the instruction argument order! This is Intel syntax which is the reverse of the AT&T syntax used in this class.)
3. Intel's own intro docs (Beware the instruction argument order! This is intel sytax, which is the reverse of the AT&T syntax used in this class.)
4. A good reference for the register breakdown: wikibooks.
5. A comprehensive reference for x86 instructions.

4.2 Memory references

Most x86 instructions can access at least one memory location (i.e. perform the equivalent of the C memory operation `*var`). This is done by replacing one of the normal instruction arguments with the memory reference syntax `offset(base_address, index, index_size)`, where offset is an integer literal, base_address is a 8-byte register, offset is an 8-byte register, and offset_size is one of 1, 2, 4, or 8. The syntax will compute $offset + base\_address + index * index\_size$, and dereferences that address to get the value in memory. The offset is meant to be used to add a constant index to a memory address and may be negative (e.g. if you want to index stack space you created by moving rsp). The index is meant to help with basic array indexing operations, which is why the index_size is configurable. Any unused components may be left out, and `offset` will default to 0, `index` will default o 0 and `index_size` will default to 1.

Here are some example of this syntax, used with the `mov` instruction:

```
movq $(%rax), %rcx /* C equivalent: uint64_t rcx = *rax */
movb $(%rax), %cl /* C equivalent: uint8_t cl = *rax */
movq -128(%rax), %rcx /* C equivalent: uint64_t rcx = *((uint64_t*)(((uint8_t
    *)rax) - 128)) */
movb (%rax, %rcx, 4), %dil /* C equivalent: uint8_t rdx = rax[rcx] */
movl (%rax, %rcx, 4), %edx /* C equivalent: uint32_t rdx = rax[rcx] */
movq (%rax, %rcx, 8), %rdx /* C equivalent: uint64_t rdx = rax[rcx] */
/* note that register, memory ordering also works */
```

Notice that since this is assembly, you do have to consider the size of the items in the array when computing offsets and indices, unlike C, where the compiler automatically adjusts this for you when you work with pointer arithmetic or array indexing.

Generally speaking, most assembly instructions will only take one memory reference. E.g.

```
/* incorrect, will not assemble */
movq (%rax), (%rcx)
```

```
/* correct, must use intermediate register */
movq (%rax), %rdx
movq %rdx, (%rcx)
```

sometimes you will want to take the address of a memory reference instead of dereferencing the pointer. E.g. this would occur any time that a c function wants a pointer, but you have a normal variable. To do this, you can use `leaq [memory reference expression], [dest register]`, e.g. `leaq (%rax, %rcx, 8), %rdx` effectively does `uint64_t* rdx = &rax[rcx]`. The `leaq` instruction actually just computes the $offset + base\_address + index * index\_size$ and stores it in the destination register using addressing hardware, so in essence it is just a special case of a series of math instruction that completes much faster (and thus `leaq (%rax), %rcx` is exactly equivalent to `movq %rax, %rcx`).

4.3 Assembler directives

In this class, we are using the GAS (GNU ASsembler) with AT&T syntax, as it is the standard toolchain for Linux. The capital `.S` file denotes that the file will be preprocessed using the C preprocessor, so you are free to create named constants using the `#define` directive. Any line in the `.S` file that begins with a `.` character is an assembler directive, and generally is not directly emitted into the executable like the instructions are. You should beware for the following directives:

1. `.section [name]` - specifies which segment in memory the following lines will be placed in. More on this in the next section.
2. `.string "[str]"` - use after a label in the .rodata section to create add string literal that can be referenced from later code.
3. `.space [size in bytes], [fill val]` - use with a label in the .data section to specify a statically allocated mutable global area of memory (e.g. for global arrays).
4. `.globl [label]` - marks the label `[label]` as a global symbol, which allows it to be exported from .o file. These should correspond to any functions you have declared in your .c file without the `static` keyword in the function signature. If you get a "undefined symbol ..." error when you try compile your assembly implementation, you probably forgot to add this directive to the symbol.

4.4 Sections

As you will learn later in the course, a modern elf executable (the application binary format on Linux) will place its code an data into segments, each with different permissions. This is an important part of the security model, as we should not be able to execute parts that contain data (such as strings and constants), while we should not be allowed to modify executable areas of the binary at runtime. In higher level languages, the compiler handles splitting everything apart and putting them in the right sections for you. In assembly, you

have to specify this yourself. a section direction is valid until another one is encountered in the file.

The primary sections of interest for this class are

1. `.text` - The section where executable code should code. You must ensure that `.section .text` is present before any assembly instructions in every assembly file you write to prevent permission error and segfaults in the final binary.
2. `.rodata` - Read-only global data. This includes string literals and constants. Anything that should not be executable or modifiable at runtime should go here.
3. `.data` - Mutable global data. This can be an easier way to allocate things like local arrays in non-recursive functions, but do note that this makes them effective global variables.

Here is an example layout for a `.S` file:

```
.section .rodata
/* everything below here will be placed in the .rodata section */
helloStr: .string "Hello!"
goodbyeStr: .string "goodbye!"

.section .data
/* everything below here will be placed in the .data section */
mutableGlobalArray: .space 100, 0 /* array of 100 bytes, e.g. char[100] */

.section .text
/* everything below here will be placed in the .text section */
.globl myFun
myFun:
    /* body of myFun */
    ret
```

4.5 Immediates, labels, and registers

Immediates are things we would generally consider to be literals in C programs, and are emitted directly into the assembly stream as part of the instruction encoding. In the AT&T syntax we use, they must be prefixed with a `$`. For instance, the C assignment `rax = 153` should be written as `movq $135, %rax`.

Registers are the special locations that instruction may directly operate on, are shared global state (the mental model is 16 global variables), and must be prefixed by `%` in the assembly syntax. E.g. to access the register `rax`, you must use `%rax` in the assembly code.

Labels allow a location (address) in the assembly to be given a name, and look like `[labelname]:` immediately before the assembly instruction, or directive it should be attached to. Labels that begin with `.L` will be treated as local labels by the assembler and

will not appear as a function in GDB. However, all label names in a file must be unique; there is no such thing as a true scope-local label.

Using the immediate prefix with a label asks for the absolute address of the label (i.e. gets the pointer defined by the label). Using the label name without the immediate prefix dereferences the label address to get the value in memory at the label. Labels may not be indexed using the memory reference syntax above. If a label is attached to an array, the label's address must first be loaded into a register before it can be indexed. A register may not be converted into an address; if a function requires a pointer(e.g. `scanf` and friends), then either stack memory or labeled space in the `.data` section must be used. This is illustrated below:

```
.section .rodata:
    /* read-only array of ten 64 bit integers */
    globalArray: .quad 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
    readString: .string "%ld"
    writeString: .string "number: %ld\n"

.section .text
.globl fun
fun:
    pushq %rbp
    movq %rsp, %rbp
    /* Stack frame:
     * -8(%rbp) - 64-bit alignment padding
     * -16(%rbp) - var int64 input_long
     */
    subq $16, %rsp
    pushq %r12
    pushq %r13
    /* read a number from stdin into input_long */
    xorl %eax, %eax /* efficient way to zero an entire register (see section
        5.3.1) */
    movq $readString, %rdi
    leaq -16(%rbp), %rsi
    callq scanf
    /* now loop over our array,a nd add the read in number to each cell */
    /* we need to use callee-save registers here since the printf call
       will kill caller-save registers */
    movq $globalArray, %r12 /* load the array's base address into a callee-save
        register */
    xorq %r13, %r13 /* zero loop counter (also must be callee-save since
                     printf is called in loop*/
    jmp .LloopCond /* jump should not have the immediate prefix on the label */
.LloopTop:
    xorl %eax, %eax
    movq $writeString, %rdi /* load printf fmtstring into first arg */
    movq (%r12, %r13, 8), %rsi /* load globalArray[i] into second function arg */
    /* note for performance reasons, we should load -24(%rbp) into a register
       if the loop happens mroe than a few times */
    addq -16(%rbp), %rsi /* globalarray [i] + input_long */
```

```
    callq printf
    incq %r13 /* increment loop counter */
.LloopCond:
    cmp $10, %r13
    jb .LloopTop /* jump while loop counter is < 10 */

    /* demonstrate that using a label without the $ is a deref */
    /* print first value in array */
    xorl eax, eax
    movq $writeString, %rdi /* load format string into first arg */
    movq globalArray, %rsi /* load *globalArray into second arg */
    callq printf

    /* clean stack frame */
    popq %r13
    popq %r12
    addq $16, %rbp
    popq %rbp
    ret
```

4.6 Signed/unsigned numbers

Since the native signed number representation on x86 is 2's complement, the additional and subtraction opcodes work the same for both unsigned and signed numbers (they set both sets of internal flags). The distinction is made in the multiply and divide instructions (`imul` signed multiply, `mul` unsigned multiply, `idiv` signed divide, `div` unsigned divide), and in the conditional jump instructions (`jae`, `ja`, `jbe`, `jb` (a is a mnemonic for above, e is the mnemonic for equals, b is the mnemonic of below, corresponding to unsigned greater than and less than), `jgt`, `jg`, `jle`, `jl` for the unsigned versions).

Reference this link for a full list of conditional jump instructions.

4.7 Odd instructions

Some notes on various instructions in no particular order:

1. Explanation for the zeroing idiom used in this doc: When zeroing a register, `xor r, r` is the most efficient way to do this on modern CPUs, instead of a mov instruction. (For the first eight registers, since touching the 32-bit subregister zeroes the upper 32-bit, it's slightly preferable to zero the 32-bit subregister even when using the full register, as the instruction encoding is slightly shorter)

2. The shift instruction takes either an immediate or register for the shift amount (`shr`, `shl`). If using the register form, it can only take the `%cl` register as the shift amount. Attempting to use any other register will give an "unknown opcode" error (e.g. `shrq %cl, %rax` is a valid shift of rax right by the value of cl, `shrq %rdx, %rax` is not). For a 64-bit instruction, the value of `%cl` is masked to 6-bits (only the lowest 6 of the

8 bits will be used to determine the shift), which is the underlying reason why shifting by an amount ¿= to the width of an integer is undefined behaviour in C. (the mask is 5-bits for any narrow forms of the shift instruction)

3. The two operand form of `imul` is restricted to registers of with 16 or above. If operating on 8-bit registers, only the single operand form is defined.

4. `mul` and `div` only exist in single operand forms. Refer to the reference docs for more details on the implicit registers used by those instructions.

5. When using `div` on anything wider than 8-bits, remember that the dividend must be zero-extended to its corresponding register pair (div uses double width dividends).

6. When using `idiv`, the dividend must be sign extended to its register pair. This is done with `cbtw`, `cwtl`, `cltq`, and `cqto` for sign-extending a 1, 2, 4, 8 byte register for signed division. These always operate on various subregisters of the rax register and the widest three will sign extend to subdivisions of rdx as needed.

5 Best Practices

Unless you know exactly what you are doing, and the full implications of violating a guideline, follow these strictly!

5.1 Follow the calling conventions

In this class, we will be adhering to the standard x86_64 Linux calling conventions. Do not violate any of the "rules" in this guideline, even for tiny helper functions that you can rigorously analyze.

5.1.1 Alignment

Alignment is to 16-bytes, and must be correct at *call sites*, not the beginning of the function. Alignment ensures that each function starts with a known good stack offset, and thus must be verified before each call. Since the stack starts every call offset by 8 (due to the return address of the function being pushed to the stack), this means that ensuring that the `rsp` offset at a call site is an odd multiple of 8, or $rsp_offset\%16 = 8$.

```
some_function:
    pushq %rbp /* save the base pointer (callee-save reg) */
    movq %rsp, %rbp /* build the stack frame in a ABI compatible way */
    /* Allocate a large block of memory on the stack for some stuff */
    subq $128, rsp
    /* -128(rsp) - char [64] */
    /* -64(%rsp) - int [8] */
    /* note: at this point, the stack is not aligned as 128 % 16 = 0 */
```

```
    /* bunch of assembly code */
    pushq %rax /* save rax for some reason */
    /* bunch of assembly code w/o stack adjustments */

    /* at this point the stack offset is 136, which is ALIGNED, so we can safely
        make the
    call without messing with alignment */
    call foo
    /* do other stuff including popping rax and restoring rbp */
    /* ... */
```

Aligning at the start of the function is acceptable if no intermediate stack operations are done between the alignment and call (i.e. if you don't want to think about alignment at every call site):

```
some_function:
    subq $8, %rsp /* now aligned */
    /* bunch of assembly */
    /* you must not adjust the stack pointer in any way here, i.e. you must not
        use push,
    pop, [opcode] ..., %rsp */
    call otherfun
```

But if you end up using a lot of registers at once, it is not always possible to only do stack operations at the start of the function e.g. you may need to "spill" (push) some registers to the stack so you can temporarily use them for something else.

However, this is always invalid:

```
some_function:
    pushq %rbp /* save the base pointer */
    movq %rsp, %rbp /* build the stack frame in a ABI compatible way */
    /* Allocate a large block of memory on the stack for some stuff */
    subq $120, rsp
    /* -128(rsp) - char [64] */
    /* -64(%rsp) - int [8] */
    /* note: at this point, the stack is aligned as 120 % 16 = 8 */

    /* bunch of assembly code */
    pushq %rax /* save rax for some reason */
    /* bunch of assembly code w/o stack adjustments */

    /* at this point the stack offset is 128, which is not aligned, so we've
        violated
    calling convention here. */
    call foo
```

Alignment is not something that is blindly solved by adding `subq $8, %rsp` to the beginning of your functions; you must think about the actual stack offset at every call site!

5.1.2 Callee vs. Caller saved registers

Refer to this table for a summary of which registers are caller-saved and which registers are callee-saved.

Register	Usage	Preserved across function calls
`%rax`	temporary register; for varargs functions passes the number of vector registers used; return value register	No
`%rbx`	callee-saved; optional base pointer	Yes
`%rcx`	4th argument register in func call	No
`%rdx`	3rd argument register in func call	No
`%rsp`	stack pointer	Yes
`%rbp`	frame pointer	Yes
`%rsi`	2nd argument register in func call	No
`%rdi`	1st argument register in func call	No
`%r8`	5th argument register in func call	No
`%r9`	6th argument register in func call	No
`%r10`	temporary register	No
`%r11`	temporary register	No
`%r12-%r15`	callee-save registers	Yes

(note that usage column is the conventional usage; all 16 registers above in x86 (with a few exceptions) are general purpose registers and ca be used to store any value at any time (barring `%rsp`))

In the table above, "yes" in the "Preserved across function calls" column implies a callee-save register and "no" implies a caller-save register.

Recall that registers are shared global state; unlike higher-level languages there is no concept of a "local variable", and every part of an assembly program shares the same registers as every other part. To tame the mess, the Linux ABI has specified that concept of callee-saved registers, where the called function saves the registers before modifying them, and caller-saved registers, where the calling function saves the registers before making a call if the values are still needed. Note the following points:

1. The "callee" and "caller" are from the perspective of the currently executing function. The caller is the function making the call (i.e. the current function), and the callee is the function it is calling.
2. This is only a convention, in order to have these semantics stick *every single function* must adhere to these conventions.
3. If a function makes no calls, there is no reason to go through the extra complication of using callee-saved registers unless you ran out of caller-save registers.

Thus, from the perspective of the current function, caller-saved registers may be used without bothering to save their contents, while callee-saved registers *must be saved* before use,

and restored before the function returns. Likewise, after a call, the function that made the call *must* assume that **all** caller-saved registers have had their contents annihilated.

Note: If you do not use a register, you don't have to do anything to it. Don't save it, or copy it, or do anything to it "just to be safe". However, as soon as you start using a register (i.e. write to its value), you must now start considering the register conventions for the register.

Note the implications of violating register conventions: your assembly is being driven by other code (even if you wrote main in assembly, glibc is still calling your code). Thus, violating any of these conventions can lead to *very undefined behaviour* as soon as control passes to a function out of your control, and you program may mysteriously segfault, crash or silently error (e.g. crash on return from main, even though everything else seems fine). **Do not violate register conventions!**

Important It does not matter if you wrote the function and know it won't touch certain registers, for your own sanity, and the sanity of anyone who will ever read your code, you *must pretend* like all caller-saved registers were overwritten with garbage after the control flow is passed back to the calling function. 99/100 times we have seen students attempt cute shortcuts like this, it has come back to bite them in *major ways*. If you want values to live across function calls, put them in callee-saved registers (after you've properly dealt with their current values). If you're out of callee-saved registered then you must start installing them on the stack, and dealing with the alignment issues that come with that, or allocating true local variables. **Whatever you do, don't take the shortcut!**.

5.1.3 Beware of clobbering the stack

Notice that `rsp` is a *callee-saved* register. `rsp` (and all its subregisters) is special because it's the stack pointer, and is implicitly used by `push`, `pop`, `ret`, `call`, among other instruction. In particular, `call` will always push an 8-byte return pointer to the stack before passing control to the function that was called, and `ret` will figure out where to return execution to by popping the topmost 8-byte value off the top of the stack and jumping to that address.

Notice the terrifying implication: if you fail to restore the stack correctly before hitting the `ret` keyword, you will jump to some random garbage address instead of returning control to the calling function. This is a very very hard to debug, as even GDB will be confused once this happens (you can generally tell this has happened in your program immediately segfaults, prints "Bus error", or seems to jump into the middle of nowhere aster you pass the ret opcode).

5.1.4 RSP can be moved down further to create free real estate

```
some_function:
    pushq %rbp
    movq %rsp, %rbp
    subq $108, %rsp /* now we've made 100 bytes of free real-estate (tm) to store
        stuff
```

```
like extra variables, arrays, and strings */
/* ... */
/* don't forget to restore it at the end of your function!
addq $108, %rsp
popq rbp
ret
```

You do have to keep track of how you split up this space though; we recommend leaving a comment with the offset to each logical division of this space.

5.2 The arguments for AT&T syntax are reversed

Most assembly resources found for x86_64 asm will give you intel syntax, which does not have the AT&T opcode argument annotations (the `$`, `%` prefixes before the opcode args), does not use with suffixes on the opcode (e.g. the l in "subl), and which puts the destination before the source e.g. *opcode*[*dest*], [*src*].

However, in this class, we use AT&T syntax which places the destination after the source (e.g. `opcode [src], [dest]`), which does use width opcode suffixes, and does use opcode argument annotation. Thus if you consult a standard x86_64 asm reference, remember to reverse the argument order before using the instruction in your code!

5.2.1 Subtraction and compare have reversed arguments

When you write `subq %rax, %rbx`, this actually does $rbx \leftarrow rbx - rax$. More confusingly,

```
cmp %rax, %rbx
jl .someLabel
```

will actually jump to `.someLabel` if $rbx < rax$. This idiosyncrasy exists because of the src to dest opcode argument order in AT&T assembly.

5.3 Beware the width suffixes when moving values to/from memory references

Remember that all pointers on x86_64 are 8-bytes. Thus any registers passed to the memory reference syntax `offset(base, index, multiple)` must be full-size registers, i.e. the ones that begin with R. To distinguish how much data to pull from the memory reference, the *opcode suffix* is used. For instance

```
movq (%rax), %rcx
```

copies eight bytes (a long) from the memory location at `%rax`, but

```
movb (%rax), %cl
```

copies a single byte to the smallest subregister of the `%rcx` register. This is the only instance where you can have registers of different widths in your assembly opcodes, as the memory reference register must always be the full-size register.

Likewise when loading memory references into a register for computation, always ensure that they are loaded into full 8-byte registers, and that all operations are done using 80byte instructions when manipulating the address.

5.3.1 Use the "width changing mov" to widen registers

Using a smaller subdivision of a larger register generally leaves the data in the upper portions intact (with one very important exception). For instance operating on the `%al` register will leave the upper 7-bytes of the `%rax` register at its previous value. This is a frequent source of bugs for the unsuspecting student.

To safely widen a register (zero extend), you can either zero the full register first and then copy the smaller value in, e.g.

```
xorl %eax, %eax
movb %cl, %al
```

or you can use the `movz[a][b]`, where [a] is a width suffix (one of `b`, `w`, `l`, `q`, for 1,2,4,8 bytes respectively), and [b] is a wider width suffix. For example `movzbq` will copy an 1-byte register to a 8-byte register, e.g. `movzbq %al, %rcx`. The registers used must match in width to each part.

To copy with sign extension (e.g. widening a signed number), use `movs[a][b]`, with [1] and [b] being the same as above.

Note that `movzlq`) do not exist). This is because **doing a `mov` into a 4-byte register always clears the upper 4 bytes**. However, `movslq` exists to sign-extend from 4-bytes to 8-bytes.

5.4 Assembly code executes from top to bottom in the absence of control flow

Often, students assume that assembly works like C, and that local labels work like magical "teleportation" markers that change control flow by simply existing. This is not the case!

```
some_function:
    /* bunch of assembly code */
    /* now we try write an if else */
    cmp $some_number, $rax
    jlt .LifTrue /* if rax < some_number, do something */
.LifTrue:
    /* some true stuff */
.LifFalse:
    /* some false stuff */
    jmp .LendIf
.LendIf:
    /* rest of function */
```

Can you spot the bug(s) and redundancy in this code?

Answer:

1. Without a control flow instruction after `.LifTrue` instructions, execution will continue right through the .LifFalse label, effectively running the else brach immediately after the if branch, instead of creating an if/else construct.

2. Likewise, consider what happens when $rax \geq some\_number$. There's no jump after the conditional jump, so it will also continue at the `.LifTrue` label without skipping to the `.LifFalse` label like we wanted it to.

3. The unconditional jump after the `.LifFalse` block is redundant, as execution flow would naturally continue at the `.LendIf` block in absence of the jump.

A naive rewrite to be functionally correct:

```
some_function:
    /* bunch of assembly code */
    /* now we try write an if else */
    cmp $some_number, $rax
    jlt .LifTrue /* if rax < some_number, do something */
    jmp .LifFalse
.LifTrue:
    /* some true stuff */
    jmp .LendIf
.LifFalse:
    /* some false stuff */
    jmp .LendIf
.LendIf:
    /* rest of function */
```

However, if we analyze the structure of the code, further redundancies become apparent:

```
some_function:
    /* bunch of assembly code */
    /* now we try write an if else */
    cmp $some_number, $rax
    jge .LifFalse /* Invert the conditional to get rid of one jump */
.LifTrue:
    /* some true stuff */
    jmp .LendIf
.LifFalse:
    /* some false stuff */
    /* remove the redundant jump here */
.LendIf:
    /* rest of function */
```

And that's probably the best this `if` construction will get.

5.5 Write useful comments

Write semantic comments. Write comments as you write each line of assembly. Yes, we've all done the thing where we write all the comments after the assignment is complete to get the style points. *Don't do that for assembly.* Instead, each time you write a line of assembly, you should tell yourself that you are not done until you've written the inline comment for the line. Trust us on this one.

Here is an example of bad comments (syntactic comments):

```
some_function:
    xorl %ecx, %ecx /* 0 rcx */
    xorl %edx, %edx /* 0 rdx */
    jmpq .L1 /* jump to .L1 */
.L2:
    addq $5 %rdx /* add 5 to rdx */
    incq %rcx   /* increment rcx */
.L1:
    cmpq %rcx, %rdi
    jb .L2 /* jump if rax is unsigned less than rdi */
```

You probably still have no idea what the snippet above is actually trying to do even with the comments right?

The problem with the comments in the snippet above is that they are *patently obvious*. They describe the syntax of the assembly language, not the actual intention of the logic. They are *useless*, as we could get that same information by reading the source.

Here is an example of assembly comments that are actually useful:

```
/*
 * function args:
 * - rdi -> iterations_to_sum
 */
some_function:
    xorl %ecx, %ecx /* zero the loop counter (unsigned i) */
    xorl %edx, %edx /* sum starts at zero */
    jmpq .LwhenCond /* check sum loop condition */
.whileTop:
    addq $5 %rdx /* sum += 5 */
    incq %rcx   /* increment i by 1 */
.LwhileCond:
    cmpq %rcx, %rdi
    jb .L2 /* loop while loop counter i is less than iteration_to_sum */
```

By commenting the semantic meaning of each line, we are instantly able to figure out the purpose of this function, which will help us debug any issues that crop up. (and yes, this is also applicable to comments written for higher-level languages!)

Don't be lazy with your comments, you will always pay later!

5.6 Debug using GDB

Yes, I understand that the vast majority have removed GDB from your memory as a traumatic memory from intermediate, and are now deathly allergic to its very mention. We are here to try dispel that notion. Firstly, putting something through GDB is much faster than guessing where to stick print statements and remembering to recompile 99% of the time. Secondly, trying to add a print statement in assembly is seriously non-trivial. Thirdly, we've seen the vast majority of you try debugging using the poke-and-check method, praying that tiny changes will magically work, and we are here to tell you that programming by coincidence is unacceptable. Finally, there are few things more instructive than stepping through your problematic logic exactly how the computer runs it. *Use the debugger!*

Firstly, use `layout reg` (type that into the GDB terminal and hit enter) when working on assembly. You can use `fs next` to set the focus to the next "window" so you can scroll it, and `refresh` to force all the windows to be redrawn if the printout breaks in any ways. (`layout src` is the same, without the register view, and is quite nice for debugging normal C and C++ source).

Secondly, here's a refresher of all the essential features of gdb you probably forgot:

1. `b [filename]:[linenum]` to set a breakpoint at the given line in the given file. you can omit the filename if you want a breakpoint in the current file, and you can use other identifiers that GDB recognizes, e.g. `b [funtion_name]`. Remember that you can set multiple breakpoint, so don't just set a break point on the first line of main and start stepping, that takes YEARS.

2. `c` to continue. This will run your code until it hits the next breakpoint. Remember that once the code has broken on a breakpoint, you can proceed to set some more breakpoints, and remove existing ones. Use this to skip your thousand iteration loops instead of trying to step through each iteration.

3. `i b` to list all breakpoints.

4. `dis [breakpoint #]` to temporary disable a breakpoint. Use `ena [breakpoint #]` to re-enable it.

5. `del [breakpoint #]` permanently removes a breakpoint.

6. `n` steps *over* a line, does not enter function calls, `s` steps *into* a line, does enter function calls, `fin` runs until the end of the function, useful if you want to get out of a function immediately.

7. `r [opt args]` starts the program being debugged, `[optargs]` denotes a optional list of arguments normally passed to the program on the command line. If your program takes input from stdin you pipe in from a file, you can use the file redirection form `r [opt args] < file` to dump the file into the program's `stdin`, without dump the input into gdb. Note that if you blow past the area of interest in GDB, just re-run your `r` command; it will restart the debug session with all your breakpoints intact. Don't exit GDB!

8. `p [expression]`, prints the value of any C expression. Yes this can include function calls, array access, almost anything you can do in C can be printed. If the print output is not in the right format, cast the expression to the correct type. Use `p/x [expression]` to print a numeric expression in hexadecimal format, `p [expression] @ [number]` to print a given number of elements from an array.

9. `disp [expression]` prints the value of `expression` every time GDB stops.

10. `watch [expression]` will automatically break execution when the expression changes. E.g. `watch varname` will break into interactive debugging mode every time `varname` is modified in any way by any code currently in scope. This is very powerful and can literally save hours of debugging.

11. `bt` prints the backtrace of the current location, which is the list of every nested function call that got you to the current line. `f [# from the bt command]` moves the debugging context to the given function call, allowing you to print variable values from functions above the current point int he call tree. Very handy to see how bad values got into segfaults for instance.

12. `b [break expression] if [expression]` sets a conditional breakpoint that is only activated when the expression evaluates to be true. `expression` can be pretty much any C expression. Want to break on line 105 when the loop counter i is true and the variable `char_val` is 53? Well just type `b 105 if i == true && char_val == 53`. Note that these are slower than normal breakpoints as they can't use the special CPU breakpoint hardware, but it's quite often worth the wait.

Note that GDB seems to have pretty good hot reload support. If you make changes to the source files, just run make in another terminal, and gdb should use the new executable when you use the `r` command again, allowing you to keep all of your current breakpoints.

To print register values, you must replace the `%` prefix with `$`, e.g. `%rax` can be printed using `p $rax`.

The `@` operator may not always work correctly for printing arrays. If it is not behaving, try the `x` command, documented here.

5.7 Optional: Use ABI compliant stack frames for enhanced debugging

This is not a hard-recommendation, but it is recommended if you want to have sensible backtraces in GDB and valgrind instead of a stack of question marks.

A ABI compliant stack frame looks like the following:

Position	Contents	Frame
`8n+16(%rbp)`	8-byte stack passed argument n	Previous
	...	
`16(%rbp)`	8-byte stack passed argument 0	
`8(%rbp)`	return address	Current
`0(%rbp)`	previous `%rbp` value	
`-8(%rbp)`	unspecified	
	...	
`0(%rsp)`	(local var space)	
`-128(%rsp)`	end of red zone (safe for vars)	

This allows GDB (and valgrind) to trace the backtrace by following rbp to to correct part of the stack, retrieving the return address, and using the known old rbp value to find the next stack frame recursively like a linked list. To build a compliant stack frame, use the following function prelude:

```
some_function:
    /* call pushes the return address, so that takes care of the topmost
    element of the ABI stack frame */
    pushq %rbp
    movq %rsp, %rbp
    /* now adjust the stack to create local var space */
    subq $100, %rsp /* 100 bytes allocated for example, can be any number */
    /* ... but then you must index using negative indexes from rbp */
    /* e.g. -100(%rbp) is now the start of the stack allocated region) */
    /* push caller saved registers here so you don't have to mess with them
       in the offset calculations for local variables */
    push %r13
    /* ... */
    /* body of function goes here */
```

If you build the stack frame in this manner, remember that the stack grows downwards (i.e. towards 0), but arrays are indexed upwards (i.e. towards infinity). Thus when calculating offsets, you must calculate the offset to the start of the array (i.e. the smaller end) not the larger end. Reversing this will lead to some very mysterious bugs.

The following technique for creating stack frames is not ABI compliant. It also works, but will lead to meaningless backtraces:

```
some_function:
    /* call pushes the return address, so that takes care of the topmost
    element of the ABI stack frame */
    pushq %rbp
    /* push caller saved registers here */
    pushq %r13
    /* ... */
    /* now adjust the stack to create local var space */
    subq $100, %rsp
    movq %rsp, %rbp
    /* now you can calculate positive indexes, e.g. 0(%rbp) is the start of the
       allocated space, but backtraces will not function properly. */
```

Meaningful backtraces are very nice, as they allow you to figure out how you got to a given point in your code. However, if it is too hard for you to visualize the stack using negative offsets, then the above way is acceptable, just remember that you should disregard your backtraces as the value of `%rbp` will confuse GDB and valgrind.

6 Pitfalls

6.1 Don't map C variables to registers 1:1

...because if you do that, you'll waste a bunch of registers and run of of them really quickly.

For example: how many registers does the following expression take to implement (without cute things like using the stack) $(1 + 2 - 3) * (4 + 5 + 7)$?

Answer: 2. See below:

```
    movq $7, %rax
    addq $5, %rax
    addq $4, %rax
    movq $1, %rcx
    addq $2, %rcx
    subq $3, %rcx
    imulq %rax, $rcx /* result in rcx */
```

likewise, registers should can be reused as soon as the value their modeling becomes dead (i.e. is not used anymore in a function). Considering the following chunk of C code:

```
int summate(int *arr, size_t len) {
    long i = len ;
    long acc = 0;
    while (i >= 0) {
        acc += arr[i];
        --i
    }
    /* after this point i is not used anymore, so we can stick a different
        variable in its
       register now */
    long some_math = (len + 3) * 2
    do_some_other_stuff_with_arr(arr, len, some_math)
    return acc;
}
```

and a possible assembly implementation of the above:

```
/*
 * rdi - int* arr
 * rsi - size_t len
 */
.globl summate
summate:
```

```
    pushq %rbx    /* save the accumulate before use (need to persist across call)
        */
    movq %rsi, %rcx /* initialize loop counter */
    xorl %ebx, %ebx /* zero accumulator */
    jmp .LwhileCond
.LwhileTop:
    addq (%rdi, %rcx, 4), %rbx /* acc += arr[i] */
    decq %rcx               /* decrement rcx (also sets all flag but CF */
.LwhileCond:
    cmp $0, %rcx
    jge .LwhileTop /* loop while i >= 0 */

    /* now calculate len + 3 / 2 without overwriting rsi */
    movq %rsi, %rcx /* reuse rcx now that we don't need i anymore */
    addq $3, %rcx
    imulq $2, %rcx

    /* function args passign reg have not been overwritten, so arr->rdi, len->rsi,
         and we
       put some_math->rcx by design and the initial push aligned the stack so we
           are now
       ready to call the function */
    callq do_some_other_stuff_with_arr

    /* at this point, we must assume that ALL caller-saved registers are clobbered
         */
    /* ...but rbx is callee saved, so we can assume it's still intact */
    movq %rbx, %rax /* return the accumulator */
    popq %rbx     /* restore callee saved register */
    ret
```