6: Encodings: Everything is numbers

6.1: Where are we now?

In the previous chapter, we described a computer's interface--the ISA-- and specifically the computer's architecture (what various memories it has) and its instructions (what it can do to that architecture). But a computer is still supposed to be a physical machine! And so far, we've only learned (chapter 4) how physical machines can store numbers, and not how they can store these instructions. For that matter, we've talked about computers as storing much more than just numbers--a search engine stores text and a game console stores images.

In this chapter, we shall bridge this final gap by demonstrating that all these things: instructions, images, graphics, and anything else we want to store, can be stored just using numbers.

A mechanism for storing numbers that correspond to a given kind of data is called an **encoding** for that data.

6.2: Generalities about encodings

There is essentially only one hard and fast rule governing what constitutes a valid encoding: It must be **unambiguous** or **decodable**. That is, if we tell someone our encoding scheme for some set of objects (characters, instructions, whatever), and then we present them with a list of bytes that encodes some of these objects using that scheme, they should be able to use the scheme to determine without ambiguity what objects we have encoded.

As long as the encoding can actually be successfully decoded, anything else is a bonus.

For example, some encodings are more **efficient** than others in that they usually require fewer bytes to encode the same data.

As another example, some encodings can be more **error-resistant** than others. This refers to what happens if we see a sequence of bytes encoding some objects, but one or more of the bytes has been modified accidentally. An error-resistant encoding will allow us to still determine what objects were originally intended, or at least will lead us to an answer that isn't so far off what was intended. A less error-resistant encoding may have its interpretation change dramatically with even just a small change in the bytes representing it.

There are many other considerations that may apply depending on how your encoding will be deployed. Some encoding schemes may rely on actually being completely undecodable unless the person decoding them has some extra secret information (such as a password). These are usually called "encryption schemes".

Aside: Encodings in history

Nowadays, we regularly send encoded text over very large distances without a second thought. This text is usually encoded as numbers simply because most of the infrastructure we have, from hard drives to cellular towers to undersea fiber optic cables, is built solely to store and transmit numbers efficiently.

In a time when this infrastructure didn't exist, communicating data across large distances was often a massive undertaking, and being able to do this quickly could have dramatic monetary or strategic consequences. Text was still encoded for communication, not as numbers (since the number-moving infrastructure wasn't in place yet) but in various other ways that were adapted to the infrastructure that did exist at the time.

One such encoding was called "semaphore", which involved a signaler standing someplace visible to the recipient and holding out flags in various configurations representing letters of the alphabet. If the recipient were too far away to see the sender, there would have to be intermediate signalers to relay the original message. Sometimes, there were many relayers in a very long line. In 18th century France, relay towers were built with larger, more visible mechanical arms. They were constructed along lines that spanned the whole width of the country, such that a commander in Paris could receive news from Strasbourg in the far north-east in just six minutes. Napoleon Bonaparte even had such a line constructed from Paris all the way to Venice.

Another encoding that surfaced when electric lines started to criss-cross whole countries (but did not yet carry digital information) was Morse code. One could connect an electrical line to a buzzer, and as the sender on the other end connected and disconnected the line with a button, the receiver could listen to the buzzer and denote the short buzzes as dots, the long buzzes as dashes, and longer pauses between buzzes as separating the different letters. Morse code was a system for corresponding sequences of dots and dashes to letters. In this encoding, "-..-" (heard as "beeeeeeep beep beep beeeeeeep") would have represented "X", whereas "." would have represented "E". This represented an early understanding of efficiency in encodings: "E" is a much more common letter than X, so it is more efficient to have it correspond to a simpler sequence.

6.2.1: Enumeration as encoding

Before we get into any specific encodings, we present the most basic strategy of all for encoding: Enumeration. To illustrate this strategy, we return to our waffle bakery example:

We're designing a computer program for the bakery. We want to represent the different waffle offerings they have using numbers. So we make a list of all the waffle offerings (starting from 0 like good computer programmers):

```
0. Donut waffle
1. Plain waffle
2. Blueberry waffle
3. Belgian waffle
4. Whole-grain waffle
5. Chocowaffle
6. Catnip waffle
...
```

And then we realize that this numbered list actually is an encoding already--Blueberry waffles correspond to the number 2, chocolate to the number 5, and so on! So when the user selects "Catnip waffle" on the website, the website the number 6 to the server and the server takes recipe number 6 and sends it to the kitchen for preparation, which is of course the recipe for a catnip waffle.

This encoding is of course decodable: If we give someone the sequence of bytes 4, 1, 3 then they can easily decode this as "A whole-grain, a plain, and a Belgian". But what happens if the chef gets really creative and comes up with 300 recipes?

```
0. Donut waffle
1. Plain waffle
2. Blueberry waffle
3. Belgian waffle
4. Whole-grain waffle
5. Chocowaffle
6. Catnip waffle
...
259. Stroopwafel
260. Maple bacon waffle
261. Sunny-side up waffle
...
```

```
297. Upside-down waffle
298. Inside-out waffle
299. Waffle waffle
```

Now suddenly some of the numbers are too big for a byte (remember bytes can only store 8 bits, or numbers 0-255). "No problem", we think. "we'll just use two bytes for the bigger ones using concatenation". Since 259 = 1*256 + 3 = 1:3, we can store the number 259 using the two bytes 1, 3. But now we have a decodability problem: What if we get the sequence of bytes 4, 1, 3. Does this mean "Whole-grain, plain, and Belgian" or "Whole-grain and Stroopwafel"?

So if we're going to use two bytes to encode some of the waffles, we'd have to use two bytes to encode all of them. Thus an order of "Plain, Stroopwafel, and Donut waffle" would be encoded as 0, 1, 1, 3, 0, 0 (since Plain = 1 = 0:1, Stroopwafel = 259 = 1:3, and Donut waffle = 0 = 0:0).

So enumeration encodings can essentially always be used to encode any finite list of objects as long as sufficient care is given to ensure decodeability.

6.3: Encoding of instructions

Our first example of an encoding is ISA instructions. Recall that the way the processor is set up, there is a memory for storing instructions--the program memory--which has width 16, meaning it can only store words (numbers 0-65535). Somehow, we can store some numbers in this memory to cause the program

```
ldi r30,89
andi r30,6
```

to be executed.

For the sake of example, let us look at one hypothetical way that one could encode instructions:

6.3.1: An enumeration-based instruction encoding

One very naive way to encode instructions is to simply enumerate them. In the previous chapter, we provided a list of all possible instruction types--ldi, add, rcall, etc. However each instruction could only take finitely many possible operands, so we can in fact enumerate all the possible instructions one could ever write in a (very long) finite list. For instance, all the possible incarnations of ldi instructions are:

```
ldi r16,0
ldi r16,1
...
ldi r16,255
...
ldi r31,255
```

There are 256 instructions of the form ldi r16,[something], and 16 different registers we can load a value into (recall ldi doesn't let you load into registers 0-15), for a total of 4096 different instructions. For an encoding, we could just make them correspond to the numbers 0-4095 respectively. Thus ldi r16,0 will be encoded with the number 0, ldi r16,1 with the number 1, ldi r17,0 with the number 256, and ldi r30,89 with the number 3673, etc.

Then we have the andi instructions

```
andi r0,0
...
andi r0,255
andi r1,0
...
andi r1,255
......
andi r31,255
```

This is another 8192 instructions. We've already assigned the numbers 0-4095 to correspond to the ldi instructions, so we may as well take the next 8192 numbers--that is, 4096-12287--and make these correspond to the andi instructions in the above order. In the above list of andi instructions, andi r30,6 will be the 7687th andi instruction, so it will be assigned the number 11783 under this encoding scheme.

Thus, if we used the naive encoding above, then our program would be stored as the two numbers 3673 and 11783. Or, in binary:

```
0000111001011001
0010111000000111
```

as 3673 is the encoded version of ldi r30,89 and 11783 is the encoding of andi r30,6.

For all we know so far, this encoding might be as good as any other--for instance, we chose to start with the ldi and andi instructions, but maybe we could instead encode the ldi instructions using the first 4096 numbers, and then encode the mov instructions using the next 1024 numbers (1024 being the number of different possible mov instructions).

But to see if we can be more systematic about it, let us see what this encoding actually means on the level of the binary representations. Under this suggested scheme, we'll have encodings:

Instruction	Decimal	16-bit binary
ldi r16,0	0	0000000000000000
ldi r16,1	1	0000000000000001
ldi r16,60	60	0000000000111100
ldi r16,145	145	0000000010010001
ldi r16,255	255	0000000011111111
ldi r17,0	256	0000000100000000
ldi r18,0	512	0000001000000000
ldi r19,0	768	0000001100000000
ldi r26,0	2560	0000101000000000
ldi r31,0	3840	0000111100000000
ldi r31,5	3845	0000111100000101
ldi r31,60	3900	0000111100111100
ldi r31,255	4095	0000111111111111

A couple of things we notice from these examples: out of our 16 bits, the low 8 bits are just the binary representation of the value that should be loaded. For instance, if we want to load the value 60 into some register, I take 60 as an 8-bit binary number: 00111100, and place that into the lower 8 bits of the instruction. Observe that ldi r16,60 was

```
0000000000111100
```

whereas ldi r31,60 was

```
0000111100111100
```

Then the next-lowest 4 bits tell us which register we're loading from: To store the value 145 (binary 10010001) into register 16, we do

```
0000000010010001
```

whereas to store 145 into r26, we do

```
0000101010010001
```

and into r31 would be

```
0000111110010001
```

Observe that these four bits aren't literally the binary representation of the register number to be loaded:

To load to register number	(in binary)	Set bits 9-13 to:
16	10000	0000
26	11010	1010
31	11111	1111

Since the bits that select which register we load into only have to select between 16 different possibilities (registers 16-31) we only need 4 bits to represent this choice. And if those four bits are, say, 1010, we see that we can get the actual register number being selected by just throwing a 1 on the front of it, giving 11010, i.e. 26.

So fundamentally, to specify an ldi instruction, we need to use 4 bits for the register selection, and 8 bits for the immediate value that we wish to load. Our naive encoding said that for ldi, we use the lowest 8 bits for the immediate value and the next 4 bits for the register select. We shall denote this encoding in the following concise notation:

```
ldi: 0000RRRRIIIIIIII loads binary value IIIIIIII into register 1RRRR
```

What this means is that the bits labeled I comprise the binary value to be loaded, and the bits labeled R comprise the last four bits of the register select, with the first bit of the register select being always 1. For example, if we get the instruction 0000101000011110, we can decode it thus:

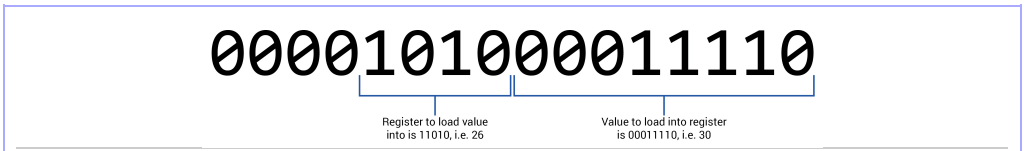


Figure 6.3.1.1: Decoding 0000101000011110 as ldi r26, 30

6.3.2: The AVR ISA encoding

But now that we understand what's really required to choose a representation for ldi instructions--we need to reserve one 4-bit blob and one 8-bit blob in the instruction for the operands--we can do whatever we like. For instance, we could swap the order in which we did the above:

```
ldi: 0000IIIIIIIIRRRR loads binary value IIIIIIII into register 1RRRR
```

And this might be an equally good choice. So rather than choosing an encoding by trying to enumerate all the instructions and assign them numbers sequentially, we can be more systematic by looking at all the types of instructions (mov, ldi, andi, etc.) and boil them down to exactly what information they require. For example, we just saw that ldi fundamentally requires only one 4-bit number and one 8-bit number.

By contrast mov requires us to specify two registers--that is, two numbers 0-31. And in binary, it takes 5 bits to represent a number 0-31. So the basic information built into a mov instruction is two 5-bit numbers. So we could encode mov instructions as:

```
mov: 000000RRRRRSSSSS moves value of register SSSSS into register RRRRR
```

Thus the instruction

```
0000000011011110
```

would have RRRRR being 00110 and SSSSS being 11110. In particular, this would put whatever was stored in register 30 into register 6.

But not so fast! Back when we were using the enumeration encoding, we were guaranteed decodability, since every instruction was two bytes long. But now that we've ventured out on our own,

we need to be careful. We intended this word to represent a mov, but as we've encoded it, this could also be interpreted as the instruction ldi r16,222, (as 1RRRR is 10000 (decimal: 16) and IIIIIIII is 11011110 (decimal: 222)).

To resolve this particular ambiguity, we can take advantage of the bits that are not storing actual information used by the instruction--those zeroes out in front. In our ldi encoding, these are the first 4 bits, whereas in mov, this is the first 6. So if we instead use:

```
ldi: 0000IIIIIIIIRRRR loads binary value IIIIIIII into register 1RRRR
mov: 000100RRRRRSSSSS moves value of register SSSSS into register RRRRR
```

Now we can always tell which instruction was intended by looking at the first four bits. If we see 0001, it was a mov, and if we see 0000, it was an ldi.

Now we have all the background necessary to understand the encoding used by actual AVR chips for the AVR ISA. We'll start by listing the types of instruction and what information they fundamentally require:

Instruction	Required information
ldi	4-bit register, 8-bit immediate
mov	5-bit register, 5-bit register
add	5-bit register, 5-bit register
sub	5-bit register, 5-bit register
inc	5-bit register
dec	5-bit register
and	5-bit register, 5-bit register
andi	4-bit register, 8-bit immediate
or	5-bit register, 5-bit register
ori	4-bit register, 8-bit immediate
com	5-bit register
rjmp	12-bit immediate
breq	7-bit immediate
brne	7-bit immediate
brsh	7-bit immediate
brlo	7-bit immediate
cp	5-bit register, 5-bit register
cpi	4-bit register, 8-bit immediate

We come out with the groupings:

Required information	Instructions requiring this information
4-bit register, 8-bit immediate	ldi, andi, ori, cpi
5-bit register, 5-bit register	mov, add, sub, and, or, cp
5-bit register	inc, dec, com
12-bit immediate	rjmp
7-bit immediate	breq, brne, brsh, brlo

We just need to take each of these types and specify which bits of the encoding are used for which operands in each of them. For instance, the AVR encoding for the first type of instruction is defined by the following arrangement:

```
CCCCIIIIRRRRIIII
```

This means we use the lowest 4 bits for the low 4 bits of the immediate operand, the next 4 bits for the register, and then the next 4 bits for the high 4 bits of the immediate. This leaves us with 4 additional bits which we have marked as C.

Note that we have four different instructions that take a 4-bit register and an 8-bit immediate. The C bits are used to distinguish between these. For instance, for any ldi instruction, the C bits are always set to 1110. This set of bits, which specifies not the operand, but rather which type of instruction is being run, is called the **opcode**.

Thus if we receive the instruction

```
1110101000101110
```

then we break it up:

```
C: 1110101000101110
I: 1110101000101110
R: 1110101000101110
```

So the opcode is 1110, which tells us we're doing an ldi instruction. The immediate is 10101110 (decimal: 174), and the register is 0010 (decimal: 2). So this instruction should be ldi r2,174. Note that in this format, the largest possible immediate we can load is by setting all the I bits to be 1, as in:

```
1110111100001111
```

which corresponds to ldi r16,255.

We now summarize the bit-level layouts of the other groups of instructions that are used in the AVR ISA encoding:

Instruction type	Format
4-bit register, 8-bit immediate	CCCCIIIIRRRRIIII
5-bit register, 5-bit register	CCCCCCSRRRRRSSSS
5-bit register	CCCCCCCRRRRRCCCC
12-bit immediate	CCCCIIIIIIIIIIII
7-bit immediate	CCCCCCIIIIIIICCC

In each grouping, we need also to say what the opcodes for the various instructions within that grouping are::

Instruction	Opcode
ldi	1110
mov	001011
add	000011
sub	000110
inc	10010100011
dec	10010101010
and	001000
andi	0111
or	001010
ori	0110
com	10010100000
rjmp	1100
breq	111100001

brne	111101001
brsh	111101000
brlo	111100000
cp	000101
cpi	0011

Thus, to encode a mov instruction--say mov r30,r21--we use the opcode 001011, the source register r21, or 10101, and the target register 30, or 11100. Using the recipe for this group of instructions:

```
CCCCCCSRRRRRSSSS
```

We arrive at the encoding

```
0010111111000101
```

where the components of the instruction are found as:

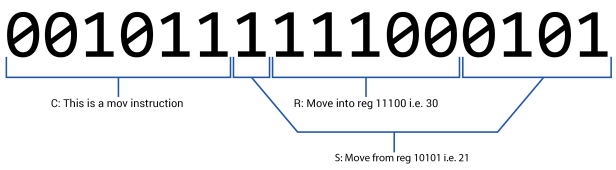


Figure 6.3.2.1: The components of the mov instruction

Similarly, to encode the instruction breq 56, we use the opcode 111100001 and immediate value 0111000, and put these together according to the recipe for instructions involving a 7-bit immediate value, which was:

```
CCCCCCIIIIIIICCC
```

So we arrive at the encoding

```
1111000111000001
```

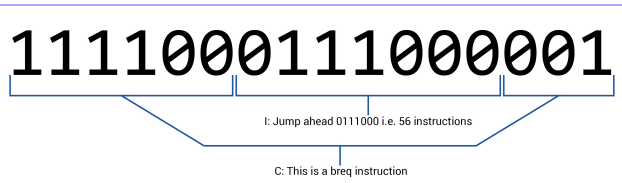


Figure 6.3.2.2: Breakdown of the breq instruction

So this is great from a programmer's perspective--we now know how to feed numbers into the processor (which is all the processor understands) that will correspond to whatever instructions we want. But because this is no longer an enumeration encoding, we must give some thought to decodability.

For instance, suppose the computer receives from the program memory the number

```
0010111011110111
```

Can it actually tell unambiguously which instruction this is?

To make this apparent, we list the opcodes for our instruction types once more, but this time, with we'll replace the bits that specify the operands with "?", since the instruction should be interpretable regardless of the values of these bits.

Instruction	Opcode
add	000011??????????
cp	000101??????????
sub	000110??????????
and	001000??????????
or	001010??????????
mov	001011??????????
cpi	0011????????????
ori	0110????????????
andi	0111????????????
dec	1001010?????1010
com	1001010?????0000
inc	1001010?????0011
rjmp	1100????????????
ldi	1110????????????
brlo	111100???????000
breq	111100???????001
brsh	111101???????000
brne	111101???????001

So if we take our input of

```
0010111011110111
```

to this table, we check it against each of these patterns and discover that the only one that matches it is mov. So then we can break it up as before:

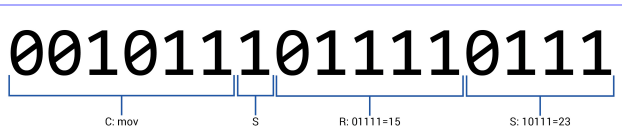


Figure 6.3.2.3: Decoding the instruction as a mov

So this is the instruction mov r15,r23.

Similarly, if we get the instruction

```
1001010000000011
```

then we go to the table and see that only inc, dec, and com can possibly start with 1001010. But these are distinguished by the three bits they end with. And only inc ends with 011. So this is an inc instruction, and the R bits are all 0, so this must be inc r0.

The fact that this encoding can be decoded is no longer ensured by enumeration, but simply by the cleverness of the AVR ISA designers. Why this particular encoding was chosen (for instance, why the S bits in a mov instruction are split up like that) may have to do with what was most convenient for the people implementing the processor, or with the way programmers worked at the time the encoding was designed. So while this choice of encoding feels a little arbitrary or needlessly complicated, its original designers certainly had more constraints to satisfy than were presented here.

6.4: Search engine: The encoding of text

Now we return to our search engine case-study in an attempt to understand it at a lower level than

before. The basic operation that happens in searching is comparing strings. But really, to compare two strings, we can just compare each letter of the strings one after the other. So what is this operation of comparing letters? After all, we now know all the operations that our computer can actually do, and none of them even mention letters. What gives?

As with instructions, and really everything else, the computer represents letters with bits. That is, when you enter the string "banana". The computer is not dealing with the string itself, but rather with a sequence of numbers that encode the string. In this case, the sequence is:

```
98,97,110,97,110,97
```

Depending on the need, there are different ways of encoding letters using numbers, two of which we'll introduce now: ASCII, and UTF-8.

6.4.1: Text--ASCII

In the early days of computing, computers were used mostly in the English speaking world, so the characters people wanted to be able to use was limited to a small set of about 98 different symbols--the 56 coming from 0-9, a-z, and A-Z, and punctuation characters.

Of course, since a computer can only store numbers, there had to be a universal encoding for text that all computers could use. What came out of this was the ASCII character encoding, which (basically) an enumeration of the relevant symbols:

Number	Character
9	[tab]
10	[newline]
32	[space]
33	!
34	"
35	#
36	$
37	%
38	&
39	'
40	(
41	)
42	*
43	+
44	,
45	-
46	.
47	/
48	0
49	1
50	2
51	3
52	4
53	5
54	6
55	7
56	8
57	9
58	:
59	;
60	<
61	=
62	>
63	?
64	@
65	A
66	B
67	C
68	D
69	E
70	F
71	G
72	H
73	I
74	J
75	K
76	L
77	M
78	N
79	O
80	P
81	Q
82	R
83	S
84	T
85	U
86	V

```
87      W
88      X
89      Y
90      Z
91      [
92      \
93      ]
94      ^
95
96      `
97      a
98      b
99      c
100     d
101     e
102     f
103     g
104     h
105     i
106     j
107     k
108     l
109     m
110     n
111     o
112     p
113     q
114     r
115     s
116     t
117     u
118     v
119     w
120     x
121     y
122     z
123     {
124     |
125     }
126     ~
```

This encoding despite being designed for a more or less bygone era is still used extremely widely. For instance, if you are reading this book online, most of the text you are reading was encoded in ASCII to be transmitted to your computer. The reason is mostly inertia: Software written back when there was only ASCII would have to be rewritten if we switched to a more modern encoding.

As an enumeration encoding, ASCII is definitely decodable. In fact, every character it encodes can be represented by a single byte, so if we see the list of bytes

```
72,101,108,108,111,32,87,111,114,108,100,33
```

we can reference the table and readily decode this as the string

```
Hello world!
```

However, we still definitely want improvements on ASCII--accented vowels, letters from non-Roman alphabets, and the like. This brings up another feature of encodings: backwards-compatibility: Suppose we have a nice fancy encoding that encodes Chinese characters and the Cyrillic alphabet and more. But suppose also that for a particular application, all we're doing is encoding English text (still using this fancy encoding). If for this task (for which ASCII would have sufficed) the new encoding generates the same output as ASCII would, then older software can still accept input in the new encoding as long as that input is just English. In this case we say that the new encoding is **backwards-compatible**.

Aside: What are the missing characters?

ASCII is kind of an enumeration encoding, but the enumeration we've given here skips some numbers. What gives? The actual ASCII enumeration doesn't skip any numbers; it's just that some of the things they chose to encode are rarely useful anymore, and so we skipped them for presentation purposes here. As a historical curiosity, though, we might wonder what were these other characters.

Many old computers were just text terminals: All you could do was input text (no mouse) and all it could do was output text. The specific way it outputted text was (roughly) the following: There was always a cursor on the screen. When the user pressed the "A" key, this was translated to the ASCII number for "A", i.e. 65 and sent to the screen. Since "A" is a normal character, the screen simply placed a picture of an "A" where the cursor was and moved the cursor one space forward.

But what happened when the user pressed "Backspace", say? This was represented in fact by with ASCII number 8, and when the screen got the number 8, it moved the cursor backward by one space. When the user pressed "Enter", first the number 10 (newline) was sent to the screen causing the cursor to move down a line. But we are used to the cursor also going to the beginning of the next line when we press enter. To accomplish this, the enter key actually sent two numbers: 10 (for "go to the next line") and 13 (for "go to the beginning of the line").

Likewise there were other numbers for moving the cursor around in various ways, numbers corresponding to the shift key, sequences of ASCII characters that could change the color of the characters being printed, and even a character, encoded with the number 7 (not to be confused with the character "7", which is encoded with the number 55) that would cause the computer to emit a "beep" when it was sent to the screen! In fact, there are still terminal programs (such as those on Linux or OS X) that will beep when you print ASCII character number 7, such as with a command like:

```
printf '\x7'
```

Sadly, many of these terminal programs have this feature turned off by default.

6.4.2: Text--Unicode

ASCII is convenient when we want to be able to use a single byte to represent every character, but there are only 256 different bytes and there are definitely languages that simply have more than 256 characters, or characters not covered by the ASCII encoding. ASCII is not enough even if you just want to store proper French text, and an encoding that uses one byte per character will never work if you want to be able to store text in Chinese.

Thus a more comprehensive character encoding is needed, and several standards exist as attempts to fill the gap. By far the most popular as of this writing is one called UTF-8 ("Universal character set Transition Format--8-bit").

As with ASCII, UTF-8 assigns to every character a unique number, called that character's **codepoint**. For example, the character pictured here

Figure 6.4.2.1: Unicode codepoint 16706: 格

has codepoint 16706. But then how does the computer actually store this character as a sequence of bytes? If we simply convert 16706 to binary, getting 100000101000010, we could store this as the bytes 01000001 01000010. But when when the computer goes to figure out what this means, how does it distinguish between the two-byte character representing codepoint 16706 and two one-byte characters representing codepoints 65 and 66 respectively (in ASCII, "A" and "B")?

This is where a few possible encoding schemes take different ideas. One called **UTF-32** simply declares that every character must take 4 bytes. In UTF-32, then, this character becomes 00000000 00000000 01000001 01000010. However this means that every single character is now 4 bytes--even simple English text that ASCII could encode in 1 byte per character. There are other issues with this encoding also that we will not detail here.

At this point, UTF-8 steps up as a reasonable solution: Any byte that matches one in the ASCII table will have that same ASCII interpretation when interpreted under the UTF-8 scheme, but characters that require more bytes to represent them will also be possible. The trick that allows it to accomplish

this is to notice that in ASCII, every byte that is defined in the table has the most significant bit always 0.

The short description of UTF-8 is this: Take the first byte of a character. If the MSB is 0, then that byte is interpreted according to the ASCII table. If the MSB is 1, though, then count off how many 1s there are until the first 0 of the byte. That is how many bytes are involved with this character. So if the first byte is 11100101, then because this byte starts with three 1s, it indicates that this byte and the next two bytes together represent a single character. These next bytes that are part of the same character must further begin with the bits 10. So some of the bits in a UTF-8-encoded character (the "1110" in the first byte, and the "10"s in the following bytes) are for determining which bytes are involved in that character. The bits that remain are combined to determine the codepoint itself.

For a simple example, a single character in UTF-8 would be:

```
1110 0010
10 011100
10 001000
```

Here we have bolded the bits that help determine what bytes are involved: The 1110 starting the first byte tells us that the character is comprised of three bytes in total. The 10s starting off the next bytes simply indicate that those bytes are not the start of a new character but are part of a character that starts at an earlier byte.

The bits that remain, then, are 0010011100001000, representing the number 9992. If we look up this codepoint in a Unicode table, we get the character ✈

For a multi-character example, then, say we have the sequence of bytes:

```
00100001
11000011
10110111
11100010
10011000
10111010
```

Then the first byte 00100001 starts with a 0, so is a full character on its own (in fact, the character "!"), and thus the next byte must be the start of the next character.

The second byte starts with 110, which indicates that this next character occupies two bytes--this one and the byte after it. And indeed the byte after it--10110111--starts with 10, so whatever this character is, it is represented by the bytes

```
11000011
10110111
```

Since the 110 starting the first byte and the 10 starting the second are just there to indicate that these bytes together form one character, the remaining bytes--00011110111--together specify which character it is. To find out which character, we can look these up in the UTF-8 table. Since in decimal, this binary number is 247, we look at the 247th entry and find that this is the division sign: ÷.

The next byte is 11100010, suggesting that the following two bytes are also a part of this character. So the character is comprised of the bytes:

```
11100010
10011000
10111010
```

Of these, some of the bytes are again being used to pick out this blob of three bytes as one character:

```
1110 0010
10 011000
10 111010
```

and the rest--0010011000111010--are specifying which character it is. In decimal this is 9786, so we can look up this entry in the UTF-8 tables to discover that this character is a smiley face.

6.4.3: Which encoding to use?

So with all of these encodings and many that we haven't mentioned, which is the best? As always, "best" can only be defined relative to a stated purpose. In this case, we need to think about how we plan to use the characters that we are encoding. For example, if we are programming a text editor, we may be storing a large number of bytes representing encoded characters, and may wish at some point to identify how many characters are actually present.

With ASCII this is easy: Each character is one byte, so the number of bytes in the text is the number of characters. With UTF-32, this is also easy: Every character is 4 bytes, so the number of characters is the number of bytes divided by 4. With UTF-8, this is not so easy--if I have 100 bytes, it might be 100 one-byte characters, or it might be 50 two-byte characters, or possibly some mix of several different length characters. To find out the true answer, we have essentially no choice but to walk through each character and count them.

On the other hand, suppose we're making a web browser. In this case, we may find ourselves having to start decoding a string of characters somewhere in the middle--maybe for a text chat application. In this case, we may be missing some of the bytes from the beginning of the representation. In ASCII this is fine--the result will be interpreted as if we just left off some characters.

In UTF-32, however, this can go very very poorly: Imagine we had the string "Hello", represented in UTF-32 as:

```
00000000 00000000 00000000 01001000
00000000 00000000 00000000 01100101
00000000 00000000 00000000 01101010
00000000 00000000 00000000 01101010
00000000 00000000 00000000 01101111
```

And suppose the first two bytes were removed, leaving us with

```
00000000 01001000 00000000 00000000
00000000 01100101 00000000 00000000
00000000 01101010 00000000 00000000
00000000 01101010 00000000 00000000
00000000 01101111
```

Now, if we try to interpret this as UTF-32, we get dramatically different results, as these 4-byte blobs represent massive codepoints with unknown characters.

By contrast, if you chop off the first two bytes of a UTF-8 representation, say:

```
01100101
11110000
10011111
10010000
10110001
11110000
10011111
10011011
10100111
```

leaving us only with

```
10011111
10010000
10110001
11110000
10011111
10011011
10100111
```

Then we see the first three bytes all start with "10". 10 can never be the starting bits of a character, so we know immediately that these are missing something, and can discard them and start interpreting from the next valid character start: 11110000. In this case, we lose some characters off the beginning like in the case of ASCII, but we do not lose all meaning entirely.

6.5: Game console: The encoding of graphics

We've just seen how text files can indeed be stored in a computer's memory (even though that memory can only store bytes) by means of various character encodings. We now turn to a more complicated sort of data, which we'll need if we ever want a hope of making games beyond the text-

based format: Images.

Just as there are several ways of storing plain text, there are many ways of storing image data, each with its own advantages and disadvantages. Before we get into the ways of storing an entire image, we'll discuss the most basic component of all images: Just as strings were comprised of characters, images are (usually) comprised of pixels.

A computer screen is a grid of lights each of which can be set to any color you want individually. Each one of these lights is called a pixel. To display an image on screen, then, is nothing more than to take the pixels in the location where you want to display the image and set them all to be the appropriate colors so that the desired image appears.

6.5.1: Pixels--RGB

So the first problem to solve is how do we specify a color using just numbers? It turns out that the physical design of an individual pixel sort of gives us a way of doing this already:

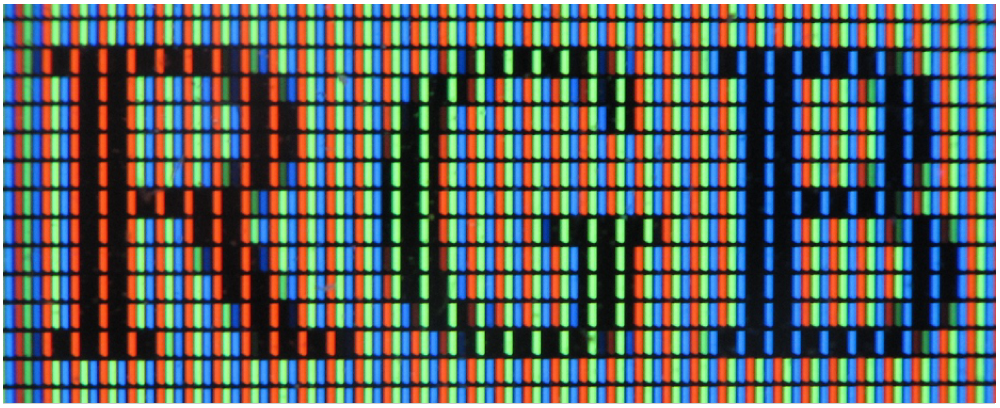


Figure 6.5.1.1:

As you can see, each pixel is not a single light at all, but in fact three lights--one red, one green, and one blue. Combinations of light of these colors can be used to create any other color:

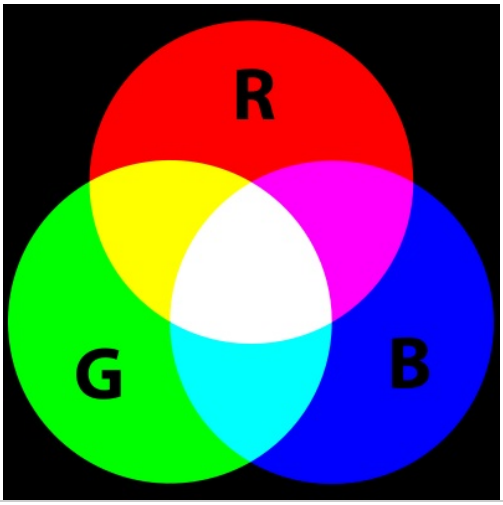


Figure 6.5.1.2: Taken from Wikipedia

and by specifying the intensity of each of the three colors, we can specify what color we want the pixel to appear as. One common way of doing this is by specifying each intensity with a single byte, so the full color of a single pixel requires three bytes to specify.

Some common colors with their corresponding RGB values are given in the table below:

Color name	Red value	Green value	Blue value	Color	Magnified pixel

White	255	255	255		
Red	255	0	0		
Green	0	255	0		
Blue	0	0	255		
Grey	204	204	204		
Magenta	255	0	255		
Puce	114	47	55		

One convenient thing about this representation is every color is represented by three bytes. Since a byte is 8 binary digits, therefore two hexadecimal digits, we can represent a color by six hexadecimal digits. This is a very common idiom in website programming, to say that e.g. puce is the color "722f37", because 0x72 = 114, 0x2f = 47, and 0x37 = 55. We shall actually adopt this idiom for the rest of the section, so in the next section when we give lists of colors or list of bytes, they will all be in hexadecimal.

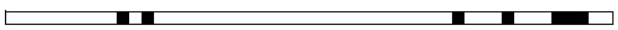
6.5.2: Images--BMP

Now that we know how to encode the color of an individual pixel, we consider what information is needed to turn a sequence of these into an image. For instance, if we just specify the list of colors:

```
ffffff,ffffff,ffffff,ffffff,ffffff,ffffff,ffffff,ffffff,
ffffff,ff0000,ffffff,ff0000,ffffff,ffffff,ffffff,ffffff,
ffffff,ffffff,ffffff,ffffff,ffffff,ffffff,ffffff,ffffff,
ffffff,ffffff,ffffff,ffffff,ffffff,ffffff,ffffff,ffffff,
ffffff,ffffff,ffffff,ffffff,000000,ffffff,ffffff,ffffff,
000000,ffffff,ffffff,ffffff,000000,000000,000000,ffffff,
ffffff
```

We don't know which of the following images this list represents:

49x1:



7x7:

Figure 6.5.2.1:

So we need some additional information to specify the height and width, among other things.

The BMP file format is one of the simplest ways of arranging pixel data so that it can be understood for display unambiguously.

Here is an example of a BMP image (here it has been blown up 10x in height and width; click it to get a link to the original image file):

And here is the actual sequence of bytes that the computer is storing in this file:

```
42 4d de 00 00 00 00 00 00 36 00 00 00 28 00
00 00 07 00 00 00 07 00 00 00 01 00 18 00 00 00
00 00 a8 00 00 00 13 0b 00 00 13 0b 00 00 00 00
00 00 00 00 00 00 ff ff ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff ff ff 00 00 00 ff ff ff
ff ff ff ff 00 00 00 00 00 00 00 00 00 ff ff ff
ff ff ff 00 00 00 ff ff ff 00 00 00 ff ff ff ff
```

```
ff ff ff ff ff 00 00 00 ff ff ff 00 00 00 ff ff
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff 00 00 00 ff ff ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff ff ff ff 00 00 00 ff ff
ff ff ff ff 00 00 00 ff ff ff 00 00 00 ff ff ff
ff ff ff 00 00 00 ff ff ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff ff ff ff 00 00 00
```

The first two bytes are the ASCII values for the characters B and M, respectively. This allows the computer to look at the file and determine that it is a BMP file even if the filename weren't something so informative as "smile.bmp".

The 11th byte tells us where the actual pixel data is stored. Since in this case it is 0x36 (decimal: 54) this means that the bytes that actually represent pixel colors start at byte number 54 onward. The rest of the bytes up to byte number 54 are there to describe the image--how many pixels tall and wide it is, etc. Incidentally, this "data about the data" is called **metadata**. Bytes 19-23 stores the width of the image--in this case 0x7, or, in decimal, 7. The next 4 bytes store the height--again 7, so the image is as advertised, 7x7.

The bytes we've skipped give some further technical information about the image that we won't go into here, but then starting at byte 54, we start to get the actual image data, which we conveniently group into 3-byte blocks for the individual pixel colors and arrange into 7 columns:

```
ffffff ffffff ffffff ffffff ffffff ffffff ffffff
000000 ffffff ffffff 000000 000000 000000 ffffff
ffffff 000000 ffffff 000000 ffffff ffffff ffffff
000000 ffffff 000000 ffffff ffffff ffffff ffffff
ffffff ffffff ffffff 000000 ffffff ffffff ffffff
ffffff ffffff ffffff ffffff 000000 ffffff ffffff
000000 ffffff 000000 ffffff ffffff 000000 ffffff
ffffff ffffff ffffff ffffff ffffff ffffff 000000
```

Oddly, even after squinting at this representation, it doesn't look anything like the smiley face we'd expect. In fact, this arrangement actually had 8 rows, so something really is amiss. The trick is that the BMP encoding also requires that each row of pixels must store a multiple of four pixels. If the image width isn't a multiple of four, extra 0s will be added to fill out each row.

Since our image is 7 pixels wide, the BMP image will include one extra pixel full of 0s at the end of each row to round the row size off to the nearest multiple of 4, namely 8. Rearranging these pixels then to 7 rows of 8, we see:

```
ffffff ffffff ffffff ffffff ffffff ffffff ffffff 000000
ffffff ffffff 000000 000000 000000 ffffff ffffff 000000
ffffff 000000 ffffff ffffff ffffff 000000 ffffff 000000
ffffff ffffff ffffff ffffff ffffff ffffff ffffff 000000
ffffff ffffff ffffff ffffff ffffff ffffff ffffff 000000
ffffff ffffff 000000 ffffff 000000 ffffff ffffff 000000
ffffff ffffff ffffff ffffff ffffff ffffff ffffff 000000
```

Now, squinting enough, we actually do see a smiley! But upside down? This comes to the last oddity (that we'll discuss, anyway) about the BMP encoding, namely that the first pixel stored is the bottom right pixel, and then the pixels proceed right-to-left, bottom-to-top. (So in fact, if we had selected a less symmetric image, it would not only be upside-down, but would also be mirror-imaged.)

BMP is only one encoding for images. It is simple to understand (which is why we included it) but it is not particularly efficient and thus sees little use in the wild these days, with preference given to encodings such as GIF, PNG, JPEG, and WEBP. To illustrate the basic idea behind the inefficiency of BMP, we could describe the image above completely and decodably by the much shorter sequence of bytes:

```
7x7 image
The pixels in order are (w=white, b=black):
9w, 1b, 1w, 1b, 17w, 1b, 3w, 1b, 3w, 3b, 9w
```

This is a simple example of what is known as "compression"--expressing the same data more efficiently. Various sophisticated compression algorithms form the basis for the other image encodings we mentioned, but we'll not go into further detail here.

6.6: Mobile phone: The encoding of sensor data

Summary: In the foregoing sections, the challenge of encoding data was always how to assign numbers to the data in a usable way. In this example, there is an obvious way to encode the data: We're sending three numbers for acceleration along the three axes, so just represent these numbers as binary and send them along! However, if the wires along which we are sending these wires are considered unreliable, then using the obvious encoding may end up with our receiving misleading data. To mitigate this, we give our first (and, for this book, only) example of an encoding that not only has to unambiguously communicate the desired information, but that must account for other considerations beyond ease of decoding.

A cell phone presents many further encoding challenges. Here, we'll focus on for now are how to encode the acceleration data collected by the accelerometer for transmission to the processor.

On the one hand, we can say that the accelerometer just outputs three bytes--one representing the amount of acceleration for acceleration in each of the cardinal directions.

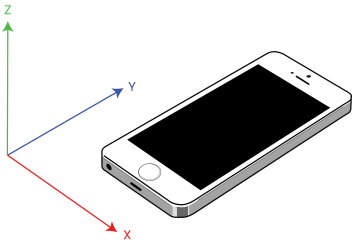


Figure 6.6.1:

However, sending a byte to the processor could mean that there are 8 wires going between the accelerometer and the processor. But since the device is also a phone, it's got an antenna transmitting as well, which means that all of these wires are very close to some decently powerful radio transmissions. And radio waves near a wire can cause current to flow through that wire--this is, after all, how an antenna works in the first place. So we run the risk of at least one of the signals on one of the wires being changed to something other than what the accelerometer device intended to send.

So, for example, if we were trying to send the x-acceleration as the number 23, i.e. as 00010111 on the 8 wires, the radio interference could flip one of these bits in transit and the processor could receive the value 01010111, or 87, instead.

Thus the simple encoding of: "Turn the number into binary and then send out the bits on wires" may be inadequate to ensure successful communication.

6.6.1: Sensor data with repetition for error-detection

One simple (if somewhat heavy-handed) change would be to have instead of 8 wires, have three sets of 8 wires, and send the same number out on all three. The calculation is that while there is a chance of one bit being flipped, there is a much lower chance of two or more being flipped. So if we are trying to send the number 23, we will send

```
00010111
00010111
00010111
```

If the processor receives this, then it can be reasonably confident that the intended number was indeed 23. But maybe there was interference and the processor receives:

```
00010111
01010111
00010111
```

Now it can be reasonably sure of all but the second bit. But either this second bit was transmitted as a 0 three times and one of those got flipped to a 1, or it was transmitted as a 1 three times and two of those got flipped to 0. Because we're estimating that even one bit getting flipped is not so likely, the odds of it being the former situation vastly outweigh the odds of it being the latter. So we can make the educated guess that it was probably sent as a 0 and just one of the three transmissions got flipped. Thus, even though there was interference, the processor can still safely conclude the number transmitted was 00010111.

This is an example of an **error-correcting code**--that is, an encoding for the data that is done in such a way that even if some bits of the data were accidentally (or maliciously!) flipped in transit, the intended message can still be deduced from the error-filled data that the processor received.

It is worth mentioning that this error-correcting code can correct up to 8 errors, if they happen to all fall in the right places. For instance, if the intended message was again

```
00010111
00010111
00010111
```

But the processor receives

```
10100111
01010111
00011101
```

then it can still analyze this by taking, for each bit, the majority value. For instance, the first bit was sent as 1 once and as 0 twice, so the processor assumes this was a 0. Continuing thus, we get:

```
10100111
01010111
00011101
00010111
```

However, if the errors happened to line up so that one bit got flipped twice, as in

```
01010111
01011111
00000111
```

then the processor would deduce an intended value of 01010111, which would be incorrect. So an error-correcting code cannot correct arbitrary errors. They are therefore designed to correct for as many errors as are considered both plausible and intolerable in the situation they are designed for. In the case of a mobile phone accelerometer, an error may result in your phone's screen rotating when it shouldn't once in a while, so if errors are so rare that this will only happen on average once per year, say, there may not be much reason to use any error-correction. On the other hand, systems used in satellites are exposed to all the radiation and associated interference from the sun (as they are unprotected by our atmosphere), and they can have more mission-critical applications, so they may have some very beefy error-correcting codes indeed.

6.6.2: Sensor data with parity-bit error detection

The above scheme of sending the same value three times required rather a lot of work: In fact it required tripling the number of transmissions being made. If you have a lot of money and cannot afford error, this may be the way to go, but if you cannot justify tripling the cost of your hardware, then a slightly lighter-weight method may be more appropriate. One such method is to use a **parity bit**.

In this encoding scheme, we send only one extra bit in addition to the required 8, rather than an additional 16. This extra bit will be sent as 0 if the number of 1s in the transmitted number is even, and 0 if it is odd. If we intend to send the number 23, i.e. 00010111, then because this has an even number of 1s, we actually send the number 00010111**0**, where this last bit is the so-called parity bit.

So if the processor receives the transmission 010101110, then it notices that the number has an odd number of 1s, but the parity bit is 0. Therefore it concludes that some bit in the transmission was corrupted. Unlike in the previous scheme, it cannot tell which bit was corrupted. The intended transmission might have been 010001110 but then the 4th bit got flipped. Or it might have been 010101111 and the parity bit got flipped! All the processor can determine is that the transmission was likely in some way corrupted in transit.

This is an example of an **error-detecting encoding**--it allows us to determine that there was an error, but not to fix it. This may be suitable in cases where errors are not too harmful, and where the cost of re-sending the data is low.

But in this example, this is fine, since the processor can in this event just ask the accelerometer for a new reading until it gets one with a correct parity bit.

Note that just like in the previous example, if two bits happen get flipped in transit, e.g. 000101101 then the error will go unnoticed, since the number now has 3 1s and the parity bit is correspondingly 1, as we expected. Once again, encodings that are robust against transmission error are always designed to protect against specified levels of corruption depending on the application. They can never defend against all possible errors.

6.7: Case-studies revisited

We've now at long last completed our study of the layers that allow complex systems to be built on top of computers! In the next chapter we'll launch into the world of how these computers are actually built, but to round off the systems part of this text, we'll conclude by revisiting our three case-studies.

At the very beginning of this text, we made the claim that a computer, with only four basic kinds of instructions, can form the foundation for these three applications. Now that we know in detail what a computer can do, we'll take one simple example of usage for each application and make a quick trip through the inner workings of the application to see that computers, in our very narrow sense, are all that are needed to make these usages possible.

6.7.1: Search engine: Life cycle of a search

For our search engine example, we'll presuppose a search engine has already crawled the internet and all the documents we want to be able to search. Into this world walks our user, who has opened a browser on a computer and subsequently types a query. This is supposed to cause links to webpages that match this query to appear on the computer's screen. Let's watch how this happens in detail:

- User presses buttons on keyboard.
- Keyboard hardware outputs corresponding numbers numbers and sends them to the computer.
- Browser program was written in a high-level language and uses something like the Python input() command (though a bit more refined) to accept the user's text. This was compiled to ISA instructions, which are running on the user's computer. The input() command was compiled to input ISA instructions (like "in") being run in a loop, waiting to receive inputs from the keyboard. Since the keyboard is now sending numbers, these input instructions receive numbers.
- The browser stores the ASCII (or perhaps UTF-8) character encodings corresponding to the buttons the user has pressed in RAM (so that it will be ready to send the user's query when "Enter" is pressed).
- The browser should also display the letters the user is typing as they are typed. To this end, the browser program has already stored in RAM encoded graphics (much like the smiley-face example above) for all the possible letters. (This set of pictures is called a "font".) Having got the numbers from the keyboard, the browser executes "load from RAM" instructions to load the corresponding encoded pictures from RAM. It then executes output instructions to send these encoded pictures to the screen, causing the letters that the user typed to be displayed.
- The browser eventually receives numbers corresponding to "Enter" button keypress. It now executes output instructions to send the encoded (ASCII or UTF-8, say) search query to the network card along with some additional metadata telling the network card to send these numbers out on the network specifically to the search engine server.
- Search engine server network card receives query and communicates them to the search engine computer.
- The search engine computer is constantly executing input instructions to check for input and receives the query in the course of doing so.
- The search engine computer then executes input instructions to read from a hard drive the various documents it has stored (all, of course, encoded as well).
- The search engine computer then executes comparison instructions to compare the numbers that encode the query string with the stored numbers that encode the searchable documents. As it does this, it also executes arithmetic instructions to count how many matches it finds in each document and makes a note of the highest few counts it has seen thus far.
- The search engine computer then outputs numbers that encode links to the highest-scoring documents to its network card with instructions to please send these numbers to the computer that sent the original request.
- The search engine computer's network sends these numbers back to the user's computer on the network.
- The browser program is waiting for the search results to come back, and so is repeatedly executing input instructions. It therefore receives the input coming in from the network card, which are numbers that encode the .
- The browser then takes these numbers, loads from RAM up the corresponding pictures of letters for each number, and displays those pictures on the screen, which the user can read as search results.

6.7.2: Game console: Life cycle of a button press

The user presses a button on the game console's controller. This is supposed to cause a character on screen to move forward. If the character in doing so knocks something to the ground, the player gets points.

From the user's perspective, they press the button and the picture on the screen changes in a way that looks like their character moving and a points counter possibly incrementing. If all this is the work of a computer, then somehow all that happened was rapid manipulation of electronic representations of numbers. Let us now consider what this looks like:

- User presses button
- Game was written in high-level language and calls a library function to listen for button presses.

- The compiled version of this function is executing input instructions that eventually receive the information about the button press.
- The program then takes the button press and runs code to model the change in game state. Since the character should move forward, this means the x-coordinate of the player should be increased. In high-level code, this looks like:

```
player_x = player_x + 5
```

  But the code that is actually being run on the computer is of course the compiled version of this--something more like

```
ldi r31, 5
add r30,r31
```

- After updating the player's position, the game has to determine whether the player has collided with anything (and if so, to either move that something if it is movable, or to bump the player back if it is not). The program does this by comparing the positions of the character's outermost edges with the positions of all nearby objects (naturally, using comparison instructions) and updating positions accordingly. The programmer who wrote the game would have coded this with if statements like

```
if(object_is_movable[1]):
   object_x_coordinate = object_x_coordinate + 5
else:
   player_x = player_x - 5
```

  which will be compiled to ISA code using branch instructions that will actually be run on the console computer.
- Now that the scene is modeled, the score needs to be updated. To determine how to modify the score, further comparisons are done between objects and the ground like:

```
if(object_y_coordinate[1] <= 0):
   player_score = player_score + 1
```

  This is, as usual, stored as encoded ISA instructions involving comparisons and branching.
- Now the scene is set: The program has numbers that describe the coordinates of all the objects on screen, and the program has stored encoded graphics of all the characters and objects that should appear. It could run some further processing code to combine all of these to one image and output that image to screen, but a game console usually has specialized rendering hardware. So instead, it can execute output instructions to output the information about the individual character and object graphics, their locations, and the text describing the current score all to some specialized hardware that combines them and in turn outputs the image to the screen.

6.7.3: Mobile phone: Life cycle of a text message

The interface for text messaging is perhaps the simplest of all: user enters a friend's phone number and some text on their phone, presses send, and that text appears on the friend's phone. In terms of the fundamental intermediate steps that enable this, this example is not too different from the others:

- Phone displays keyboard on its touchscreen.
- User presses areas on the touch screen
- Phone executes input instructions from the touch screen to get the coordinates where the user touched..
- Phone has stored the coordinates of all the keys it displayed, and executes comparison instructions to see whether the user pressed within one of the keys and if so, which one.
- Phone finds that user pressed "M" key. Stores the number 77 (ASCII for "M") into RAM as a part of the text message.
- User eventually finishes and pushes down on the part of the touch screen displaying a picture of the "Send" button.
- Phone performs more comparisons of coordinates and discovers that the user indeed pressed "Send".
- Phone takes the text message and prepares to output it to the radio. Radio is a very noisy means of communication, and the cellular network tower may receive a corrupted version of what the phone's radio sends due to all kinds of interference from other phones, from gamma rays, distortion from the fact that the phone is moving as it sends the message, and who knows what else. So before the phone outputs the encoded message to the radio, it further encodes it with an error-correcting code. For example, the message "Meow", which would normally be encoded with

```
01001101
01100101
01101111
01110111
```

would be encoded using a three-repetition error-correcting code as:

```
01001101
01001101
01001101
01100101
01100101
01100101
01101111
01101111
01101111
01110111
01110111
01110111
```

- Phone now outputs the text message, encoded in ASCII and then encoded further in an error-correcting code, to the radio.
- Friend's phone is executing input instructions and receives a message on its radio indicating the presence of a new text message. It executes input instructions to get the encoded message into its RAM.
- Friend's phone then checks that in every block of three bytes, all the bits are identical. It finds that in one of them--the three bytes representing the second character--there are differences:

```
01100101
01000101
01100100
```

Since two of the bytes have third bit 1, however, it concludes that the other byte must have originally had third bit 1 as well. Likewise for the 8th bit, and it therefore uses 01100101 as its interpretation of the character.

For example, the programmer would have written something like:

```
if(byte1 == byte2 and byte2 == byte3):
   text_message[i] = byte1
```

to say that if the three bytes are all the same, then just use the first byte as the corresponding character in the text message.

- Now having decoded the error-correcting code and having stored ASCII encoding of the characters corresponding to the original text message, the phone also has a font stored. Since the first character is 77, it looks up the 77th picture in the font and uses output instructions to communicate that picture to the screen, ultimately showing the user an 'M'. The second character is the number 101, so it displays picture number 101 onto the screen, showing the user 'e', and so on until the user sees the whole message.

6.8: Exercises

6.1: What is encoding?

6.2: We create binary _____ to begin to encode AVR ISA instructions.

1. functions
2. representations
3. methods
4. casings

6.3: What are the five groups of information required used to differentiate different instructions?

6.4: What is an opcode in AVR ISA instructions?

6.5: The “C” bits of an AVR instruction represent its ____, which specifies _____.

1. operand; the source/destination registers or immediate value
2. opcode; the source/destination registers or immediate value
3. operand; which instruction is to be performed
4. opcode; which instruction is to be performed

6.6: What differentiates the encodings for dec, com, and inc (e.g. how can you distinguish a 16-bit dec instruction from com, inc)? What do they have in common?

6.7: Now provide an example of your answer by writing out the encodings for dec r1, com r1, and inc r1.

6.8: What type of instructions follow the CCCCCCIIIIIIICCC format? What do the "I" bits represent?

6.9: What is ASCII? How is it used to encode?

6.10: What is UTF-8? How is it used to encode?

6.11: What benefits are there of using UTF-8 over ASCII?

6.12: Briefly explain why UTF-8 allows for many more characters than ASCII in terms of how many bytes they each allow for their encodings.

6.13: What is the significance of a UTF-8 encoding that starts with a 0? Give an example to explain your answer.

6.14: If a UTF-8 encoding starts with "11110", what does this signify about the following 3 bytes?

6.15: Just as letters are encoded in text, _____ are encoded in games.

6.16: What is a pixel?

6.17: How many bytes are used to represent colors which represent light for a computer?

6.18: Fill in the red, green, and blue values for the following colors (refer to the table in 6.4.1) in their HEXADECIMAL representation. Fill the rightmost column in with their full RBG encoding.

Color	Red	Blue	Green	RGB
White	0xFF	0xFF	0xFF	0xFFFFFF
Red	0xFF	0x00	0x00	0xFF0000
Green	0x00	0xFF	0x00	0x00FF00
Blue	0x00	0x00	0xFF	0x0000FF
Grey	0xCC	0xCC	0xCC	0xCCCCCC
Magenta	0xFF	0x00	0xFF	0xFF00FF
Puce	0x72	0x2F	0x37	0x722F37
Black	0x00	0x00	0x00	0x000000

6.19: What is error-correcting code and error-detecting encoding, and how are they similar and different?