1. Overview: Computing-System Design & Security

This chapter effectively puts the entire book's material into a single, one-chapter-long narrative — a framework to help you better understand the discussions that come later. Do not panic if you run into material you do not understand yet, as all of this will be covered in more detail in later chapters.

1.1. Perspective: Why Teach the Two Topics Together

Jim Stevens, a Ph.D. student at the University of Maryland in the 2010s, was writing his dissertation on computer hardware and spent several years interning at the Johns Hopkins Applied Physics Lab, doing R&D in the field of security. At the time, he made the following wry observation:

> Hardware people don't know [much] about security,
> and security people don't know [much] about hardware.
>
> — Jim Stevens, U. Maryland Ph.D. student

That observation is a large part of why we wrote this book. Computing-system design goes hand-in-hand with computing-system securi-

ty. One might think that it shouldn't need to be stated explicitly, but it evidently *does* need to be stated explicitly, because computer hardware and computer security have indeed been traditionally considered separate and orthogonal fields, and the result has been a bit of a disaster. Specifically, in the name of simplicity and convenience, past computer-system designers have intentionally removed and/or undermined hardware and software primitives that would have otherwise provided *absolute security* in today's systems. One has to believe that the only reason this was done was ignorance of the ramifications.

Among other intentional design choices, modern operating systems attach superuser privileges to *software* instead of to *users and their actions*, and modern microprocessors allow user-level applications to generate explicit physical memory addresses. The first design choice opened the door to all of our modern privilege-escalation exploits, and the second design choice opened the door to *Spectre* and *Meltdown*. All of these exploits would be rendered powerless if our system designers had simply followed the traditional computing models. We will explore this fact in later chapters.

Once one knows both hardware and security, it becomes obvious that today's systems are extremely vulnerable in ways that they should *not* be, and nearly all of today's security exploits were enabled by explicit undermining of existing protections. Knowledge of both system design and security should help tomorrow's system designers from making the same mistakes as past system designers. Hopefully, by teaching/learning these two topics together, we can regain secure computing systems once again.

1.2. Background/Review: Stuff You *[should]* Already Know

Presumably, you know (a) how to program, and (b) that underlying all of computer hardware there are these things called *circuits*. The big hole in between those two concepts is called *computer organization*, or what it means to produce executable code and then execute it. This course will cover computer organization, but it is important that you understand what the fundamentals are regarding programming and hardware. This section will review the notion of computer programs and what they do — feel free to skip it if you already know this material.

1.2.1. What Is Programming?

A computer is an 'intelligent' number-cruncher. The *number-cruncher* part means that it is essentially a calculator: it has a *processor* that crunches numbers (performs operations such as addition, subtraction, logical-OR, logical-AND, etc.), and it has a *memory* that stores both the numbers to be crunched and the instructions that specify how to crunch those numbers.

The *'intelligent'* part means that a computer has one thing the traditional calculator lacks: a decision-making capability. This comes in many forms, but the primary one is called *if-then-else*. In an *if-then-else* operation, you test to see whether or not some desired condition is true, and if you find that the condition *is* true, you do one thing; if the condition is *not* true, you do another thing.

If-then-else is often expressed in programs as something like the following:

```
if raining_outside == TRUE
then:
    dress warmly
    wear a raincoat
    bring an umbrella
else:
    dress for sunny weather
    wear flip-flops
    bring sunglasses
```

It should be clear that this does not represent true *intelligence*, but it should also be clear that the data-driven decision-making capability is certainly quite valuable. Essentially, it is what enables computers to do all of the things that mere calculators cannot. Thus, *programming*.

The notion of *if-then-else* gives rise to 'loop' structures, which can be used to perform iterative tasks. An example: you perhaps learned matrix multiplication in high school or college? Recall that in the fundamental step performed over and over, one walks down two linear arrays of numbers — a row and a column — multiplying each matching pair and adding their product to a running sum. This operation is called a *dot product.*

In English, the process of having a computer perform a dot product can be written something like the following:

```
initialize a variable to act as an accumulator
do until you've hit the end of the two arrays:
    read the first element of each array
    multiply the two values together
    add their product to the accumulator
    read the second element of each array
    multiply the two values together
    add their product to the accumulator
    read the third element of each array
    multiply the two values together
    add their product to the accumulator
    etc …
```

This describes an operation that goes down the length of two linear arrays, multiplying the *first* terms of each array and adding their product to the running sum, then multiplying the *second* terms of each array and adding their product to the running sum, and so on. The operation ends when you reach the ends of the two arrays.

This is the intuition behind *iterative* behaviors (often just called *iteration*): they do the same thing over and over, just perhaps on different data items. The main thing is that they perform a small, well-defined operation over and over again until some well-defined end-condition is reached, at which point the process stops.

Because the dot product is achieved through an iterative process, it is easily expressed in computers as a *loop* operation. A computer loop is a formalism that expresses the repetitive behavior shown above in a language that a computer can easily turn into executable behavior. The loop has an *if-then-else* construct in it and a built-in notion of going back to the top and starting again. The loop checks a condition, decides whether or not to perform a given operation (here, the operation is called the *loop body*), and then the code goes back to the top and performs the condition-check again … this repeats until the condition is found to be false, at which point the loop is *exited*.

The following expresses the operation above in a more formalized loop structure, based on the *if-then-else* structure described earlier. It also adds in the concept of labels — the 'top' and 'bottom' of the loop.

```
        sum = 0
top:    if still_more_to_do == TRUE
        then:
            read the next element of each array
            multiply the two values together, save as product
            sum = sum + product
            set still_more_to_do to FALSE if we've reached the end of the arrays
            jump to top
        else:
            jump to bottom
bottom:
```

There is a test at the top, called a *loop condition* or *loop control*, and a body of code that is executed if the loop condition is true. If the condition indicates that the loop is done, then the thread of control *exits* the loop — meaning that the program moves on to executing whatever code comes *after* the loop.

Note that we have been moving from expressing our algorithms in high-level human-readable English to forms that increasingly resemble what a computer does. This is what designers do when designing their systems, both hardware and software — i.e., moving from initial abstract designs to increasingly detailed and specific designs — and it is also what is done by software programs to translate your high-level human-readable code to low-level machine-executable code.

The dot product can be expressed as a C-like pseudocode* loop as follows, a form that even more clearly shows the loop structure of the code (the loop body is the set of instructions enclosed by the curly braces, and the loop condition is the part outside of the loop body that, in this case, reads *while i < length of arrays*):

```
sum = 0
i = 0
while i < length of arrays {
    product = A[i] * B[i]
    sum = sum + product
    increment i
}
```

The difference between this code and the English (and pseudo-English) expressions before it is that this one is more readily translated into a form that the computer can understand. When one writes code in forms like pseudocode, it is meant for human designers and programmers to read, debug, use, and modify. The computer does not act on this, because one must speak to the computer in its own language, and the computer's language — its *instruction set* or *instruction set architecture* — is far more basic than anything shown above. Software programs take as input high-level expressions such as those above, and then they translate these high-level expressions into low-level ones that the computer *can* understand.

---

* The point of using *pseudocode* is to enable a designer to build with initially ambiguous steps, processes, or components at the outset, which allows the designer to focus more on the structure of his solution than its details. The initially ambiguous aspects are made more concrete over time, as the pseudocode is refined by the designer and is transformed little by little into 'real' code.

For instance, a C-language version of the dot-product pseudocode above — something that would actually compile and run — might look like the following:

```
// int *a contains pointer to one array (at the start, a points to a[0])
// int *b contains pointer to another array (at the start, b points to b[0])
// int len contains the length of both arrays

int sum;
for (sum = 0; len != 0; len--) {
    sum += (*a) * (*b);           // sum += (a[i] * b[i])
    a++;                          // increment a from a[i] to a[i+1]
    b++;                          // increment b from b[i] to b[i+1]
}
```

You may notice that this is very low-level C — it is using pointers instead of the array indices used just previously, which might have been a more natural way of expressing this algorithm. The reason for using this low-level expression instead of array syntax is so that, a moment from now, we will be able to show a nearly one-to-one correspondence between this code and the code that the machine understands and executes. So if you understand this code, you understand how computers work. And if you do *not* understand this code very well, we will annotate it with a ton of comments in a moment, and hopefully that will clear things up …

1.2.2. What Does it Mean to 'Execute' Code?

The following is an explanation of what the computer *does* to make each statement of the C code above happen, in the order that it happens. Note that the discussion uses the expressions **a[i]** and **b[i]** to indicate indexing into the array structures, even though there is no actual 'i' in the calculation … all of the array indexing is done through direct pointers into the arrays. So the expression ***a** corresponds to **a[i]**, because **a** is a pointer that references the i^{th} element of the **a** array, and the expression ***a** means *use the **a** pointer to access the referenced element*; similarly, the expression ***b** corresponds to **b[i]**, because **b** is a pointer that references the i^{th} element of the **b** array, and the expression ***b** means *use the **b** pointer to access the referenced element*.

```
for (sum = 0;// the value 0 is placed into the variable sum to initialize it

// here is the top of the loop (the above 'init' code happens before the loop)

len != 0;     // a conditional branch tests to see if len is zero;
              // if true, then the branch exits (jumps past the end of the loop)
              // otherwise the loop continues to execute

(*a) * (*b); // the value of a[i] is loaded from memory (load-word operation)
              // the value of b[i] is loaded from memory (load-word operation)
              // the two loaded values are multiplied together, and the product
              // is held in a temporary register

sum +=        // that product of a[i] and b[i] is added to the variable sum

   a++;       // the pointer to a[i] is incremented (now points to a[i+1])
   b++;       // the pointer to b[i] is incremented (now points to b[i+1])

len--) {      // the variable len is decremented by one, done at end of loop,
              // followed by an unconditional branch back to the top of the loop
```

Even though this is very low-level programming, and you were just told a moment ago that it is almost a one-to-one match to the machine's low-level instructions, this code is *still* higher-level than what the machine expects. The machine cannot execute this and instead requires something far lower-level still. This C code would be translated by a *compiler* into something that the machine understands, over the course of two different steps.

1.3. Fundamentals of Code Representation

Now we will take the previous example and demonstrate some of the basics of computer organization, i.e., what the computer's underlying hardware does to make the computer execute each of the steps of the computer program. In particular, different levels of the computer use their own languages, and the human user at the top uses yet another language … translation between these levels was once done manually, but today it is automated, which means that you get to write high-level code, while the computer executes low-level code.

To recap, a C-language version of a dot-product operation might look like the following:

```
// int *a contains pointer to one array (at the start, a points to a[0])
// int *b contains pointer to another array (at the start, b points to b[0])
// int len contains the length of both arrays

int sum;
for (sum = 0; len != 0; len--) {
    sum += (*a) * (*b);           // sum += (a[i] * b[i])
    a++;                          // increment a from a[i] to a[i+1]
    b++;                          // increment b from b[i] to b[i+1]
}
```

This first gets translated into executable code in two steps, and once an executable program exists, the hardware can run it.

1.3.1. Step 1: The Compiler Translates High-Level Code to Assembly

In the first step, the compiler translates the C code to *assembly code*. The following is an example of *assembly code* or *assembly language* (or just *assembly*) and represents the first translation from the C code above. We have to pick a specific language, and so we will be using RiSC-16 assembly code in this book … more details will follow, but this should be self-explanatory enough to give intuition behind how just about any computer works.

```
# a-pointer held in register r1
# b-pointer held in register r2
# length held in register r3
# running sum held in register r4
# registers r5 and r6 hold array values loaded through r1 and r2

       add    r4, r0, r0    # load the 'immediate' value 0 into r4           r4 <- 0
top:                        # top of loop body
       bez    r3, out       # jump past the end of the loop when r3 == 0
       lw     r5, r1, 0     # load from mem what r1 (a-ptr) points to   r5 <- mem[r1]
       lw     r6, r2, 0     # load from mem what r2 (b-ptr) points to   r6 <- mem[r2]
       mul    r5, r5, r6    # multiply a[i] and b[i], result in r5      r5 <- r5 * r6
       add    r4, r4, r5    # add product to running sum in r4          r4 <- r4 + r5
       addi   r1, r1, 1     # increment a-pointer (from a[i] to a[i+1])    r1 <- r1+1
       addi   r2, r2, 1     # increment b-pointer (from b[i] to b[i+1])    r2 <- r2+1
       addi   r3, r3, -1    # subtract 1 from r3 (the length value)        r3 <- r3-1
       bez    r0, top       # go back to the top of the loop (r0 always == 0)
out:                        # end of loop - points to first instruction after loop
```

The code above shows the basic operations that the computer performs, and the comments to the right of each operation/instruction explain to the reader what each basic operation does. The first word in each line is the *opcode* (short for *operation code*) and is the *de facto* verb of the instruction, indicating what the instruction does. The opcode is usually followed by registers (temporary storage locations), numbers, and/or labels to be used in the operation. The *registers* are called **r1**, **r2**, **r3**, etc., and are the processor's temporary storage. The *labels* are names that refer to locations in the code — examples above are *top* and *out*. Memory, also called 'main' memory, holds the program's instructions and the data to be operated on — data items in memory are first brought into temporary storage to be used and manipulated.

Many assembly-code instructions have the following form:

```
op  rA, rB, rC
```

This can be thought of as the following:

```
verb   object, object, object
```

The assembly code above translates to the following behavior:

```
rA <— rB op rC
```

In other words, the verb 'op' acts on objects 'rB' and 'rC,' and whatever the result of that interaction, its result is stored into a third object called 'rA.'

For instance, the instruction 'add r1, r2, r3' would mean take the values in registers **r2** and **r3**, add them together, and put the sum into register **r1**. The instruction 'mul r1, r2, r3' would mean take the values in registers **r2** and **r3**, multiply them together, and put the product into register **r1**. An even more common form overwrites a register once it is no longer being used: 'add r2, r2, r3' means overwrite register **r2** with the sum **r2** + **r3**, because, once we read out the value held in register **r2** for the addition operation, we perhaps no longer need that previous value, and so it is safe to overwrite it. This overwriting of registers is often used to implement a running sum, for instance.

Looking at the dot-product code above, the first instruction in the program is 'add r4, r0, r0' … this adds register **r0** to itself, which produces a zero value because we define register **r0** always to have the value 0 in it, and the result '0' is put into register **r4** — we have initialized register **r4** to zero.

The next instruction is 'bez r3, out' … **bez** is short for *branch-if-equal-to-zero*. A branch instruction is the thing that makes *if-then-else* possible. In this case, the instruction indicates that the processor is to test register **r3** to see if it equals the value zero ('0'). If it is, then the computer should go to location **out** in the code, which is at the bottom of the loop. Essentially, this instruction decides when the loop will end: when **r3** contains the value 0 and not before.

The next instruction is 'lw r5, r1, 0' … **lw** is short for *load-word*. This instruction loads a data word from memory. The memory address of the data word is found by adding together the contents of register **r1** and the instruction's *immediate value*, which in this case is the number 0. By 'immediate' we mean *a number that will be present in*

the instruction word itself. In this case, the number zero is the immediate we want to use, and we want to add it to the value in the **r1** register to produce a memory address. The processor will form this address and send it to the memory system. The memory system will use the address to look the value up and return it. The address is effectively an index into the memory array. When the corresponding data word returns from memory, this instruction indicates that the data word should be placed into in temporary register **r5**.

Another load-word instruction similar to the first follows, and that second load-word instruction reads a value from memory and places it into register **r6.** After the two load-word instructions put memory values into registers **r5** and **r6**, the next instruction in the program is 'mul r5, r5, r6' … this instruction multiplies those two loaded values together and leaves the resulting product in register **r5**.

The next instruction is 'add r4, r4, r5' … this adds the product that was left in register **r5** to the running sum (which represents the dot product of the two vectors) that is being kept in register **r4**.

The rest of the instructions are similar to these … the opcode indicates what the processor should do, and the remainder of the instruction indicates the details of how the operation should be performed.

Before we move on, let's compare the two 'real' code forms that we have seen — the C code and the corresponding assembly code — because with some inspection one can determine that the two forms do exactly the same thing. This inspection should give you some intuition as to what a compiler does and how it does it. In the left column of the following table is the annotated C code from above, and in the right column is the annotated (commented) assembly code from above.

C code	Assembly Code	
for (sum = 0; // put the value 0 into the variable **sum**	**add r4, r0, r0**	# initialize r4 to zero
// here is the top of the loop	**top:**	# top of loop body

len != 0; // a conditional branch tests to see if **len** is zero;
// if true, then the branch exits
// otherwise the loop continues to execute
	bez r3, out	# jump past end of loop when r3 == 0

(*a) * (*b); // the value of **a[i]** is loaded from memory
// the value of **b[i]** is loaded from memory
// the two values are multiplied, and their
// product is held in a temporary register
	lw r5, r1, 0
	lw r6, r2, 0
	mul r5, r5, r6
		# load from mem what **r1** (***a**) points to
		# load from mem what **r2** (***b**) points to
		# multiply a[i] x b[i], leave result in **r5**

sum += // that product is added to the variable **sum**	**add r4, r4, r5**	# add product to running sum in **r4**
a++; // the pointer to **a[i]** is incremented **b++;** // the pointer to **b[i]** is incremented	**addi r1, r1, 1** **addi r2, r2, 1**	# increment a-pointer (a[i] -> a[i+1]) # increment b-pointer (b[i] -> b[i+1])
len--) { // the variable **len** is decremented by one, // followed by branch back to the top of the loop	**addi r3, r3, –1** **bez r0, top**	# subtract 1 from r3 (the length value) # go back to the top of the loop
	out:	# points to first instruction after loop

In each case, it should be clear that the assembly code on the right *does* what is described in the comments on the left-hand side. And, if one knows C, it should be equally clear that the comments on the left-hand side do explain what each portion of the C code does. Initializing a variable to zero is straightforward. The label 'top' (a bit of text at

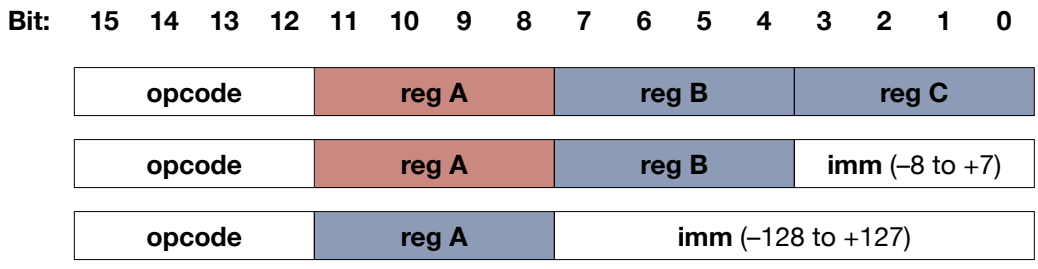
the beginning of a line, followed by a colon) indicates the top of the loop, just as the label 'out' corresponds to the first instruction beyond the loop body. The condition **len != 0** indicates that a block of code should be executed if the length value **len** is non-zero. An alternative interpretation is that the block of code should be jumped over (*not* executed) if the length value **len** *is* zero. As described previously, that is what is done by the **bez** instruction. The sequence of two *load* instructions and a *multiply* is what brings the variables ***a** and ***b** into the processor and produces their product. The following *add* instruction adds the product to the running sum held in **r4**, which corresponds to the variable **sum**. The following two **addi** instructions increment the array pointers, and the last row of the table decrements the **len** value and jumps back to the top of the loop, where the **len** variable will be tasted to see if it has reached zero, which would indicate the end of the loop.

1.3.2. Step 2: The Assembler Translates Assembly to Machine Code

Even though assembly code represents the processor's fundamental instructions, it is not the final form; assembly cannot be executed directly by the computer. It is a human-readable version of what the computer executes. A further translation is necessary, because the processor cannot read text — it must have numbers to execute, not text. The assembly-code representation above, as low-level as it may be, is still higher-level than what the computer wants … and so another translation step is needed.

Accordingly, the second step is the text-to-numbers translation. In this step, an *assembler* takes the assembly code above and translates it into *machine code,* or an *executable*. In this section we present an example of such a translation. As before, the example uses the RiSC-16 architecture for specifics, and more details will be provided in the chapter on microprocessor design.

RiSC-16 instructions are 16-bit instructions, and some of them include immediate values. The three instruction forms, each of which is a 16-bit number, are shown in the figure below. Red coloring indicates that the instruction overwrites the indicated register; blue coloring indicates that the instruction reads the indicated register. White fields indicate constant numbers.



RiSC-16 instructions take three forms: a simple instruction and two ways to form an immediate value

Let's look at each of the three forms in turn:

- Top form — **opcode regA, regB, regC**. This represents most ALU-type instructions, like add, subtract, multiply, divide, etc. The most significant four bits of the instruction make up a 4-bit num-

ber representing the *opcode*. The least significant eight bits of the instruction (the *reg B* and *reg C* fields) make up two 4-bit numbers representing which registers in the register file should be read out and operated upon. Four bits in the middle, bits 8–11 (the *reg A* field), make up a 4-bit number representing the register in the register file into which the result of the operation should be stored.

- Middle form — **opcode regA, regB, immediate**. This form allows an instruction to use a small constant that is known ahead of time, like if you know you want to increment a number by 1. As before, the most significant four bits of the instruction make up a 4-bit number representing the *opcode*. Bits 8–11 (the *reg A* field) are a 4-bit number representing the register in the register file into which the result of the operation should be stored. Bits 4–7 (the *reg B* field) are a 4-bit number representing a register in the register file that should be read out and operated upon. Bits 0–3 represent a constant value between –8 and +7 that will be used in the operation.
- Bottom form — **opcode regA, immediate**. This is used for branch instructions. The top four bits are the opcode, and bits 8–11 represent a register that is to be read out and used, not a register to be written. Thus, that section of the instruction is colored in blue, not red. The bottom eight bits (bits 0–7) represent a constant value between –128 and +127 that will be used in the operation.

The fact that each opcode and register specifier is four bits wide means that a hexadecimal number representing the instruction word should be easily read for its opcode/register contents.

The assembly code we are translating is reproduced below, with the addresses of where each instruction will be found in memory (assume the program starts at location 0), to make it easier to compare with the machine code that follows.

```
00:    add    r4, r0, r0    # load the 'immediate' value 0 into r4           r4 <- 0
top:                       # top of loop body
01:    bez    r3, out       # jump past the end of the loop when r3 == 0
02:    lw     r5, r1, 0     # load from mem what r1 (a-ptr) points to   r5 <- mem[r1]
03:    lw     r6, r2, 0     # load from mem what r2 (b-ptr) points to   r6 <- mem[r2]
04:    mul    r5, r5, r6    # multiply a[i] and b[i], result in r5      r5 <- r5 * r6
05:    add    r4, r4, r5    # add product to running sum in r4          r4 <- r4 + r5
06:    addi   r1, r1, 1     # increment a-pointer (from a[i] to a[i+1])    r1 <- r1+1
07:    addi   r2, r2, 1     # increment b-pointer (from b[i] to b[i+1])    r2 <- r2+1
08:    addi   r3, r3, -1    # subtract 1 from r3 (the length value)        r3 <- r3-1
09:    bez    r0, top       # go back to the top of the loop (r0 always == 0)
out:                       # end of loop - points to first instruction after loop
```

The following shows how these names and symbols are interpreted so that they can be translated into numbers:

```
00:    add=0     r4=4, r0=0, r0=0                    ->   0 4 0 0
top=location 1
01:    bez=6     r3=3, out = distance +9 from here   ->   6 3 +9
02:    lw=4      r5=5, r1=1, 0                       ->   4 5 1 0
03:    lw=4      r6=6, r2=2, 0                       ->   4 6 2 0
04:    mul=11    r5=5, r5=5, r6=6                    ->   11 5 5 6
05:    add=0     r4=4, r4=4, r5=5                    ->   0 4 4 5
06:    addi=1    r1=1, r1=1, 1                       ->   1 1 1 +1
07:    addi=1    r2=2, r2=2, 1                       ->   1 2 2 +1
08:    addi=1    r3=3, r3=3, -1                      ->   1 3 3 -1
09:    bez=6     r0=0, top = distance -8 from here   ->   6 0 -8
out=location 10
```

The final form of the RiSC-16 machine code produced from the assembly code above looks like the following:

```
0400      # load the 'immediate' value 0 into r4                    r4 <- 0
6309      # jump past the end of the loop when r3 == 0
4510      # load from mem what r1 (a-ptr) points to          r5 <- mem[r1]
4620      # load from mem what r2 (b-ptr) points to          r6 <- mem[r2]
b556      # multiply a[i] and b[i], result in r5             r5 <- r5 * r6
0445      # add product to running sum in r4                 r4 <- r4 + r5
1111      # increment a-pointer (from a[i] to a[i+1])           r1 <- r1+1
1221      # increment b-pointer (from b[i] to b[i+1])           r2 <- r2+1
133f      # subtract 1 from r3 (the length value)               r3 <- r3-1
60f8      # go back to the top of the loop (r0 always == 0)
```

The machine code above is nothing more than a sequence of hexadecimal numbers, which may not make much sense to a human, but it is finally something that the computer *can* understand. When you ask the computer to perform an operation, the computer loads one of these numbers from memory, and different parts of that number tell the computer what to do. For instance, in our example above, the sixth line is the hexadecimal number 0445. When the computer reads this number as an instruction, the computer translates it such that the '0' indicates to perform an ADD operation; the last two numbers (the '4' and '5') indicate which registers should be read out and their contents added together (registers **r4** and **r5)**; and the '4' in the middle indicates where the result should go: to register **r4**. We will get into these details in the chapter on microprocessor design.

The comments from the assembly code before are retained in the excerpt above, to explain to a human what each assembly-code instruction gets transformed into, but they are not kept in memory with the executable program.

As it would appear in the computer's memory, the machine code above would look like the following (spaces added for a bit of clarity):

```
0000 0100 0000 0000
0110 0011 0000 1001
0100 0101 0001 0000
0100 0110 0010 0000
1011 0101 0101 0110
0000 0100 0100 0101
0001 0001 0001 0001
0001 0010 0010 0001
0001 0011 0011 1111
0110 0000 1111 1000
```

Again, this is something the computer can understand easily, but a typical human would probably not.

The point of all this is simply to demonstrate that the execution of software, which a human understands by looking at human-readable code, is actually performed by the computer by loading numbers from memory and interpreting them. And those numbers are produced by software: they are a computer-generated translation of the human-readable code.

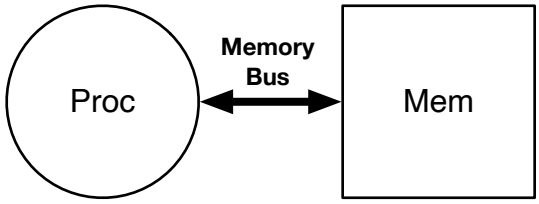
1.4. Fundamentals of Microprocessor Design

At this point, we have a preliminary understanding of what it means for a computer to *execute* your high-level C code (or code in any other language, for that matter). After you write a program consisting of

some statements in that high-level language, you hand the program off to a compiler that produces an executable file. The high-level statements in your language are broken down into lower-level fundamental operations that the computer can do, and this happens over two steps: first, the compiler produces a human-readable version of the fundamental operations called *assembly code*, and second, an assembler translates the assembly code into executable instructions. There are actually more steps than that, but this is the main idea.

The next question you may be asking is *okay, then, what does the processor **do** with that code once it gets it?*

We have previously discussed how high-level code is ultimately translated down to a numeric formulation, and we have asserted that the numeric form makes it possible for the processor to read an instruction and *do* what it says to do. Now we are getting into the field of computer organization, which answers the question above. To begin, a computer looks like this:



A computer is just a processor and a memory connected together

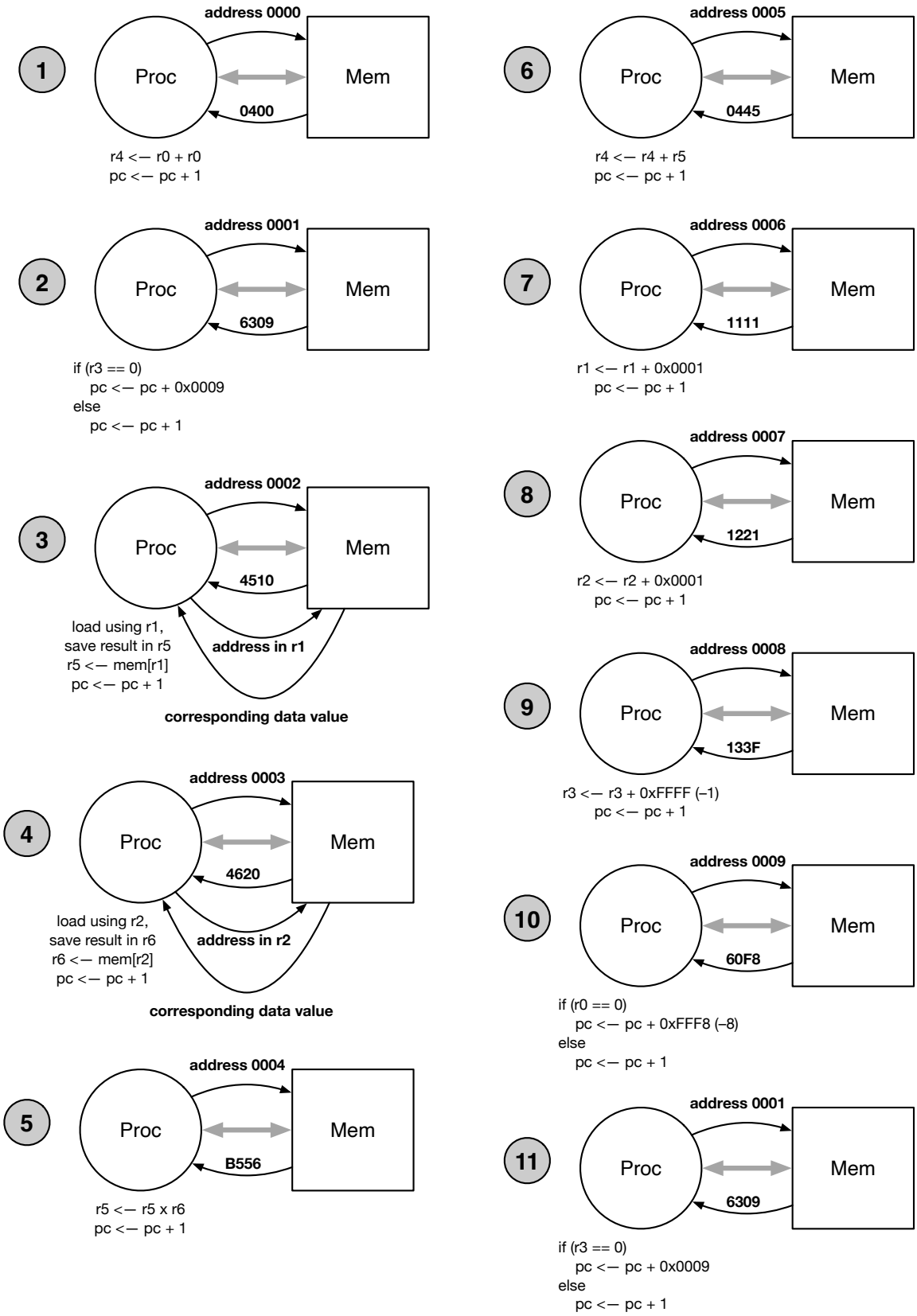
A computer is simply a processor connected to a memory. The connection point is called a *memory bus*, and all of our instructions and data travel back and forth over that bus. The process is termed *fetching* an instruction and then *executing* it … the following explains in detail.

1.4.1. The Processor's Fetch-Execute Cycle

Recall our working example, in machine-code form:

```
00: 0400     # load the 'immediate' value 0 into r4                    r4 <- 0
01: 6309     # jump past the end of the loop when r3 == 0
02: 4510     # load from mem what r1 (a-ptr) points to          r5 <- mem[r1]
03: 4620     # load from mem what r2 (b-ptr) points to          r6 <- mem[r2]
04: b556     # multiply a[i] and b[i], result in r5             r5 <- r5 * r6
05: 0445     # add product to running sum in r4                 r4 <- r4 + r5
06: 1111     # increment a-pointer (from a[i] to a[i+1])           r1 <- r1+1
07: 1221     # increment b-pointer (from b[i] to b[i+1])           r2 <- r2+1
08: 133f     # subtract 1 from r3 (the length value)               r3 <- r3-1
09: 60f8     # go back to the top of the loop (r0 always == 0)
```

One thing has been added since we previously saw it: on the left-hand side is the memory address at which the corresponding instruction can be found (assuming that the program starts at address zero). This address in the left-hand column is what the processor uses to fetch each instruction from memory. The execution of the code example above looks like the following:



The execution of code

The above example shows the execution of the first loop iteration. The steps in the diagram above are described in detail below:

1. The processor sends the program counter to the memory, which tells the memory which instruction it wants to read and execute. We will assume that the first instruction is located at address 0, so the processor sends the 16-bit number 0000 to the memory.

   What returns is the hexadecimal number 0400, corresponding to the assembly-code instruction ***add r4, r0, r0***. The number 0400 is interpreted by the processor to mean 'add the value in register **r0** to the value in register **r0**, and put the result into register **r4**.' This is an *add* instruction (**ADD**). Because register **r0** in this particular architecture has the value zero at all times, the operation is effectively 'put the value 0 into register **r4,**' which initializes **r4** to zero.

   In addition to performing the operation (moving 0 into register **r4**), the processor increments the program counter by the size of the instruction, which is 1 word … so the processor increments the program counter by 1, from 0 to 1.

2. The program counter now has the value 1; this is sent to memory as an address for instruction-fetch.

   Hex 6309 returns: ***bez r3, 0x9***. The processor interprets it as 'see if the value in register **r3** is zero or not; if it is, add 0x0009 (decimal 9) to the program counter; otherwise, just go to the next instruction.' This is a *branch* instruction (**BEZ**), also called a *conditional jump*. If **r3** is zero, we jump 9 memory locations, which would take us just past the end of the loop to the label ***out*** in the assembly code. Recall that register **r3** holds the value of the array-length variable, which we assume is not yet down to zero. Therefore, the processor increments the program counter from 1 to 2.

3. The program counter is now 2; it is sent to memory.

   Hex 4510 returns: ***lw r5, r1, 0***. The processor interprets it as 'use **r1** as a memory address; add 0 to it; send the sum to memory and ask memory to perform a read operation using that address … put the corresponding data that comes back into **r5.**' This is a *load-word* instruction (**LW**): it loads a data word from memory into the processor, for use in a later computation.

   Meanwhile, the program counter increments from 2 to 3.

4. The program counter is now 3; it is sent to memory.

   Hex 4620 returns: ***lw r6, r2, 0***. The processor interprets it as 'use **r2** as a memory address; add 0 to it; send the sum to memory and ask memory to perform a read operation using that address … put the corresponding data that comes back into **r6.**'

   Meanwhile, the program counter increments from 3 to 4.

5. The program counter now is now 4; it is sent to memory.

   Hex b556 returns: ***mul r5, r5, r6***. The processor interprets it as 'read **r5** and **r6,** multiply them together, and put the result into **r5**.'

   Meanwhile, the program counter increments from 4 to 5.

6. The program counter is now 5; it is sent to memory.

   Hex 0445 returns: ***add r4, r4, r5***. The processor interprets it as 'read **r4** and **r5,** add them, and put the result into **r4**.'

   Meanwhile, the program counter increments from 5 to 6.

7. The program counter is now 6; it is sent to memory.

   Hex 1111 returns: ***addi r1, r1, 1***. The processor interprets it as 'read **r1,** add to it the number 0x0001, and put the result into **r1**.' This is an *add-immediate* instruction (**ADDI**). The number 0x0001 comes from sign-extending the four bit immediate value in the instruction word (four bits: 0001), by replicating the top bit of the group of four, which produces a full 16-bit value: 0000000000000001, or 0x0001.

   Meanwhile, the program counter increments from 6 to 7.

8. The program counter is now 7; it is sent to memory.

   Hex 1221 returns: ***addi r2, r2, 1***. The processor interprets it as 'read **r2,** add to it the number 0x0001, and put the result into **r2**.'

   Meanwhile, the program counter increments from 7 to 8.

9. The program counter is now 8; it is sent to memory.

   Hex 133F returns: ***addi r3, r3, 0xF***. The processor interprets it as 'read **r3,** add to it the number 0xFFFF, and put the result into **r3**.' The number 0xFFFF comes from sign-extending the four bit immediate value in the instruction word (four bits: 1111), by replicating the top bit of the group of four, which produces a full 16-bit value: 1111111111111111, or 0xFFFF. This is the 16-bit decimal number –1, which means we are actually subtracting 1 from the value in register **r3**.

   Meanwhile, the program counter increments from 8 to 9.

10. The program counter is now 9; it is sent to memory.

    Hex 60F8 returns: ***bez r0, 0xF8***. The processor interprets it as 'see if **r0** is zero or not; if it is, add 0xFFF8 (decimal –8) to the program counter; otherwise, just go to the next instruction.' Register **r0** is always zero, so we jump back to the top of the loop — the program counter is once again 1.

11. The program counter is now 1; it is sent to memory.

    Hex 6309 returns: ***bez r3, 0x9***. The processor interprets it as 'see if **r3** is zero or not; if it is, add 0x0009 (decimal 9) to the program counter; otherwise, just go to the next instruction.'

Et cetera. As you see, we are back at the top of the loop, and we will continue to repeat the loop until the **len** value in register **r3** becomes zero. At that point we will exit the loop by jumping to the **out** label, which points to the next instruction in the program after the ***for()*** loop.

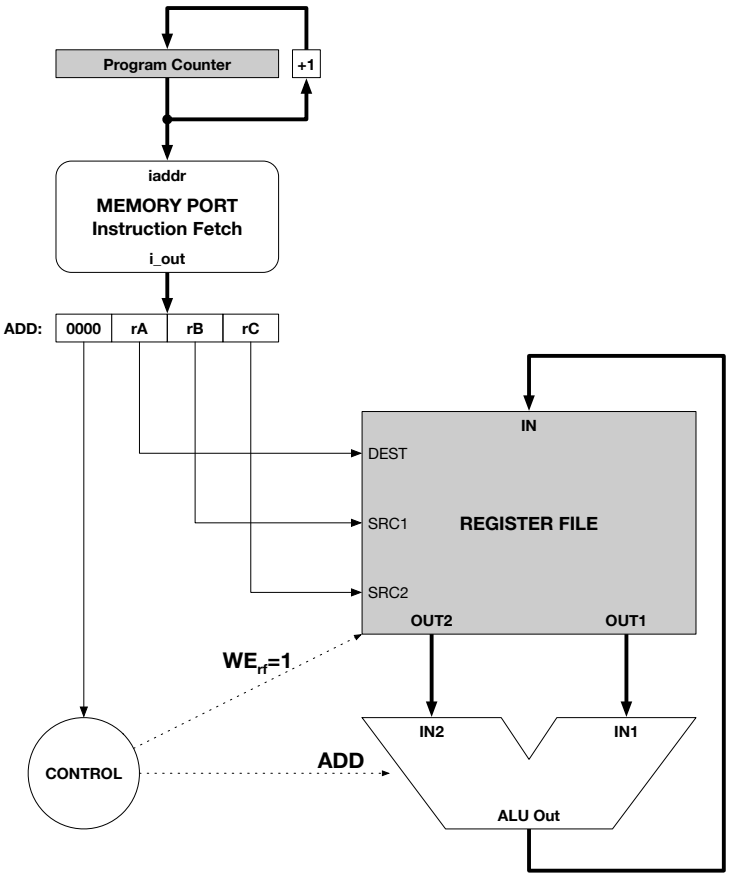
1.4.2. The Internal Architecture of the Microprocessor

Next, what does the processor do? How does it work? Recall what the processor's code looks like. An example instruction — an **ADD** operation — is reproduced below. Its form is similar to other arithmetic/logic operations such as **SUB**, **MUL**, **DIV**, **AND**, **OR**, **XOR**, etc., that appear in most architectures.

```
add       r4, r4, r5    # add product to running sum in r4          r4 <- r4 + r5
```

The processor has a bunch of *circuits* to do the functions required of it. The processor has a *register file*, which is a collection of temporary storage locations ('registers') used as a scratch-pad. It has an *arithmetic-logic unit (ALU)*, which does all of the addition, subtraction, logical-AND operations, logical-OR operations, etc. And it has a bunch of control logic that decides what should happen at each point in time.

For example, the following shows the circuitry (e.g., the processor components and the wiring to connect them up) needed to implement the **ADD** operation:



The components and operations involved in executing an ADD instruction

In the figure, thick lines correspond to 16-bit data values, thin lines correspond to control paths that are less than 16 bits wide, and dotted lines are control paths coming from the computer's decision-making control circuitry. The shaded boxes (Program Counter and Register File) are clocked memory devices — they hold values that can be read and written. They can be read at any time, and on the clock edge, they

record whatever is presented at their inputs (they overwrite their contents, or some subset thereof).

As the diagram shows, the computer sends the program counter to the memory as an address and thereby fetches an instruction word, which is just a piece of data. The initial part of the instruction (the top four bits 0000) are called the *opcode* and indicate to the processor that this is an **ADD** operation. The **rA** portion of the instruction indicates the instruction's 'destination operand,' which is where the result of the operation should be stored (it specifies a temporary register into which we should write the resulting sum). The **rB** and **rC** portions of the instruction indicate two different 'source operands' to be used for input: these are the temporary registers to read out and add together to produce the result that will be stored in the register indicated as the destination operand.

The lines connecting the instruction to the register file are actual wires, sending the information to the register file. The **SRC1** and **SRC2** inputs of the register file cause the register file to read out the corresponding registers and place them on the **OUT1** and **OUT2** data busses connected to the ALU. For example, a 4-bit register number would specify one out of 16 different registers. Whatever numbers are put on these wires — whatever 4-bit numbers are part of this instruction word fetched from memory — would cause corresponding temporary registers to be read and their values sent out on the two 16-bit busses below the register file.

The two outputs of the register file are tied to the two inputs of the *arithmetic-logic unit,* the thick 'V' shaped component below the register file. Thus, whatever values are read from the temporary registers in the register file are fed directly into the ALU for processing. Meanwhile, the *opcode* is sent to the control unit, which interprets it ('decodes' it) as an **ADD** instruction, which means that an **ADD** signal needs to be sent to the ALU.

The ALU receives two register values from the register file and the **ADD** signal from the control logic. It responds by adding together its two inputs, placing the result on its 16-bit output bus, the thick line coming out of the ALU at the bottom and which wraps around to the input of the register file at the top.

The output of the ALU is intended to be stored into the register file. In addition to sending an **ADD** signal to the ALU, the control logic also places a '1' value on the $\mathbf{WE_{rf}}$ control line, indicating that the register file's *write-enable* control should be turned on during this operation. Thus, when the result of the ALU is presented at the input to the register file (labeled **IN** at the top of the register file), the register file ***writes*** the incoming value into the temporary register specified by the **DEST** input on the left-hand size, which is coming from the instruction's **rA** field. This is what was meant when we said that the **rA** portion of the instruction indicates which temporary register should receive and store the result of the operation.

Meanwhile, independently of all this register reading, number adding, register writing, and control logicking, the program counter has been incremented by 1 and updated. This is shown in the upper left-hand corner of the diagram.

All of these operations happen in the space of a computer *clock cycle*. At the beginning of the clock cycle, a new value of the program

counter comes out of the program counter, and that is sent to the instruction memory as an address. This causes the instruction memory to read its contents at that address and drive them onto the *instruction bus*, where the bits and fields of the instruction are distributed to different components of the processor. The topmost bits are sent to the control logic, and the rest are sent to the register file. The topmost bits indicate that this is an ADD instruction, and so the control logic responds by sending an **ADD** signal to the ALU and a '1' signal to the register file on the $\mathbf{WE_{rf}}$ control line. The register file reads out the values from the registers corresponding to the **SRC1** and **SRC2** inputs and sends those values out on its **OUT1** and **OUT2** output busses; the register file output busses are connected to the ALU input busses, and so the ALU takes the two register values as input and performs whatever operation it is told to do — in this case, it is an **ADD** operation. The output of the ALU is fed into the input of the register file, and because the $\mathbf{WE_{rf}}$ control line has been set to a '1' value, the register file input is written to a register, the very register specified by the 4-bit **rA** value coming from the instruction word as the register file's **DEST** input (indicating the *destination* of the operation). The register-file input value is written at the end of the clock cycle. Meanwhile, the program counter (PC) is being incremented, and its new value is written to the PC at the end of the clock cycle.

The end of the clock cycle is also the beginning of the *next* clock cycle, and so once the new program-counter value is written to the PC (the program-counter register), the process begins anew, with the new program counter value being sent to the instruction memory to fetch the next instruction word for execution.

… and so on. The process described above, of fetching an instruction, doing whatever it says to do, updating the program counter, and moving to the next cycle, repeats until the computer is told to shut down.

1.5. Fundamentals of Logic Design

So now you may be asking what is under the hood of these circuit components. The circuit diagram above is just boxes and arrows. What is inside each of these boxes?

The answer: logic circuits — logic gates connected by wires.

While we will show you what logic gates look like, as well as showing you the transistor implementations underneath them, a detailed treatment of logic is outside the scope of this textbook. In terms of actual instruction and assignments, we are going to bypass quite a bit of traditional logic design and jump straight to higher-level concepts (e.g., behavioral Verilog design). Otherwise, it would be difficult to fit everything into a single semester. What this means in practical terms is that we will be focusing more on digital design (the creation of digital circuits) as opposed to logic design (the combination of low-level logic gates like AND and OR, and their optimization). Thus the circuit diagrams comprising boxes and arrows represent exactly the type of system we want to design, and we will focus on the tools to get us there.

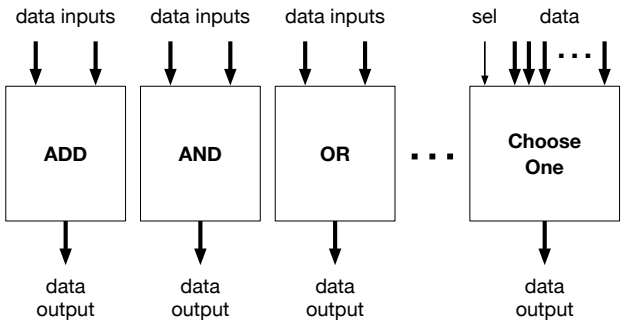
1.5.1. Combinational Logic

One of the distinctions that you are likely to hear about in the literature is that of *combinational logic* versus *sequential logic:*

- **Combinational logic** is when there are no memory primitives like latches or registers or flip-flops present in the circuit — in this type of logic, signals pass through the circuit, and the logical circuit continuously generates an output signal that is a function of its inputs.
- **Sequential logic** is when there ***are*** memory primitives present in the circuit — and so, in this type of logic, the output signal only changes when a clock signal arrives; when the clock indicates, new values are stored in the memory primitives, and the outputs change accordingly … it is similar to the behavior of traffic through a city with stoplights, and the resulting circuits are often called *state machines*.

In this book, we will be building systems that combine these two concepts — in particular, we will build a microprocessor, which is a sequential circuit, a state machine, and its components are divided roughly half and half between *combinational* components (e.g., the arithmetic-logic unit, the control logic) and *sequential* components (e.g., the register file, the memory, the program counter).

Combinational design is done something like the following. Suppose that we want to build an *arithmetic-logic unit* and that we have primitives such as those pictured in the following figure:



Set of primitives

The figure above shows that we have several circuits to choose from:

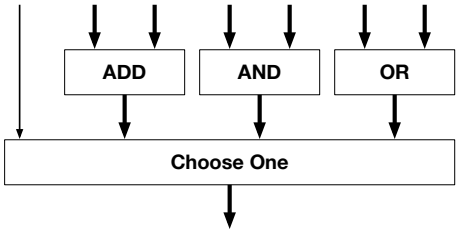
- An **adder**, that takes two 16-bit data inputs, **adds** them together, and outputs a 16-bit sum as its data output (let's ignore the details of overflow for the sake of simplicity)
- An **AND unit**, that takes two 16-bit data inputs, **ANDs** them together, and outputs a 16-bit result as its data output
- An **OR unit**, that takes two 16-bit data inputs, **ORs** them together, and outputs a 16-bit result as its data output
- Perhaps other units as well, in particular units like those described above, which take two 16-bit data inputs, perform some function, and output a 16-bit result as their data output

- A **Choose-One unit**, called a *multiplexor* or *MUX*, that takes some number **N** of 16-bit data inputs, as well as a selector input 'sel' that is a number from 0 to **N**–1 … the circuit outputs the one 16-bit input that corresponds to the number in the select input 'sel' … that is, if the select input is the number 0, then the logic block produces as its output the *first* 16-bit data input; if the select input is the number 1, then the logic block produces as its output the *second* 16-bit data input; if the select input is the number 2, then the logic block produces as its output the *third* 16-bit data input; and so on

How would you combine these primitives in such a way as to produce an ALU, an arithmetic-logic unit? That is a circuit that can perform any of those functions, given an input that identifies which operation to perform. For example, if the ALU's function input is a number representing an **ADD** operation, then the ALU outputs the sum of its two inputs; if the ALU's function input is a number representing an **AND** operation, then the ALU outputs the logical **AND** of its two inputs; if the ALU's function input is a number representing an **OR** operation, then the ALU outputs the logical **OR** of its two inputs; and so on.

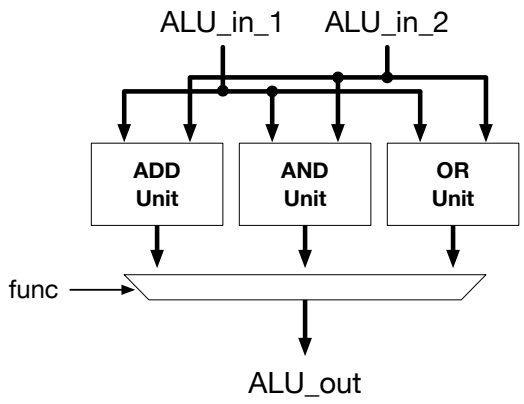
What if we defined 0 to be 'the number representing an **ADD** operation,' and 1 to be 'the number representing an **AND** operation,' and 2 to be 'the number representing an **OR** operation.' Would that make the process simpler?

It should. Given that, and the previous circuit descriptions, it should be clear that the following circuit performs the desired function:



The primitives rearranged to create an arithmetic-logic unit

Again, the 'choose one' function is what we call a *multiplexor (MUX)*, and if we replace the 'Choose One' box above with the appropriate circuit symbol for a MUX, a trapezoid, and wire all of the inputs appropriately, what we get is the following — what a typical ALU circuit looks like on the inside.



The inside of an arithmetic-logic unit

Inside a typical ALU are a bunch of functional units that perform predefined functions. In this example, there is an adder (the 'ADD Unit' in the figure), a unit that performs the AND function, and another that performs the OR function. There might also be a multiplication unit, a divide unit, something that can test equality, perhaps something that can test for less-than or greater-than, and so forth. The ALU's two inputs are sent to each of these functional units, and each does its operation on the inputs it is given, producing its output onto a common bus. The ALU's *function-control* input ('func') is the MUX's *select* input and simply selects one of the results among the various functional units. The bottom trapezoidal figure indicates a *multiplexer (MUX)*, and its function is that of *selection* (it selects one and only one of its inputs to be driven to its output). So, for instance, if the ADD Unit is the first of the multiplexer's inputs, then a '0' input to the MUX would cause the output of the ADD Unit to be driven on the output of the MUX. In the figure above, a '0' would select the ADD Unit's output; a '1' would select the AND Unit's output; and a '2' would select the OR Unit's output. Thus, the *function-control* input to the ALU does not cause the ALU to perform that function — as mentioned, it simply *selects* one of the already-produced outputs.

We could represent this in code (in particular, Verilog, a *hardware description language*) as the following:

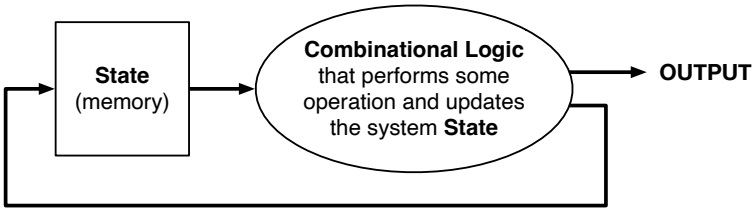
```
module arithmetic_logic_unit (
    input   [1:0]   func,
    input   [15:0]  ALU_in_1,
    input   [15:0]  ALU_in_2,
    output  [15:0]  ALU_out
);

    wire    [15:0]  add_function  = ALU_in_1 + ALU_in_2;
    wire    [15:0]  and_function  = ALU_in_1 & ALU_in_2;
    wire    [15:0]  or_function   = ALU_in_1 | ALU_in_2;

    assign  ALU_out = (func == `OP_add)   ? add_function  : 16'bZ;
    assign  ALU_out = (func == `OP_and)   ? and_function  : 16'bZ;
    assign  ALU_out = (func == `OP_or)    ? or_function   : 16'bZ;

endmodule
```

This code is in the style of *behavioral Verilog* — 'behavioral' because it indicates the desired *behavior* of the circuit and not necessarily the *structure* of the circuit. For instance, the logical blocks from the previ-

ous figures are represented with very simple operators like '+' and '&' instead of numerous logic gates. The lines of code indicate the desired behavior of the circuit but leave the implementation unspecified.

1.5.2. Sequential Logic (Memory Logic)

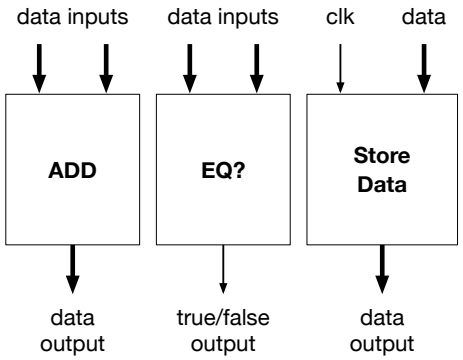
Sequential logic is what happens when we add *state* to combinational logic. Once we can store information ('system state'), the types of problems that we can solve grows tremendously. The general configuration of sequential systems, also called *state machines*, is given in the figure below, which shows a memory structure connected to some combinational logic that performs some function:



A state machine is a bunch of state connected to some combinational logic

Fundamentally, memory structures give us the ability to *store* information indefinitely, as opposed to merely *processing* information as is done with combinational circuits. A *clock* is typically the signal used to cause these memory cells to remember their inputs. Thus, the fundamental unit of computation is what can be accomplished in a single clock cycle: this was seen earlier in the design of the microprocessor.

We present another design example to explain. Suppose we have primitives such as the following:



Set of primitives

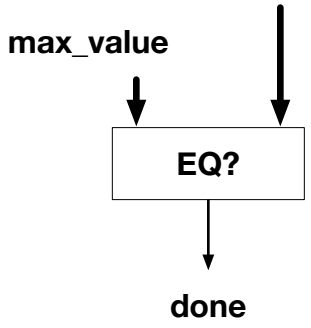
The figure above shows that we have the following example circuits to choose from:

- An **adder**, that takes two 16-bit data inputs, **adds** them together, and outputs a 16-bit sum as its data output
- An **EQ unit**, that takes two 16-bit data inputs, compares them, and outputs a Boolean indication of whether or not the two 16-bit inputs are equal
- A **store-data unit**, for instance a *latch* or *flip-flop* or *register*, that takes a 16-bit data input and a clock signal **clk** and, whenever the

> clock indicates* to do so (e.g., every time the clock transitions from high to low), the circuit stores the value that is at its data input and continues to store that data value and make it available on the data output until the next indication of the clock … and let's assume for the sake of simplicity that the **store data** component is a non-transparent ***register***

How would you combine these primitives in such a way as to produce a circuit that counts up to a given input value and sends a 'done' output signal when the desired value is reached. You can assume that, at the outset, the **register** has the value **0** stored in it.

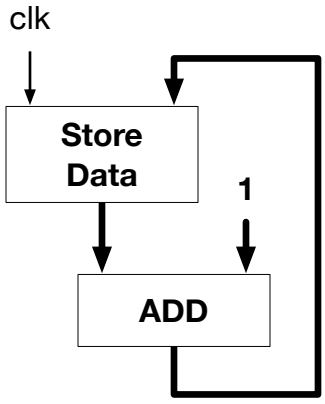
Let's approach the problem in stages. First, this is how one could generate the 'done' output signal, assuming the given input value is called ***max_value***:



Generating the 'done' output signal

We have some value (arriving via the arrow on the upper right) coming into the **EQ** module, and we also have the desired terminal value called **max_value** coming into the **EQ** module. The **EQ** module will output **1** if the values match and **0** otherwise. Thus, the output of the **EQ** module is the desired 'done' signal.

Next, this is how one could perform a count-up function:



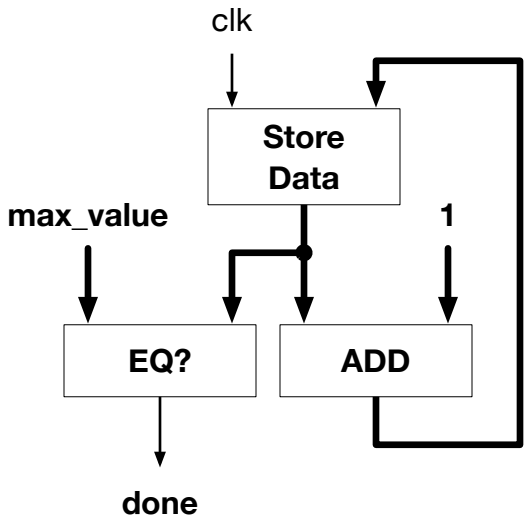
Counting upwards

The **register** stores whatever it is given, on every clock cycle, and the **adder** is adding 1 to whatever it is given as its input. Thus, on every

* For those interested, the technical term for this 'indication' is *strobe*, a word that is used as both noun and verb.

cycle, the **register** outputs a value; the **adder** adds **1** to that value; and the resulting sum is stored in the **register**[*] at the end of the clock.

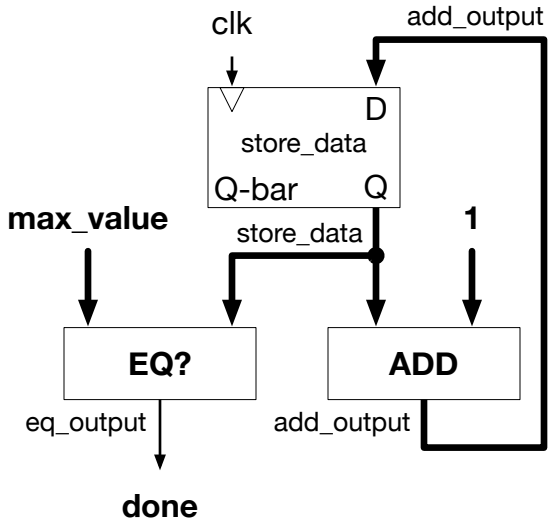
Now we can simply add these pieces together to get the final product:



A count-up unit that indicates when the desired value is reached

If the initial value is equal to the desired **max_value**, then the circuit says so without even performing an addition. As soon as the value in the **register** reaches the desired maximum, the **EQ** component outputs a ***true*** signal indicating that we are 'done' computing.

Let's replace the **register** with its correct circuit symbol, assuming that it is a D-type register.



A count-up unit that indicates when the desired value is reached

[*] In reality, this operation is actually slightly harder than it looks, given the design of the **store data** component, and we will discuss that briefly in the *Logic* chapter. In particular, many types of memory circuits can be *transparent*, which means that the above circuit might have what are called *race conditions* that would affect its correct operation.

The **D** input is for data input; the **Q** and **Q-bar** outputs are for the stored value and its complement, and the triangle input represents the *strobe* or *clock* input.

We could represent this circuit in Verilog as the following:

```
module count_up_timer (
    input           clk,
    input   [15:0]  max_value,
    output          done
);

    reg     [15:0]  store_data;

    wire            eq_output  = (max_value == store_data);
    wire    [15:0]  add_output = store_data + 1;

    always @(posedge clk) begin
       store_data <= add_output;
    end

    assign  done = eq_output;

endmodule
```

There are many ways to improve this, such as using the 'done' signal to halt the process by turning off the clock and thereby stop counting upward, having an 'init' signal that explicitly sets the **store_data** value to zero to begin the counting process, etc. But this should suffice as an initial example.

1.6. Fundamentals of Operating Systems

We have discussed how your program is translated into low-level machine code and how the computer hardware interprets your low-level machine code to execute its individual instructions. We have also shown how to implement the circuits that execute the instructions inside a microprocessor. These topics cover microprocessor operation as well as its implementation, and as mentioned previously, this is what is known as *computer organization*.

At this point, you may be asking yourself some follow-on questions. For instance, how does the computer know that I want to run my software? How does it know what software I want to run? What does it mean to 'double-click' on an icon or type 'grep' on the command line? Indeed, what even *is* the command-line — where does it come from? And how does my computer run lots and lots of programs all at the same time?

The answer to these questions is *the operating system*.

1.6.1. What Is an Operating System?

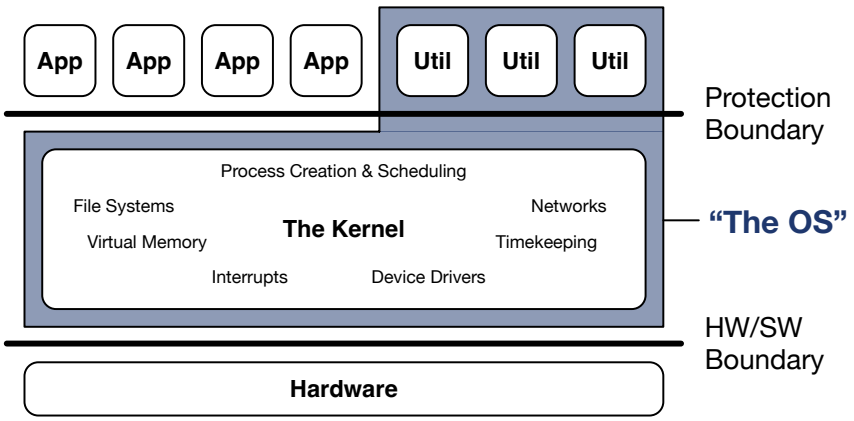
The operating system is several things:

- It is **the hardware's gatekeeper**, managing software access to all shared hardware components so that the various software programs don't step on each other's toes.
- It is **the system's scribe and data guardian**, keeping a record of everything that goes on in the system, enabling data to be long-

lived in the form of *files*, and providing authorized access to those files in a way that guarantees a level of data privacy and security.

- It is **the user's interface to the computer system**, giving you, the user, the ability to browse files and applications and run whatever software you want, either by clicking on an icon or by typing the software's name at a prompt.

The figure below provides an illustration of the structure of the operating system.



The Operating System is the kernel and a collection of user-level utilities

At the top are user-level applications — i.e., what we think of as normal software. These applications are said to be 'user-level' because they do not run with special privileges — they lie on the outside of the protection boundary, and the kernel lies within. When we say 'privileges' we mean that hardware places restrictions on what actions can be taken, what instructions can be executed, and what peripheral devices can be accessed by software running in non-privileged mode. Practically speaking, it means that user-level software cannot do whatever it wants, whenever it wants to. The kernel, shown in the middle, is the only software in the system that runs in privileged mode. The kernel is thus the one piece of software that ***can*** do whatever it wants, whenever it wants to. If a piece of software wants to perform a privileged task, for instance to use a shared resource like the processor, the monitor, the keyboard, the network, the disk, etc. … that piece of software must go *through* the operating system to perform the privileged task, which usually means asking the operating system to perform the task on its behalf.

The following are a few of the essential functions that the kernel performs:

- **Secure Access to Hardware**. As mentioned, one of the kernel's important tasks is that of providing secure access to all of the system's shared resources — for instance, so that one user cannot intentionally or accidentally read or write a file belonging to another, or so that one software application cannot hog all or even most of a particular resource to itself. This is done by having user-level software make *system calls* to ask the kernel to perform hardware tasks on its behalf. By inserting itself into the chain of operations in this way, the operating system is in a position to enforce things

like authorized access, equal sharing of resources, correct usage of those resources (e.g., do not overwrite data that is marked 'read only'), and so forth.

- **Multitasking**. Another one of the kernel's important tasks is that of *multitasking*, the process of running multiple software programs simultaneously on the processor. At the top of the diagram above, there are a number of user-level applications running, and on a computer they would appear to a human user to be running simultaneously. In reality, only one is running at any given instant, and the kernel round-robins the processor between them very very quickly, so that a typical person would never notice. In such an arrangement, the hardware periodically *interrupts* the system (imagine the hardware poking the kernel with a stick), at which point the kernel moves the running task off the processor and moves another task onto the processor in its place. This act of task-swapping happens hundreds of times a second, so what is actually a sequential process (running a little bit of software A for a moment, then a little bit of software B for a moment, then a little bit of software C for a moment, etc., and at some point coming back to software A) appears to the human user to be simultaneous execution of all software programs.
- **Virtual Memory**. Another one of the kernel's important tasks is that of providing *virtual memory* to the software applications. This is both a convenience and a protection mechanism. It allows software to think that its range of memory starts at address 0 and ends at the top of the address range, whatever that might be — we will call it 0xFFFF for now. The practical implication of this arrangement is that when a compiler generates its translated software, it can assume that the software — in fact *all* software — begins at address 0x0000. This makes the job of the compiler ***way*** easier than it would be otherwise. Obviously, though, this cannot be real, because one cannot have two different applications that both want to read/write the same address at the same time; that would invite chaos. The idea of virtual memory is to allow each application to *think* that it owns the entire memory space from 0x0000 to 0xFFFF, and the kernel maintains this illusion, providing a mapping function that translates between the software's *virtual* memory space and the hardware's *physical* memory space. This arrangement also walls off applications from each other, so that they cannot read or write each other's data because they don't even know that those memory addresses exist.

We will explore these essential functions and others in much more detail in the later chapters.

Some of the user-level software programs shown in the figure above are considered to be part of the operating system, even though they are not part of the kernel. To distinguish them, they have been labeled as 'utilities' above. These user-level utilities include administrative programs and services such as data backup, network access, and mail transport. Arguably one of the most important of these OS-utility programs is the system's *user interface*. Your graphical user interface — the thing that shows you icons and lets you move objects around — that is a user-level program. It is a critical part of the oper-

ating system, but it is not inside the kernel. In traditional computer systems, this was called the 'shell' around the operating system: the part of the operating system with which the user interacted. The traditional shell, before graphical user interfaces arrived, was a text-based program that provided a prompt to the user and responded only when the user typed something, typically the name of a piece of software to run.

Accordingly, later in this course, we will build the two most important aspects of an operating system: a *kernel* and a *shell*.

1.6.2. The Operating System Is the First Line of Defense

In early days of computing, the system's 'operator' was the guy who sat on a chair next to the computer and loaded your *card deck* (your program, as realized word-by-word onto pieces of thick card stock) into the computer so that it would then run your program. If there was a line of students waiting to use the machine, the operator might reorganize things a bit. For instance, if the next person to run had a huge card deck (i.e., a large, long-running program), and three people behind him had tiny card decks (i.e., small, short-running programs), then the operator might run the short programs first before getting to the long program. Today, this is called *process scheduling* and is one of the main things that an operating system does to implement fairness, so that no one process can hog the whole machine.

Early systems offered libraries of routines that one could call, for instance to access I/O devices such as the keyboard or printer, but there was no concept of multitasking, and there was no concept of security. Later, the scheduling activity of the system's operator became implemented in software as the 'operating system' … so that multiple processes could be active within the system at the same time and allowed to run in a round-robin arrangement as described earlier. Many of these early *multitasking* systems used a cooperative scheme in which a program had to willingly give up the machine, for instance by making a system call into the operating system. If a program never gave up the machine, then it could take over the computer … early Macintosh and Windows computers from the 1980s were both of this variety. In Windows, it was considered a feature, not a bug, because it enabled gaming software to extract as much performance out of the computer as was possible (because by taking over the computer, the gaming software wouldn't have to share the computer with any other program). Games on Windows computers were extremely responsive, and this is one reason why Windows became the *de facto* standard platform for gaming. However, one can see that this would be a disaster for security.

Later developments in commodity operating systems borrowed from the Unix operating system, developed by AT&T in the 1960s and 70s and which 'did it right' from the outset. Following the lead of Unix, commodity operating systems moved from *cooperative multitasking* to *preemptive multitasking*, in which the hardware helps the operating system recover control of the machine by 'preempting' the running software. As described earlier, the hardware generates an interrupt several hundred times per second, and at the servicing of each interrupt the operating system retakes control of the machine. At that

point, the operating system can decide which process to run next. This arrangement of preemptive multitasking improved system security tremendously, which is why the other operating system vendors adopted it. Unix/Linux, Windows, and MacOS all are security-aware operating systems and have been for several decades.

Following up on the notion that the operating system is the computer system's first line of defense, we can see this in a number of interrelated concepts:

- Users 'log in' to a computer or phone — you have to prove who you are to use the computer, and you have an ID (usually a unique large number) associated with you.
- All activity that you do on the computer, while logged in, is done under the auspices of your ID — it determines which files you can access, how you can access them, what programs you can run, etc.
- Your ID also determines the privilege level/s that you are allowed to activate … for instance, are you allowed to add other users to the computer? Are you allowed to reset other people's passwords? Are you allowed to access other people's files? Typical users can do none of this, but system administrators can. These privileges are attached to the user ID.

Thus, the operating system provides an important first line of defense. If any software component can take ownership of the machine and can access whatever hardware resources it wants, then there is no such thing as security — your files on the machine would be transparent to anyone else on the machine, and that is a non-starter in most scenarios (e.g., business, finance, academia). The operating system is the first line of defense against this.

The next section will get into these aspects of the system in more depth.

1.7. Fundamentals of Microsystem Security

At this point, we have established that the process of running software on your computer involves the following steps/aspects:

1. You write your code in a high-level language
2. Your high-level code is translated by a *compiler* down to a low-level representation in assembly language
3. The assembly code is translated by an *assembler* into a sequence of numbers that the computer understands to be instructions … this form is called *machine code*, or an *executable*
4. The machine-code instructions enable the processor to perform your high-level operation, one tiny step at a time
5. When you tell your computer to execute a block of instructions, otherwise known as an 'app' or 'program,' the computer fetches those instructions one at a time from memory, interprets them, and then does whatever each instruction indicates … for instance, some instructions move data between the processor and memory;

others operate on the data held locally in the processor, such as addition, multiplication, etc.

6. When a processor is said to 'do' what an instruction indicates, it is actually the case that portions of the instruction (via the instruction bus) are directly wired to different components in the processor, and different values on those wires make the components do different things … for instance, different numbers in the opcode field make the ALU produce different values as its output (addition, subtraction, multiplication, division, logical-AND, logical-OR, etc.), and different numbers in the register fields make the register file read and write different temporary registers
7. Circuits made of logic gates create all of these components: the memory structures, the data-crunching arithmetic/logic structures, and the control structures that direct all of the activity and data movement — all of these are made of logic gates
8. While we have not yet discussed the following, we will: the logic gates are implemented as sets of *transistors* wired together in various topologies, such that they produce a desired logical output … transistors are switches that are controlled electronically
9. The design of these digital circuits is typically done at a high level, using behavioral expressions to indicate what *function* the circuit should perform, as opposed to indicating the precise circuit *structure* that would perform the function
10. If one wants multiple apps running on a computer, or multiple users, then one needs an *operating system* … the operating system manages all of the activity in the system — among other things, it reads the disk and runs the programs that we request of it
11. The operating system is also responsible for protecting the system's users from each other, and that is accomplished by protecting each user's data and programs from the programs of other users — for instance, one user's program may not read or write the data of another user, without that user's explicit permission
12. Users are represented by their unique user IDs, and the operating system maintains a database of user IDs and their corresponding capabilities … this information dictates what a specific user is and is not allowed to do (e.g., add users to the system, change the passwords of other users, etc)

These are the fundamentals of modern computer systems — meaning microprocessors running operating systems and user-level software. We will have you build a system from the ground up, and we will also show how these systems can be attacked and defended.

1.7.1. Some Basics

First, some fundamentals to put other discussions into perspective.

User Authentication

As described above, to use a computer running a 'real' operating system, a person must log in via some password-type mechanism (could be biometric, etc.), which establishes a user ID for the length of time

that person is logged in. From that point on, until the person logs out, all activity initiated by that person will be tagged by the user ID, including authentication checks to see if the I/O subsystem requests (e.g., to files, to the network, etc.) are allowed. This is the first line of the defense that the operating system uses to keep track of who is allowed to do what in the system.

Privileged Mode

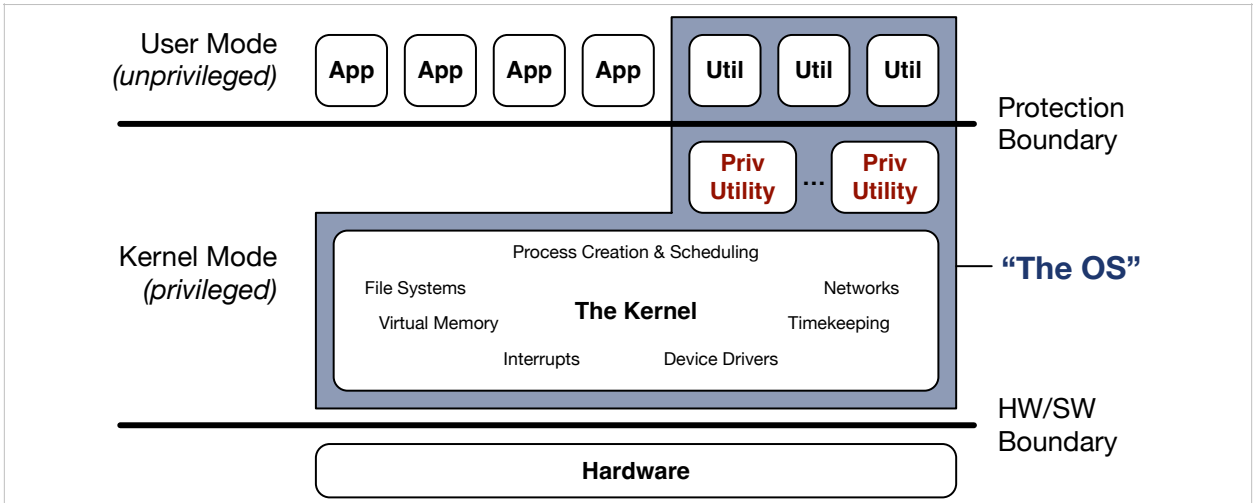
The kernel runs in a 'privileged' mode: while certain instructions, memory locations, and I/O devices are off-limits to normal software, they are ***not*** off-limits to the kernel. The privileges are enforced by the hardware: a 'privileged' bit is set to an on/off state in some configuration register (note: some systems may have multiple levels of privileges), and the state of that bit (or bits) determines the types of things that the currently running software is allowed to do. If the privileged bit is ***off***, then privileged instructions cannot be executed; privileged memory locations cannot be read or written; most I/O devices cannot be accessed. If, however, the privileged bit is ***on***, then everything is allowed.

When the computer boots, the privileged bit is on, and the first piece of software to run in the system should be the operating system's kernel, which is trusted. As part of its startup protocol, it initializes everything in the system, and the last thing it does before handing the system over to user code (the shell) is to turn that privileged bit off.

The primary way to turn the privileged bit back on is to *interrupt* the kernel — this is a hardware function that invokes the kernel directly. Because the kernel is trusted, and because an interrupt causes code inside the kernel to run (it is like a hardware function call into the kernel), a by-product of calling an interrupt is the enabling of privileged mode, so that the kernel regains full control of the system. When the kernel finishes handling the interrupt, it returns back to user-level software via a *return from interrupt* function, which makes sure to turn privileged mode off before resuming user-level code.

Privileged Utilities

Note that the previous paragraph said 'primary' way. It turns out that, in reality, some operating system utilities are allowed to run in privileged mode, even though they are not part of the kernel. This is an OS design choice — it is clearly not a necessary component in a system, but it is a design choice that most modern operating system designers make. Thus, the figure from above should be redrawn as follows to portray more accurately how modern computing systems are actually arranged:



The Operating System comprises the kernel and a collection of user-level utilities

Privileged utilities include most servers and administrative programs. As we will see, the existence of these privileged-level utilities causes *significant* security problems.

1.7.2. Attacking and Protecting the Logic

Many of the ways to cause logic to malfunction have been discovered, and most (but not all) of these are accounted for in modern designs.

Forbidden (Unaccounted-for) States

Logic circuits can have vulnerabilities in them if they are designed in such a way that certain states are not expected and are therefore not accounted for. If it is possible to drive the logic intentionally into one of those unexpected states, then an attacker could cause the hardware to behave in an unexpected/undefined manner … which could include anything from generating incorrect results to crashing, and quite probably a mixture of both.

One example: a portion of the system could be in one of some number of states, and each state is represented by a 4-bit number. This means that there are 16 possible states. However, assume the designer only used 12 of those numbers, because there are only 12 well-defined states that the machine can be in at any given time. Somewhere is a register that holds the 4-bit state number in it … if it is possible to corrupt that register so that its state value becomes an invalid state, then it is possible to attack the system, meaning one could cause the machine to misbehave and act in an undefined and unexpected manner.

Another example: a MUX is designed to choose one of 7 different values for output. Let's say that this is in the implementation of an arithmetic-logic unit like the one described earlier. The *select* input to the MUX has to be at least a 3-bit value, but given that the MUX only chooses between 7 inputs, that means that one of the *select* states is invalid. If it is possible to drive onto the *select* line an invalid choice (e.g., to instruct the MUX to choose the 8th input, which doesn't exist), it would cause the MUX to behave in an undefined manner; i.e., it would be possible to attack the system. An ALU is often driven by the values in the instruction opcodes … thus, one could try to execute

numerous instructions with invalid opcodes in them, to see if any of them cause the machine to freak out.

The solution to these types of attack is to account for all possible states. In the *Logic* chapter, we will show how this is done in Verilog.

Data and Control Inputs

All wires are antennae … this means that all wires in a system will *radiate* (transmit) whatever signals they are carrying, and they will also *couple with* (receive) all nearby electromagnetic emissions and then carry those received signals back to whatever electronic circuitry the wires are attached to.

So let's consider as some very important wires the pins on a chip and the traces on a circuit board that connect to them — in particular, let's consider those pins that are connected to the chip's *data* and *control* inputs. If we are able to blast those wires with EM radiation, then it is possible for us to cause incorrect inputs on those pins. Overwhelming noise is the easiest signal to use in this scenario, so that is what we will consider — and it should be obvious that such an attack would succeed and cause the system to misbehave. Instead of receiving the desired signal values, the chip receives random garbage, which is not the desired result.

A solution is already in place, in two parts. The first is the overdrive portion of the scenario, which is required for electrostatic discharge protection (ESD). Whenever you touch a chip, you have some amount of latent static electricity in you, and even a small amount may be enough to overwhelm the chip's inputs and destroy the I/O circuits there. To protect from this scenario, chip inputs have diodes connected to the power rails so that overwhelmingly positive voltage spikes get shunted to the positive voltage rail (usually called VDD), and overwhelmingly negative spikes get shunted to the negative voltage rail (ground).

So ESD protects the inner circuits from the high-voltage aspects of our scenario. What ESD does not protect against is the garbage data values on the line due to noise — ESD will protect against spikes, effectively chopping them off above a certain value, but the garbage value will still remain. We protect against noise, or alternatively data corruption, by using any number of solutions, such as parity bits, checksums, error correcting codes (ECC), differential signaling, etc. These can protect against both malicious attacks and simple noisy environments.

Clock Inputs

The heart of any digital system today is its clock signal. If the clock is corrupted, then none of a chip's internal circuits will work correctly. Thus, it is imperative to protect the clock from attack.

However, while we can correct virtually any amount of corruption on control and data lines through the use of various redundant codes, we cannot do the same with the clock input. This is because the clock, unlike commands and data, is not a 'data' input — it is not received at a register on the receiving end and then interpreted, but rather it is allowed immediately into the chip, where it is used directly to control the chip's internal registers.

Because of this, any corruption of the clock signal (e.g., phase shifting it or simply putting a ton of noise on it) will cause a chip to malfunction. We will get into details in the *Logic* chapter.

1.7.3. Attacking and Protecting the Microprocessor and its Subsystems

There are numerous ways to attack the hardware at a higher level than the circuit behavior. These are but a few.

Handling Privilege Levels in a Pipelined or Multi-Issue Microprocessor

As described above, nearly all microprocessors today have a facility in which a 'privileged mode' bit in the processor determines whether or not certain instructions can be executed, or certain regions of memory can be accessed. In addition, most high-performance microprocessors can execute numerous instructions in parallel — either in a processor pipeline reminiscent of Henry Ford's assembly line, in which many instructions are in various stages of execution simultaneously; or through multiple instruction issue, in which more than one instruction is fetched and executed at the exact same time.

The problem that such an arrangement poses for security is that the privilege-level bit is singular, and yet in high-performance designs there is more than one instruction currently executing in the microprocessor. What if user-level code calls the operating system, an action that generates an interrupt and thereby turns privileges 'on' … and the user-level code is thereby given the ability to run at the privileged level? One could imagine the following sequence of instructions:

```
sub    r1, r2, r3
bad1   r4, r5, r6
syscall
bad2   r7, r8, r9
or     r1, r2, r3
```

Let us suppose that the **bad1** and **bad2** instructions are not allowed by the hardware because they perform disallowed functions, or they access disallowed memory locations. The **syscall** instruction (which in many instruction sets is called 'trap') is the one that interrupts the operating system, a side-effect of which is the enabling of privileged mode.

What could go wrong here?

What we do *not* want to happen is the enabling of either the **bad1** or **bad2** instructions. In addition, the fact that the **bad1** instruction is disallowed, no instructions following it should execute, which means that the syscall should also not happen. Hardware would normally recognize the **bad1** and **bad2** instructions as disallowed and refuse to execute them — it would also notify the operating system of the infraction (these represent forbidden actions), so as to have the offending user-level program terminated. However, if the microprocessor executes numerous instructions simultaneously, then several things are possible, including the following:

1. The processor could interpret and execute the **syscall** instruction before it recognizes that the **bad1** instruction is a disallowed instruction.

2. The privilege escalation that is enabled by the execution of the **syscall** instruction could put the entire microprocessor into a privileged mode, thereby giving privileges to the user-level instructions that are still in the machine, which could include both **bad1** and **bad2**.

This clearly is not desired behavior. Microprocessor designers must explicitly avoid such scenarios. The solutions are multi-fold.

- When interrupts are taken, the processor handles them in a 'precise' manner, which means that all of the instructions before the interrupting instruction are allowed to finish, and then the interrupting instruction is handled all by itself. At that point, all remaining instructions, meaning the ones that come *after* the interrupting instruction, are flushed from the processor pipeline, so that when the operating system begins executing, there are no other instructions in the microprocessor. Thus, in a correct implementation, the interrupt that the **bad1** instruction generates is handled in a precise manner, which means that the **syscall** instruction, the **bad2** instruction, and all following instructions, are flushed from the pipeline before the interrupt from **bad1** is serviced. Thus, the processor will never complete **syscall** or **bad2**. Similarly, the interrupt that the **syscall/trap** instruction generates is handled in a precise manner, which means that all instructions before it must finish before the **syscall** interrupt is handled. This means that the **bad1** instruction must be completely finished before the **syscall/trap** takes effect, and because the **bad1** instruction causes a terminating interrupt, the **syscall** interrupt will (should) never happen in a precise pipeline.
- In complex processors, one could envision the privileged-mode to be attached to each instruction individually. For instance, when an instruction is fetched into the microprocessor, the state of the global privilege bit could be copied and stored alongside that instruction. When the instruction is executing, the microprocessor would not check the system-wide privilege-mode setting to see if the instruction is allowed but would instead check the instruction's local copy of the privilege-mode bit.

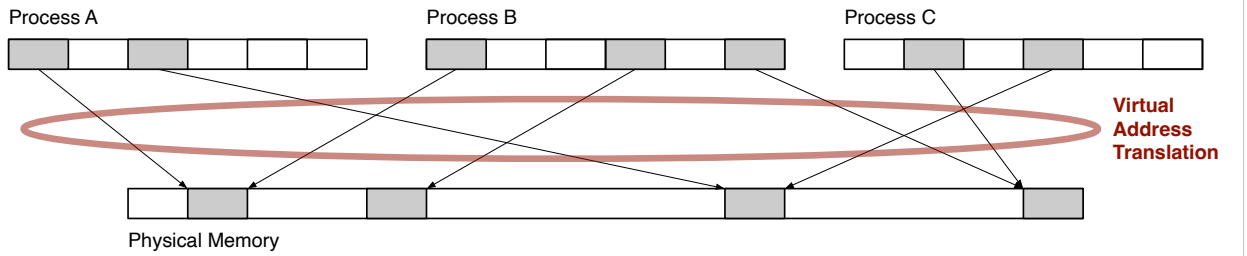
We will get into more detail in the *Pipelined Microprocessor* section.

Virtual Memory and I/O

The kernel protects the memory system by using a facility called *virtual memory*, devised in the 1960s to automate the process of moving data back and forth between main memory and the disk system. In those days, main memories were several thousand bits of information, not much at all, and program sizes were such that a single program's data set might exceed the size of all of main memory — something *very* unlikely to happen on your laptop today. In those days as today, the disk was used as a backup for main memory, and at that time programmers had to manage their memory space explicitly, meaning manually writing code in their own programs to do it. The invention of virtual memory moved that explicit management of memory into the operating system kernel, and the hardware provided support by

requiring all application memory requests to go through a translation step managed by the operating system.

The following figure shows the idea. Three processes, A, B, and C, have virtual addresses that are mapped onto physical memory at the granularity of virtual pages. The virtual addresses are translated at run time by the hardware and operating system from the process virtual address spaces onto physical memory. Sharing is possible (e.g., one can have shared libraries and application binaries) but not required.

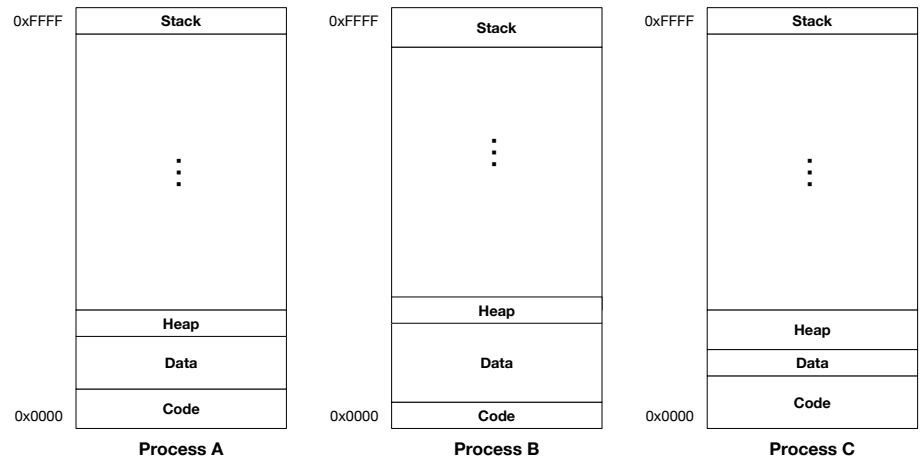


The general concept of virtual memory: translation from a virtual space onto a physical space

Today, the kernel uses this facility to control software's access to memory: no application, unless running in privileged mode, can access physical memory directly. All memory requests (the **load/store** instructions from earlier) must go through a translation step that is managed by the operating system, and the operating system makes certain regions of memory off-limits for certain programs and user IDs. Thus, the kernel can protect itself from user-level applications, and the kernel can also protect user-level applications from each other.

In addition, modern systems typically use 'memory mapped I/O' which means that access to hardware controllers for external devices (disks, keyboard, monitor, network, printer, etc.) all go through the memory system, using load/store instructions. Thus, the same facility that protects memory from errant user applications, whether malicious or faulty, is used to protect a modern computer's I/O subsystem as well. In modern operating systems, all I/O accesses must go through the operating system, because, thanks to the protection aspects of virtual memory, an application is unable do I/O directly on its own.

The general notion of virtual memory is shown in the figure below, which illustrates three different user-level processes, each having its own virtual environment, an address space ranging from address 0x0000 to address 0xFFFF (or whatever the largest integer happens to be):

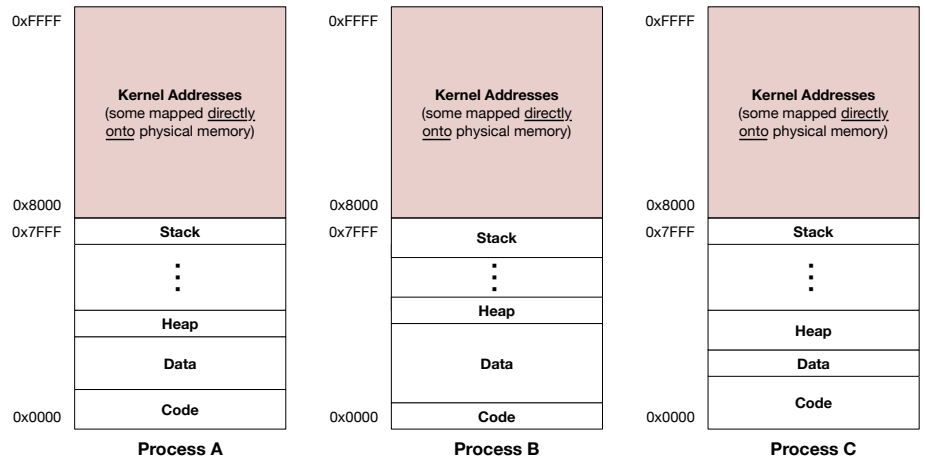


The theory: processes execute in virtual address spaces that translate every address between zero and *max integer*

The software uses numeric addresses to reference locations in its address space, and these numeric addresses are allowed to range over the entire space that an integer can represent, from zero to the largest integer that exists. These are called 'virtual' addresses because they only have meaning within the process's virtual address space and must be translated into physical addresses before they can be used for accessing memory (to read and write data). Any valid virtual address is translated by the hardware and the operating system into the corresponding physical address.

Spectre/Meltdown Vulnerabilities

Here is the problem: modern computer systems break this model. Instead of treating all numbers in the range [0 .. max] as virtual addresses to be translated, modern hardware and software, for purposes of convenience, have divided the address space into **user** and **kernel** partitions, as shown in the figure below.



The reality: user processes have direct access to kernel addresses; they just aren't supposed to use them

Here, the top half of the address space, consisting of all addresses having a '1' in their topmost bit, is considered to belong to the kernel and is therefore off-limits to the user process. The user's address space only ranges over the positive integers — all addresses with a '0'

in the topmost bit. Thus, in modern computer systems user addresses range from 0x0000 to 0x7FFF instead of from 0x0000 to 0xFFFF … this was a deliberate design decision, made to enable the implementation of 'memory-mapped' I/O operations via ***load/store*** instructions instead of special legacy I/O instructions, and to facilitate the kernel's ***copyin/copyout*** implementation, such that it would require no special virtual-to-physical ***load/store*** instructions.

Note that, with the previous figure (the 'theory' illustration), a user application had no notion of physical memory — all of its addresses would be virtual by definition. In the figure immediately above, however (the 'reality' illustration), we have **exposed** kernel addresses directly to the user application. This change has devastating effects on security: it is the fundamental reason that the *Meltdown* exploit exists. Previously, there was an impenetrable level of security in place: a user process, by definition, could not even *name* a physical address. When systems expose kernel addresses as illustrated above, however, the user process ***can*** name physical addresses, and the impenetrable level of security is consequently dissolved. The only security to prevent the access of physical memory is the microprocessor's privilege check, and Meltdown circumvents that, by using speculative execution to break the processor's rules without getting caught. Later we will have a project on writing a Meltdown exploit instance, but the following code should give an idea of how Meltdown works.

```
char array[256];

for (PA=0x8000; PA<=0xFFFF; PA++) {
    flush cache
    if (0) {
        load-byte-using-physical-address  r1, PA
        load-byte                         r0, r1, array
    }
    for (i=0; i<256; i++) {
        t0 = now();
        x = array[i];
        t1 = now();
        if t1-t0 is low, we know that MEM[PA]==i
    }
}
```

This is *Meltdown*, fundamentally. For each physical address that you want to read, you flush the cache, read the physical address, and then use the value you read to access an element in your array. Later, you sweep through the array … and because you previously flushed the cache, all of your array elements will be uncached and will therefore take a long time to load. One value, however, let us say the n^{th} value, will be cached and will take less time than all the others to load. That is the value that was used to access an element in your array. N is therefore the value that is found at that particular physical address.

There are a few obvious questions. First, what is going on with the **if (0)** statement? It turns out that microprocessors will forgive you for performing illegal operations if they are performed mistakenly … so the illegal code in that **if** statement was not supposed to execute, and therefore, if it is ever mistakenly executed, then the user application will be forgiven and will **not** be terminated for trying to execute illegal operations. The trick, then, is figuring out how to fool the microprocessor into executing this illegal code … in the simple processor pipeline that we build in this course, we will implement a very simple

form of branch prediction: *predict-not-taken*, which simply predicts upon reading a branch that the branch will not be taken and that, instead, the microprocessor should go down the *not-taken* path. Thus, one can implement the code above in the following manner:

```
        bez                                 r0, next
        load-byte-using-physical-address    r1, PA
        load-byte                           r0, r1, array
next:   ...
```

When the microprocessor encounters the **bez** instruction, it predicts that the branch should NOT be taken and keeps executing down the sequential path — thus, it might fetch and execute the two **load-byte** instructions that follow immediately after the branch. This will be a misprediction, because the branch says to jump to the ***next*** location if **r0** is equal to zero. Register **r0** is always equal to zero, so this branch is definitely supposed to jump to the ***next*** location, thereby jumping over the two **load-byte** instructions.

At some point, the microprocessor realizes that a mistake was made: the branch was mispredicted, and therefore the processor must rewind and remove from the pipeline any speculatively executed instructions — e.g., the two **load-byte** instructions. Then the microprocessor revisits the branch instruction, instead implements a *taken* branch, and jumps to ***next***.

Modern branch predictors are far more complex than the simple example above, but they can be fooled into mispredicting any specific branch, and the literature is filled with such examples on how it can be done. Thus, getting the microprocessor to execute illegal instructions on our behalf, and getting away with it, is relatively simple to do.

Another question is *how did we successfully retrieve the value that we loaded from a physical address?* Should that not have been disabled or disallowed somehow? The answer is yes, it was indeed disallowed. The microprocessor does not allow data results from speculatively executed instructions to modify any *architectural state* (the program counter, the register file, the memory and I/O system). This rule was not broken. We did not directly retrieve the value obtained from the first **load-byte** instruction. It was passed to the second **load-byte** instruction, but the loaded value was never passed directly to any instructions outside of the invalid **if (0)** statement.

The key word here is *directly*.

The value was not passed directly, because the speculative instructions were not allowed to modify *architectural state*. However, the processor cache is ***not*** considered to be architectural state, and therefore the speculative instructions *were* allowed to modify *that*. Our code at the end, the ***for()*** loop that sweeps through the array and does a series of timing tests, is an example of ***side-channel analysis*** that gets values from places *other than* architectural state. As the example above illustrates, these sources can be just as effective as architectural state as passing data that would otherwise be illegal and/or out-of-bounds to user applications.

HOWEVER — this is all well and good, but it overlooks the most important point, which is that the following instruction is even ***allowed to exist*** in modern microprocessors:

```
load-byte-using-physical-address r1, PA
```

Why does this exist? Why are user applications even ***allowed*** to generate physical addresses? As mentioned earlier, this was a change made to simplify system implementations. However, it quite clearly created an ENORMOUS security hole. In a system where a user application can *only generate virtual addresses*, then there is a 100% guarantee that the user application cannot possibly read memory locations outside of its virtual sandbox. The decision to reveal physical memory to the user application was *incredibly* short-sighted, and it was most likely allowed because hardware people do not know much about security and vice versa. Meltdown and Spectre only happened because we opened the door for them.

Side-Channel Analysis, More Generally

In addition to normal input/output channels like the computer's monitor, printer, memory channel and disk channels, a computer has a number of unintentional 'output channels' that can convey significant and detailed information about the current state of the machine. These output channels include the computer's moment-by-moment signature in the electromagnetic spectrum, the amount of electrical current it draws at any given time, the amount of heat it generates at any given time, the time that it takes certain operations to execute, the sounds that it can make (keyboard noises, hard disk clicking, fan turning on and off), the state of the processor cache, and so forth. These unintentional outputs, called *side-channel information* or just *side channels*, can convey to an aware listener more or less exactly what the computer is doing at any given moment, and numerous studies have demonstrated exploits that use these side channels to deduce data values such as passwords and otherwise protected information.

Solutions to the side-channel problem today tend to be reactive: when a side channel is discovered and either published or exploited or both, then a solution is developed by the designers of the hardware and/or software architecture that was compromised. Software compromises are far more common than hardware compromises, which is why it is good to keep one's operating system up to date, but hardware security holes do exist (e.g., *Meltdown* and *Spectre*), and when these are discovered, their solution can require all new hardware and not just a simple firmware update.

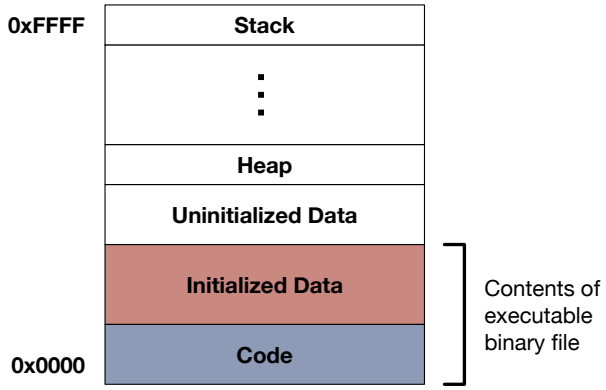
Later, we will describe a proposed defensive mechanism that is more proactive than reactive, a solution in which one publishes far less about one's hardware specifics — because detailed knowledge of the hardware specifics is precisely why side channels are so easily discovered and exploited. Therefore, it stands to reason that publishing less information about the hardware specifics would, by definition, reduce a system's exposure on this channel.

1.7.4. Attacking and Protecting the Programming Language

One of the main ways that systems are attacked is through the ***stack***. In these attacks, one can exploit operating system utilities, which are user-level applications that run with elevated privileges so that they can perform important tasks on the behalf of the operating system.

Essentially, these utilities *are* part of the operating system, but they are not housed inside the kernel.

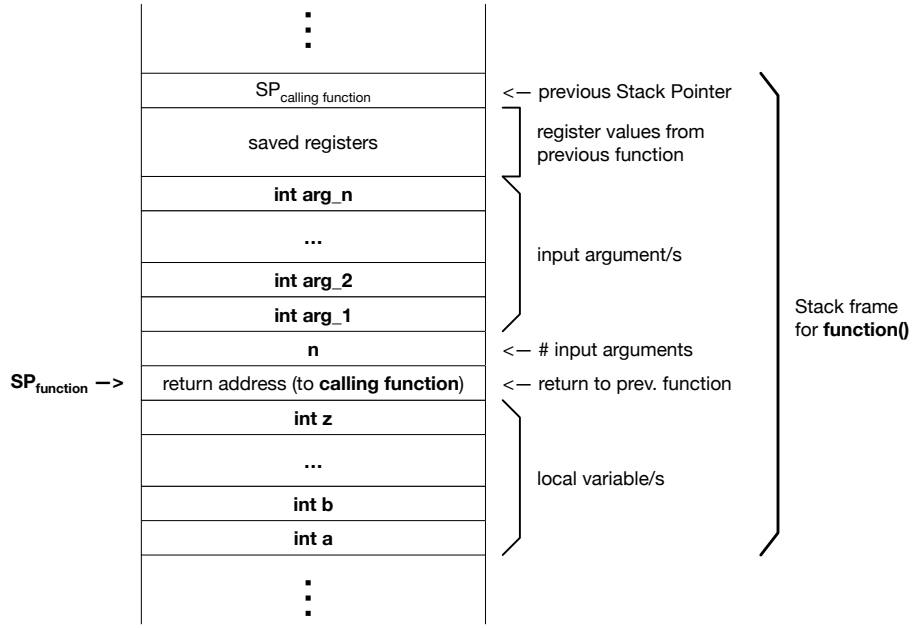
As mentioned, attacks on these utilities can be made through the stack. First, a quick overview of what a stack is. The figure below shows the memory layout of a typical Unix process:



Program layout for a typical Unix process — a process *address space*

A program starts at the bottom of memory, e.g., address 0x0000, and ranges all the way to the top of memory — in this example, 0xFFFF. The program has *code* and *data* segments, which come from the application file on disk. The *heap* and *uninitialized data* segments contain additional data items that the program is working on. At the top of the address space is a *stack* segment, and this region of memory has a very specific organization. It holds a number of ***stack frames***, one for every function that is currently active. For instance, if ***main()*** calls ***function1()***, and if, while ***function1()*** is executing, it calls ***function2()***, then at that point there are three functions active, and each has a stack frame on the stack.

The general layout of a stack frame looks something like what is shown in the following figure:



A general stack frame for a generic function — SP is the stack pointer

We will describe this from the top down. At the top of the stack frame are the values put there by the calling function:

- A saved copy of the previous function's stack pointer (termed the 'calling function')
- The calling function's *state* at time it made this function call — these are register values saved from the register file
- The set of argument values to be given to the called function
- Perhaps an indication of how many arguments are in that set

The rest of the stack frame comes from the called function itself:

- The return address that will take the program back to the calling function
- Any local variables that the function may have as automatic variables

Can this arrangement be attacked? Yes, it can … if we discover a privileged utility that uses *unsafe* system calls, invoke it, and intentionally misuse one of those unsafe system calls, then we can take over the machine. But what are 'unsafe' system calls? And how is it that we take over the entire *machine* instead of simply taking over the poorly written *application*?

Regarding 'unsafe' system calls, sometimes software developers, even those developing operating system code, forget to check how big a buffer is before writing into it. When software does this, it can be vulnerable to a *buffer-overflow* attack. Note that, in the diagram above, the function's local variables are on the stack and are located at addresses that are *lower* in memory than the function's return address. For instance, let's assume that **int z** which is one of the function's local variables, is a four-byte quantity and is located immediately below

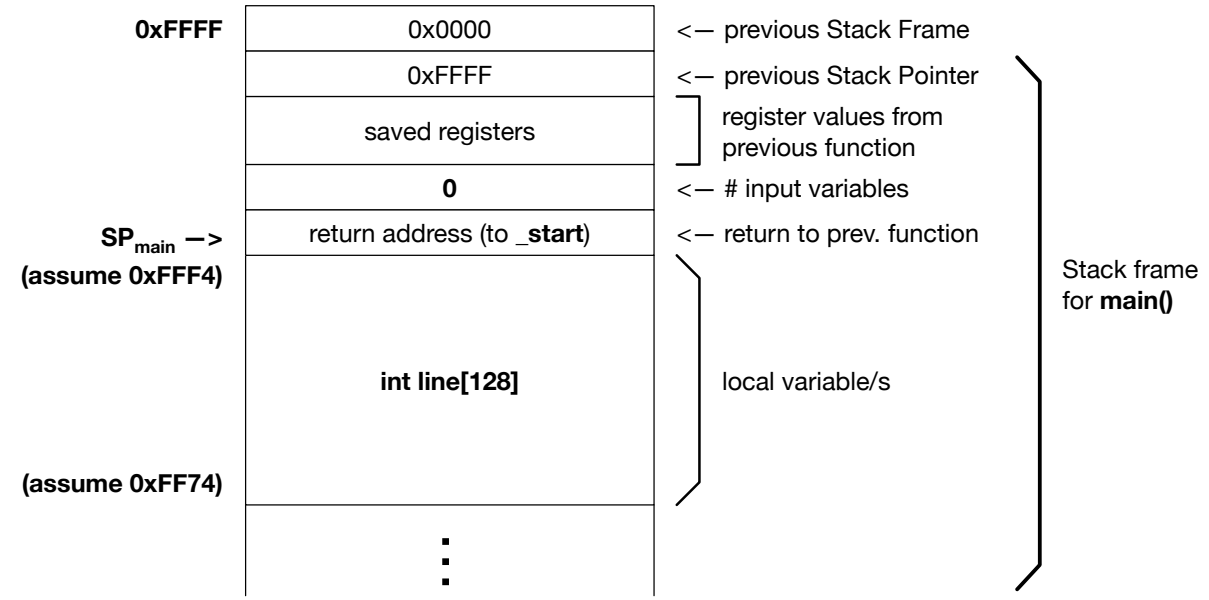
where the stack pointer is pointing. What would happen if we accidentally (or even purposefully) wrote an eight-byte value to **int z**? Remember that **int z** is a four-byte variable. Writing eight bytes to it would overwrite **int z**, but it would also overwrite the next four bytes in memory, which is the return address saved for taking us back to the calling function once the present function has finished executing. Overwriting this return address means that returning from the function will take us somewhere else.

This forms the heart of the attack. The following code provides an example:

```
void
getline(char *buf, int max)
{
    int c;
    while ((c = getchar()) != '\n') {
        *buf++ = c;
    }
}

int
main()
{
    int line[128];

    while (1) {
        getline(line, 128);
        if (process(line) == DONE) {
            return;
        }
    }
}
```

The problem here is that, even though ***getline()*** receives a value representing the maximum length of the input buffer, it never checks that value and never stops filling the buffer, even if too much data is received as input. It can therefore overflow the input buffer.

Note that the input buffer, **line[128]**, is on the stack of the ***main()*** function. Therefore, it is possible to give this application too much user information such that it will overwhelm the input buffer and overwrite the return address in the stack frame of ***main()***. For the sake of simplicity, we will assume that all data items, even characters, are 16-bit integers. This just simplifies the discussion for now. The stack frame looks something like the following:



Stack frame in the code example above

If the user input provided to this application has, for example, 129 things in it, then one can see from the illustration above that the 129th item will be located at one spot beyond the buffer. That 129th item will therefore overwrite the return address on the stack. Then, when ***main()*** returns, it will jump to an address that is potentially different from where it is *intended* to return, which is the ***_start*** routine (the initial code block that sets up the stack and calls ***main()***). Let us suppose that the stack pointer is currently pointing to location 0xFFF4 on the stack and that the variable **line[128]** can be found at location 0xFF74 on the stack. Further, let us suppose that the ***process()*** function will return ***DONE*** if the first item in the input array is a zero. Assuming all of this, we can thus imagine providing as input to this application a carefully crafted block that looks like the following:

```
0:        0x0000
1:        # machine code that implements a rudimentary shell
          # ...
          #
127:      # last (valid) entry in the input buffer
128:      0xFF75
130:      '\n'
```

What does this accomplish? The newline character at location 130 in the input block will cause the ***getline()*** function to stop collecting input data and return to ***main()***. When ***getline()*** returns, the code above will be in the following memory locations on the stack:

```
0xFF74:       0x0000
0xFF75:       # machine code that implements a rudimentary shell
              # ...
              #
0xFFF3:       # last (valid) entry in the input buffer
0xFFF4:       0xFF75
0xFFF5:       '\n'
```

The first data item will be located at address 0xFF74 on the stack; the second will be located at address 0xFF75 on the stack; etc. The 128th

element will be located at address 0xFFF3 on the stack. The next item in the input buffer, located at address 128 in the buffer, will be located at address 0xFFF4 in the stack. This item will therefore overwrite the return address of the ***main()*** function.

When the ***getline()*** function returns, ***main()*** sends the input to the ***process()*** function. The initial zero character will cause the ***process()*** function to return DONE, which will cause ***main()*** to return. This means that it will jump to the return address that has been stored in its stack frame, located at address 0xFFF4. However, as noted above, the previous return value has been overwritten with the value 0xFF75. Thus, ***main()*** will not return to the ***_start*** routine but will instead 'return' to a place it has never been before: address 0xFF75. What is located there? Machine code that implements a rudimentary shell.

This example thus turns our application into an unwitting shell program.

As the example above illustrates, and as we will show further in the *Security* chapters, it is possible to exploit errors in system software by intentionally writing too much data into a buffer located on the stack. We will also get into some solutions to the problem.

HOWEVER — this is all well and good, but the real issue is *why does this affect the entire system?* Certainly, this exploit would allow an attacker to take over a particular targeted application, but why would it go any further than that? The answer is what we discussed earlier: the decision to make some applications run with the same privilege level as the kernel. *Those* are the applications that hackers target with this attack — because, if you can take over an application that runs at the same privilege level as the kernel, you have effectively taken over the kernel as well.

Like Meltdown and Spectre, this enormous class of vulnerabilities exists only because we opened the door for it. Now is perhaps time to rethink process privileges.

1.7.5. Attacking and Protecting the Operating System

There are a handful of ways to attack the operating system directly. Several have already been described. Here are two more.

Stack Values Exposed

When your user code calls the operating system, it makes a system call. This invokes the operating system in a relatively secure manner:

- At the beginning of the call, a ***trap*** instruction is executed, which flushes the system of user instructions, jumps to the corresponding kernel handler, and increases privileges
- The system call is handled, and, depending on operating system implementation, the kernel may very well use the exact same stack as the user's code — but just below the user's stack frame
- At the end of the call, the kernel executes a ***return-from-interrupt*** instruction, which jumps back into user code and turns off privileges

The privileges are reasonably well guarded in this example, but the stack values are not. If the kernel uses the application software's stack to perform its calculations, then all of its results would be visible to

the user-level application, by simply walking down the stack a few bytes to see what is there.

Obvious solutions to this include the following:

- Have dedicated kernel stacks (can get expensive if the kernel is multi-threaded)
- Use something other than a stack interface — we will describe such a solution later

Side-Channel Exploits

A number of exploits have appeared in recent years in which user-level software observes the kernel's behavior and deduces, with alarming accuracy, precisely what the kernel was doing.

A few of the side channels that have been exploited:

- Looking at the timing of kernel responses to see whether the kernel's memory references were in cache or not, which would indicate whether or not they have been previously referenced
- Making memory references within speculatively executed code paths, to see if the targeted memory addresses are protected or not

One of the common threads is the sharing of the memory hardware with the kernel (user applications and kernel code both share the same caches, for example). One obvious solution is to segregate the kernel in hardware from the rest of the system code — we will describe such a solution later.

1.8. The Scope of the Book

This book covers a lot of ground … this is by necessity, because Cyber degrees tend to cover a wide swath of topics, often leaving less time to get into significant detail in many of the areas. Cyber degrees typically do not have the luxury of spending an entire semester on each of the topics of logic design, computer organization, computer architecture, operating systems, compilers, and computer security. Doing so would require at least six semesters' worth of courses, which takes away from other topics. Therefore, many Cyber curricula cover these topics in 1–2 semesters total.

The book provides the material for the Cyber architecture course at the Naval Academy and includes chapters on logic design, computer organization, operating systems, intro to programming-language structure, and security fundamentals. It has also been used to teach a computer-organization course, covering the topics of microprocessors, caches, pipelines, operating systems, and intro to security. Most importantly, however, the book is intended to cover security and computer design concepts together, because doing so will help ensure the security of tomorrow's systems.

The reader will also notice that instead of homework assignments the book presents programming (design) projects. The concepts in this course are learned best by doing the projects. The course website has project write-ups as well as the skeleton code.