

First Steps to C Mastery





Today's Lecture

First steps towards C mastery

- Arithmetic: signed, unsigned, wraparound
- Software: hallmarks of good software
- Pointers, structs, and memory

Some fun: MIDI

1 -Bit Adder

Add 2 1-bit numbers

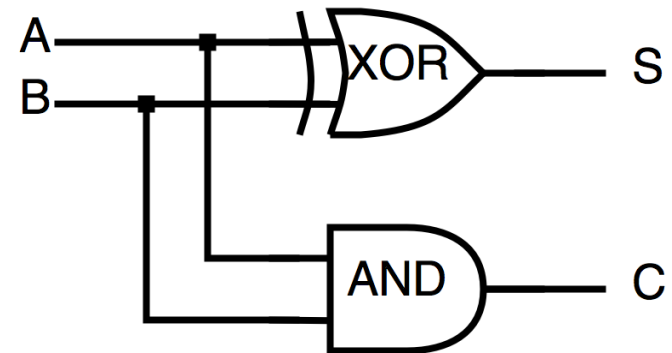
a	b	sum
0	0	00
0	1	01
1	0	01
1	1	10

Add 2 1-bit numbers (Half Adder)

a	b	sum
0	0	00
0	1	01
1	0	01
1	1	10

bit 0 of sum: $S = a \oplus b$

bit 1 of sum: $C = a \& b$



Have reduced addition to bitwise,
logical operations!

Add 3 1-bit numbers

a	b	c	=	s	c
0	0	0		0	0
0	1	0		0	1
1	0	0		0	1
1	1	0		1	0
0	0	1		0	1
0	1	1		1	0
1	0	1		1	0
1	1	1		1	1

Add 3 1-bit numbers (Full Adder)

a b ci = s co

0 0 0 0 0

0 1 0 0 1

1 0 0 0 1

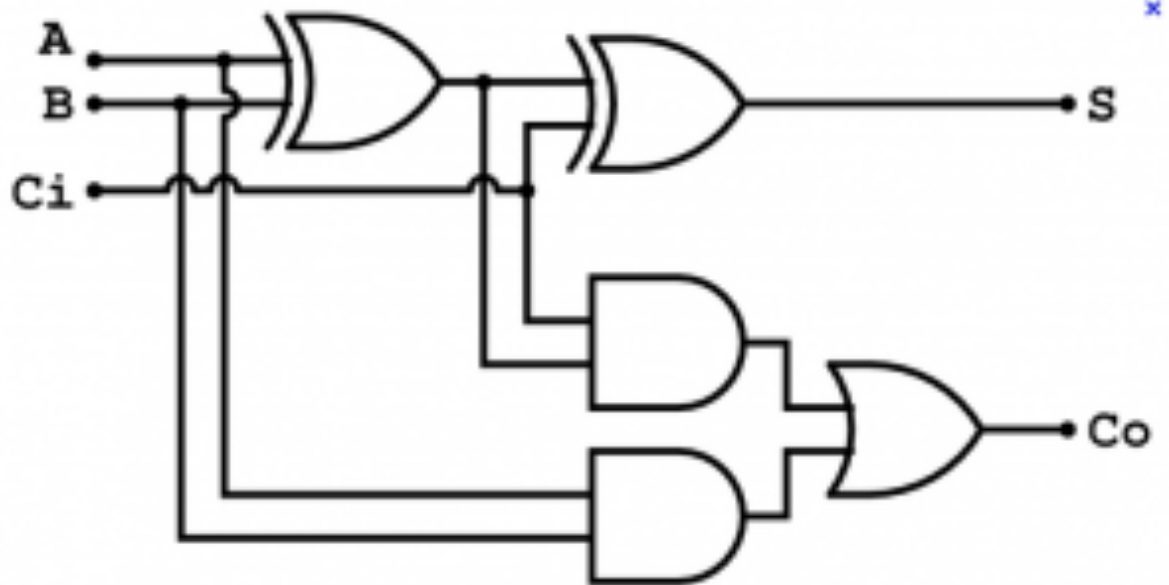
1 1 0 1 0

0 0 1 0 1

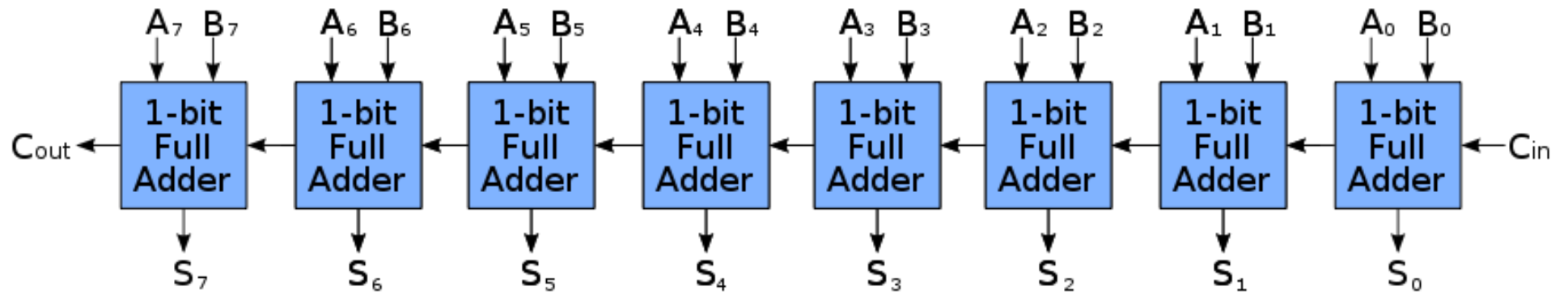
0 1 1 1 0

1 0 1 1 0

1 1 1 1 1



8-bit adder



CLMUL instruction set

From Wikipedia, the free encyclopedia

Carry-less Multiplication (CLMUL) is an extension to the [x86](#) instruction set used by [microprocessors](#) from [Intel](#) and [AMD](#) which was proposed by Intel in March 2008^[1] and made available in the [Intel Westmere processors](#) announced in early 2010.

One use of these instructions is to improve the speed of applications doing block cipher encryption in [Galois/Counter Mode](#), which depends on [finite field](#) $GF(2^k)$ multiplication, which can be implemented more efficiently^[2] with the new CLMUL instructions than with the traditional instruction set. Another application is the fast calculation of [CRC values](#),^[3] including those used to implement the [LZ77 sliding window DEFLATE](#) algorithm in [zlib](#) and [pngcrush](#).^[4]

Contents

[hide]

1

New instructions

2

CPUs with CLMUL instruction set

3

See also

4

References

New instructions [edit]

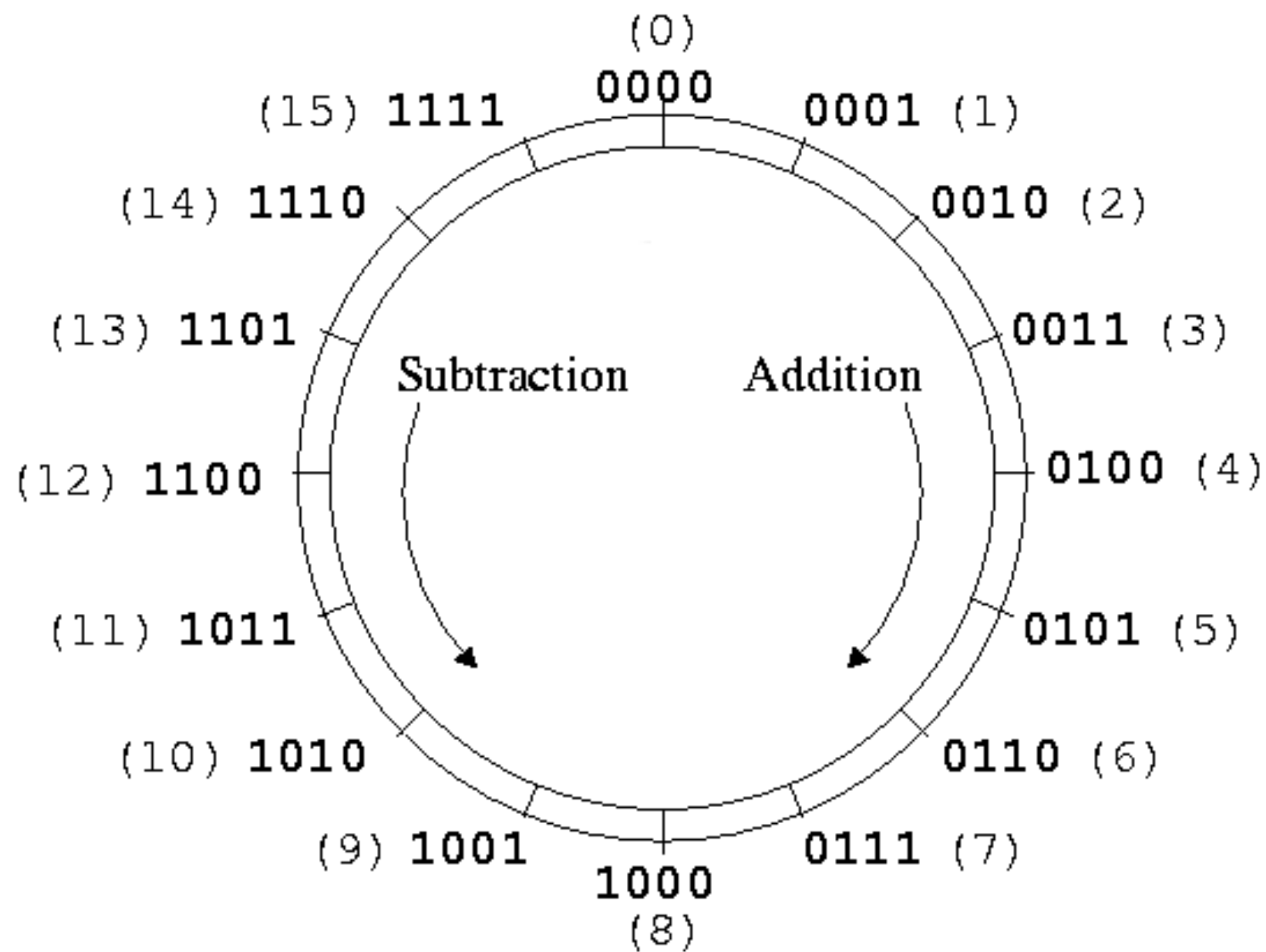
The instruction computes the 128-bit [carry-less product](#) of two 64-bit values. The destination is a [128-bit XMM register](#). The source may be another XMM register or memory. An immediate operand specifies which halves of the 128-bit operands are multiplied. Mnemonics specifying specific values of the immediate operand are also defined:

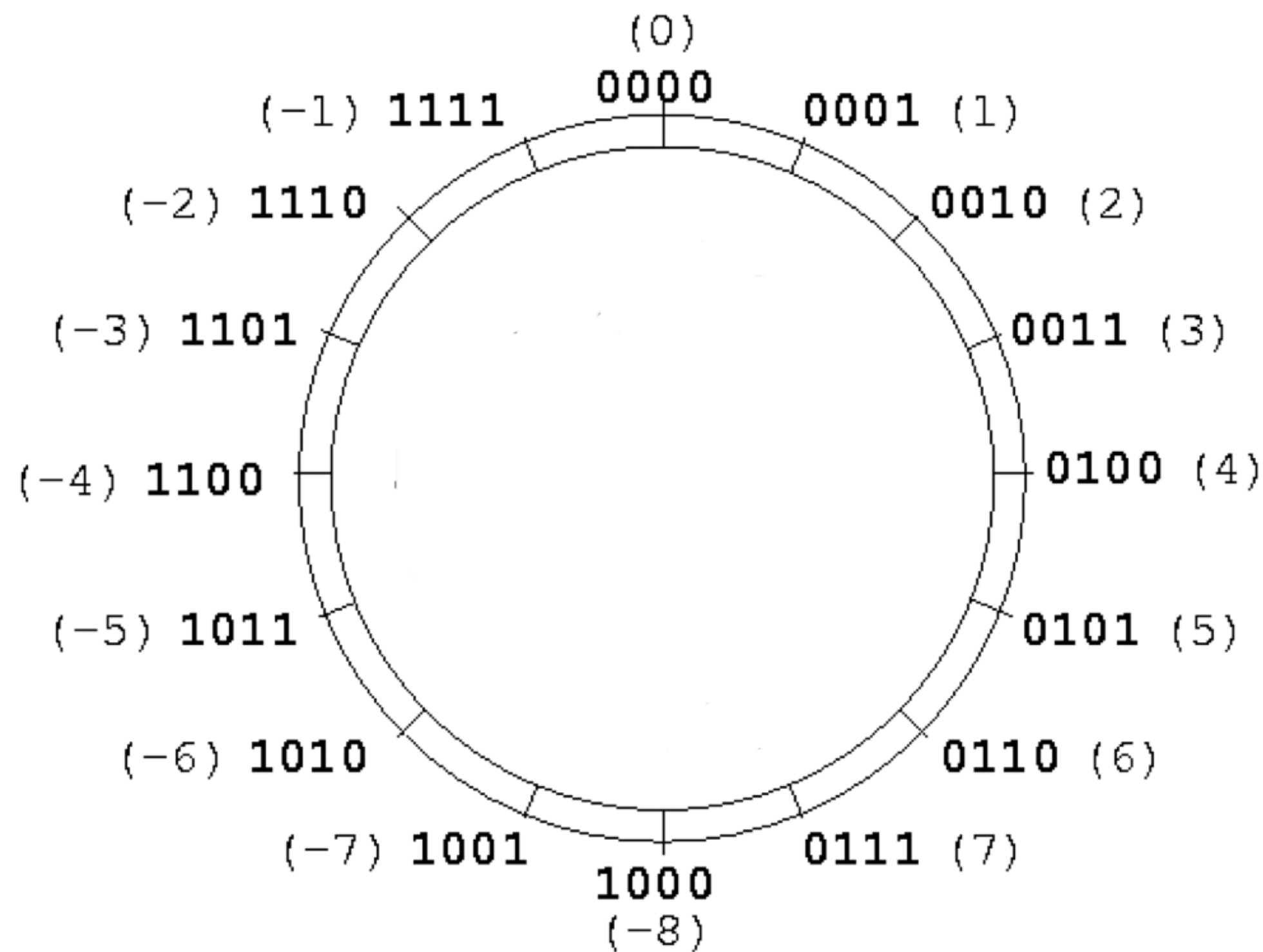
Instruction	Opcode	Description
<div>PCLMULQDQ</div> <div><code>xmmreg, xmmrm, imm</code></div>	<div>[rmi: 66 0f 3a 44 /r</div> <div>ib]</div>	Perform a carry-less multiplication of two 64-bit polynomials over the finite field $GF(2)$.
<div>PCLMULLQLQDQ</div>	<div>[rm: 66 0f 3a 44 /r</div>	Multinlv the low halves of the two registers

9 9 9 9 9 9

9 9 9 9 9 9

0 0 0 0 0 0





Numerical Representations

We've been talking about two's complement

- Negative numbers add 1 and flip the bits

But there's one's complement

- Negative numbers just flip the bits!

And also BCD (binary coded decimal)

- Each 4 bits encode a decimal digit

Generally settled on two's complement as best representation

But What Type Is It?

```
unsigned int b=2;  
int a=-2;
```

```
if(a>b)  
    printf("a>b");  
else  
    printf("b>a");
```

```
int b=2;  
int a=-2;
```

```
if(a>b)  
    printf("a>b");  
else  
    printf("b>a");
```


But What Type Is It?

```
unsigned int b=2;  
int a=-2;
```

```
if(a>b)  
    printf("a>b");  
else  
    printf("b>a");
```

```
int b=2;  
int a=-2;
```

```
if(a>b)  
    printf("a>b");  
else  
    printf("b>a");
```

C converts a into unsigned

Conversions Everywhere

```
#include <stdio.h>
#include <limits.h>
#include <assert.h>

int main(void) {
    assert(sizeof(unsigned char)==1);

    unsigned char uc1 = 0xff;
    unsigned char uc2 = 0;

    if(~uc1 == uc2) {
        printf("%hhx == %hhx\n", ~uc1, uc2);
    } else {
        printf("%hhx != %hhx\n", ~uc1, uc2);
    }
    return 0;
}
```



Operand Conversion

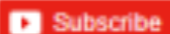
	u8	u16	u32	u64	i8	i16	i32	i64
u8	i32	i32	u32	u64	i32	i32	i32	i64
u16	i32	i32	u32	u64	i32	i32	i32	i64
u32	u32	u32	u32	u64	u32	u32	u32	i64
u64	u64	u64	u64	u64	u64	u64	u64	u64
i8	i32	i32	u32	u64	i32	i32	i32	i64
i16	i32	i32	u32	u64	i32	i32	i32	i64
i32	i32	i32	u32	u64	i32	i32	i32	i64
i64	i64	i64	i64	u64	i64	i64	i64	i64

`code/wraparound`

- In two's complement, when you exceed the maximum value of int (2,147,483,647), you “wrap around” to negative numbers:



PSY - GANGNAM STYLE (강남스타일) M/V

 officialpsy 

 7,600,830



-2142584554

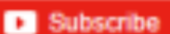
+ Add to < Share ... More

 8,761,309  1,139,933

- Here is the link after Google upgraded to 64-bit integers:



PSY - GANGNAM STYLE (강남스타일) M/V

 officialpsy 

 7,600,830

2,152,382,740

+ Add to < Share ... More

 8,761,309  1,139,933

Writing Good Systems Software

```

void serial_init() {
    unsigned int ra;

    // Configure the UART
    PUT32(AUX_ENABLES, 1);
    PUT32(AUX_MU_IER_REG, 0); // Clear FIFO
    PUT32(AUX_MU_CNTL_REG, 0); // Default RTS/CTS
    PUT32(AUX_MU_LCR_REG, 3); // Put in 8 bit mode
    PUT32(AUX_MU_MCR_REG, 0); // Default RTS/CTS auto flow control
    PUT32(AUX_MU_IER_REG, 0); // Clear FIFO
    PUT32(AUX_MU_IIR_REG, 0xC6); // Baudrate
    PUT32(AUX_MU_BAUD_REG, 270); // Baudrate

    // Configure the GPIO lines
    ra = GET32(GPFSEL1);
    ra &= ~(7 << 12); //gpio14
    ra |= 2 << 12;     //alt5
    ra &= ~(7 << 15); //gpio15
    ra |= 2 << 15;     //alt5
    PUT32(GPFSEL1, ra);
    PUT32(GPPUD, 0);
    for (ra = 0; ra < 150; ra++) dummy(ra);
    PUT32(GPPUDCLK0, (1 << 14) | (1 << 15));
    for (ra = 0; ra < 150; ra++) dummy(ra);
    PUT32(GPPUDCLK0, 0);

    // Enable the serial port (both RX and TX)
    PUT32(AUX_MU_CNTL_REG, 3);
}

```



```
void uart_init(void) {  
    int *aux = (int*)AUX_ENABLES;;  
  
    *aux = AUX_ENABLE; // turn on mini-uart  
  
    uart->ier = 0;  
    uart->cntl = 0;  
    uart->lcr = MINI_UART_LCR_8BIT;  
    uart->mcr = 0;  
    uart->ier = 0;  
    uart->iir = MINI_UART_IIR_RX_FIFO_CLEAR |  
                MINI_UART_IIR_RX_FIFO_ENABLE |  
                MINI_UART_IIR_TX_FIFO_CLEAR |  
                MINI_UART_IIR_TX_FIFO_ENABLE;  
  
    uart->baud = 270; // baud rate  $((250,000,000/115200)/8)-1 = 270$   
  
    gpio_set_function(GPIO_TX, GPIO_FUNC_ALT5);  
    gpio_set_function(GPIO_RX, GPIO_FUNC_ALT5);  
  
    uart->cntl = MINI_UART_CNTL_TX_ENABLE |  
                MINI_UART_CNTL_RX_ENABLE;  
}
```



**Well-written software is easy for
someone to read and understand.**

Well-written software is easy for someone to read and understand.

Code that is easier to understand has fewer bugs.

Long comments != easy to read and understand.

Understand at the line, function, file, and structural levels.

Example: Julie's malloc implementation.

Lesson: Imagine someone else has to fix a bug in your code: what should it look like make this easier? *Hint: be a section leader, you'll have to read student code and you'll learn a lot.*

**Systems Code Is Terse But
Unforgiving**

Systems Code Is Terse But Unforgiving

Think about your PS/2 scan code reader: if any part of it is wrong, you won't read scan codes. It's only 20-30 lines of code!

The mailbox code you'll use for the frame buffer is ~10 lines of code: we once debugged it for 9 hours.

Lesson: if you know exactly what you have to do it can take only minutes; throwing away and rewriting can often be *faster*. Sunk cost fallacy!

```
void uart_init(void) {
    int *aux = (int*)AUX_ENABLES;;

    *aux = AUX_ENABLE; // turn on mini-uart

    uart->ier = 0;
    uart->cntl = 0;
    uart->lcr = MINI_UART_LCR_8BIT;
    uart->mcr = 0;
    uart->ier = 0;
    uart->iir = MINI_UART_IIR_RX_FIFO_CLEAR |
                MINI_UART_IIR_RX_FIFO_ENABLE |
                MINI_UART_IIR_TX_FIFO_CLEAR |
                MINI_UART_IIR_TX_FIFO_ENABLE;

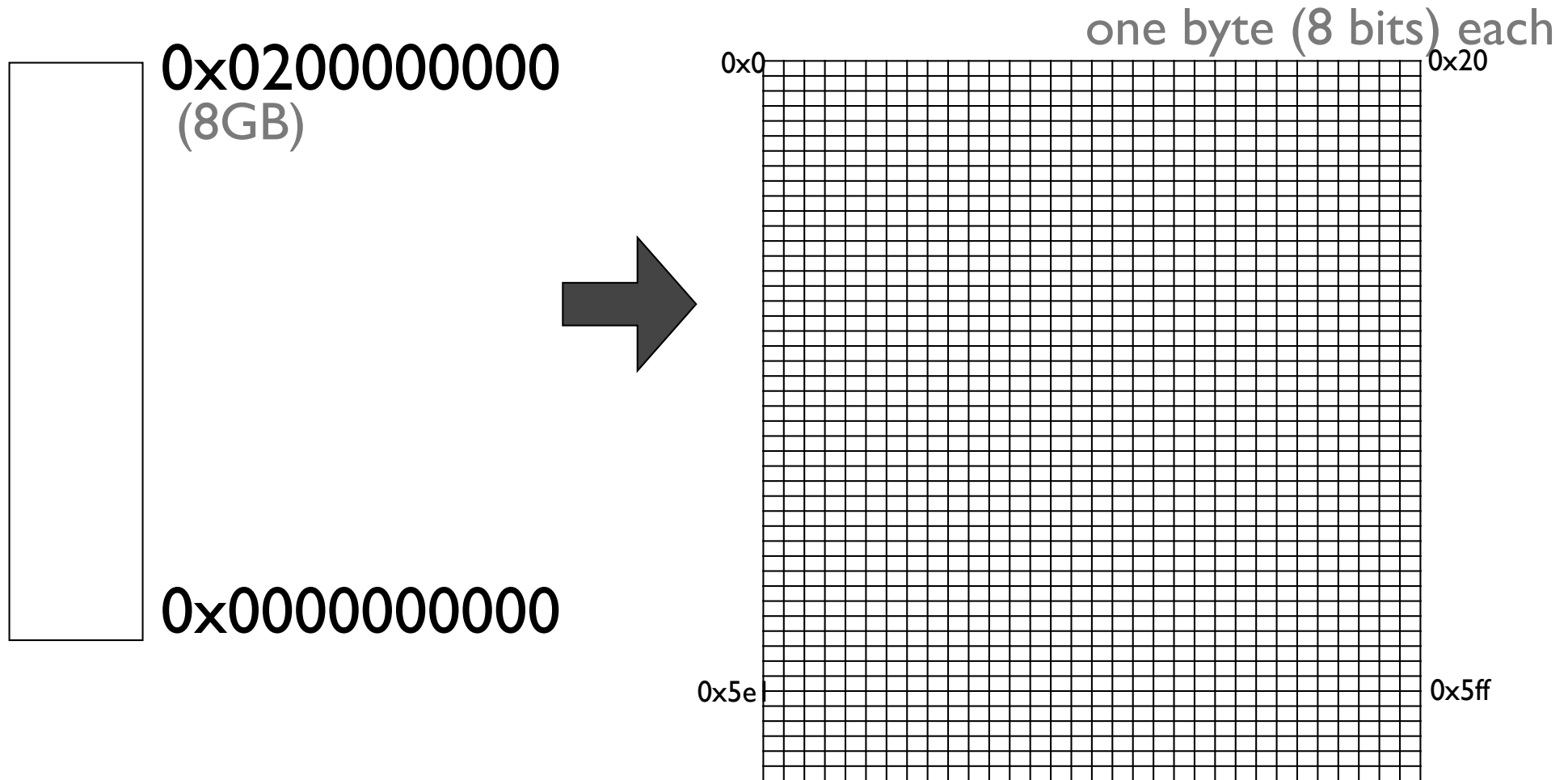
    uart->baud = 270; // baud rate ((250,000,000/115200)/8)-1 = 270

    gpio_set_function(GPIO_TX, GPIO_FUNC_ALT5);
    gpio_set_function(GPIO_RX, GPIO_FUNC_ALT5);

    uart->cntl = MINI_UART_CNTL_TX_ENABLE | MINI_UART_CNTL_RX_ENABLE;;
}
```

Pointers, Structures, Etc.

Computer Memory



Endianness

Multibyte words: how do you arrange the bytes?

1,024 = 0x0400 =

?	?
---	---

Little endian: least significant byte is at lowest address

- Makes most sense from an addressing/computational standpoint

0x00	0x04
------	------

Big endian: most significant byte is at lowest address

- Makes most sense to a human reader

0x04	0x00
------	------

“Arrays are Pointers”

Sort of true. Sometimes.

When are arrays and pointers different?

- A pointer is a location in memory storing an address (you can change the pointer)
- An array is a location in memory storing data (you can change the *elements* of an array, but not the array itself)

code/pointers

C structs

```
struct data {  
    unsigned char fields;  
    unsigned int num_changes;  
    char name[MAX_NAME + 1];  
    unsigned short references;  
    unsigned short links;  
}
```

How big is this structure?

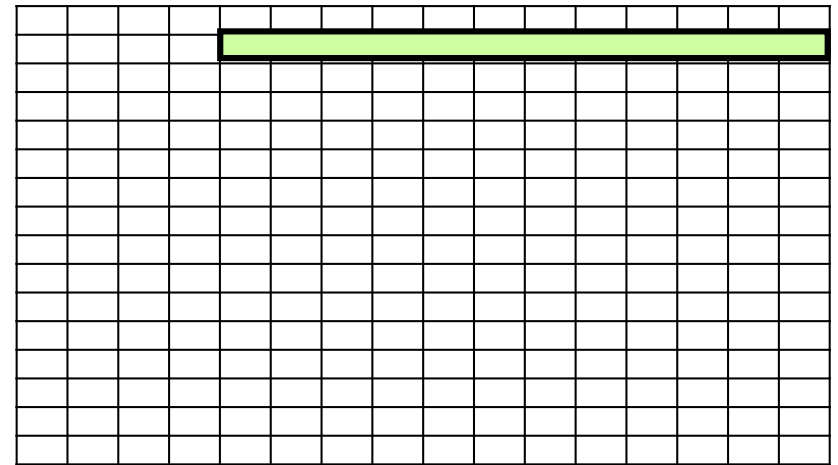
How is it laid out in memory (on an ARM)?

C structs

A C struct is just a shorthand and convenient way to allocate memory and describe offsets from a pointer.

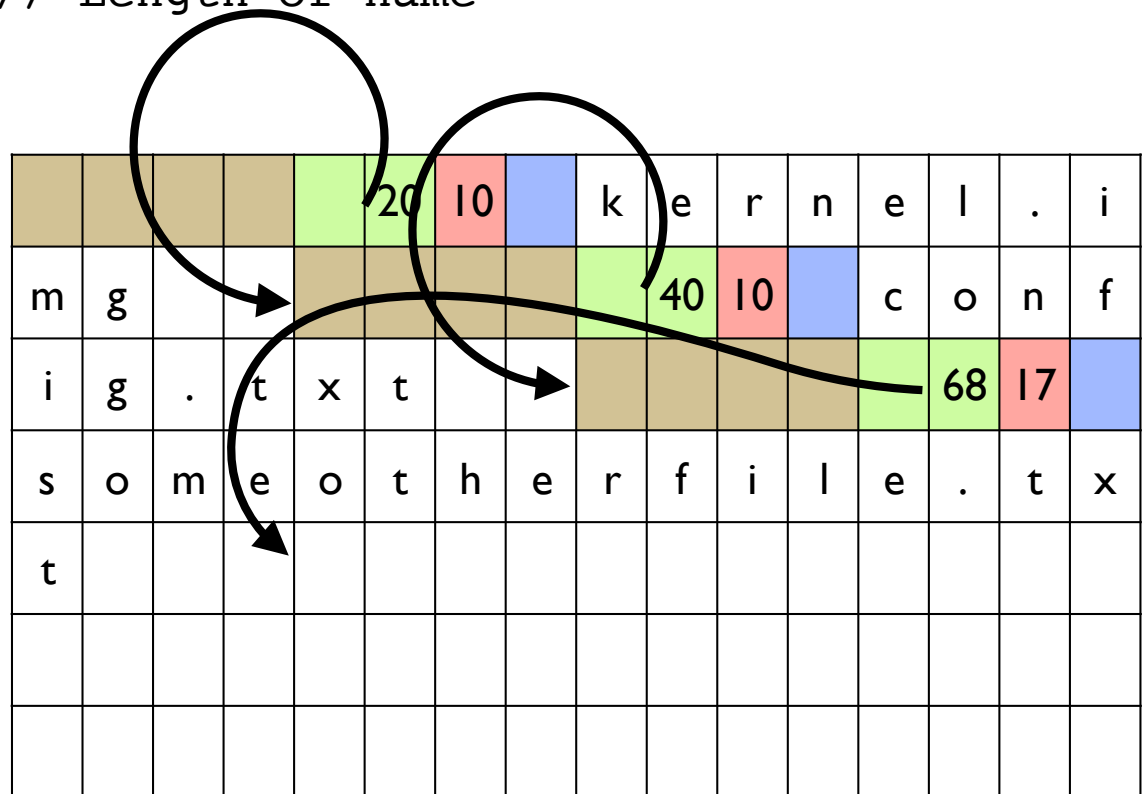
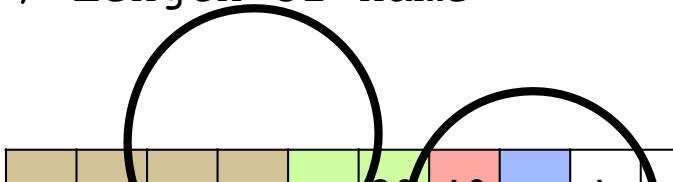
```
struct data {  
    unsigned char fields;  
    unsigned int num_changes;  
    char name[MAX_NAME + 1];  
    unsigned short references;  
    unsigned short links;  
}
```

```
struct data* d = ...;  
d->fields = 0;           // d + 0  
d->num_changes = 0;      // d + 4  
d->references = 0;       // d + 12  
d->links = 0;            // d + 14
```



File System Directory Entries

```
struct data {
    unsigned inode;
    unsigned short rec_len;    // Offset of next record
    unsigned char name_len;   // Length of name
    unsigned char file_type;
    char name[];
}
```



What happens if

```
struct data* = (struct data*)0x4; ?
```

MIDI

Digital != Analog

Our CPU is generating a square pulse wave

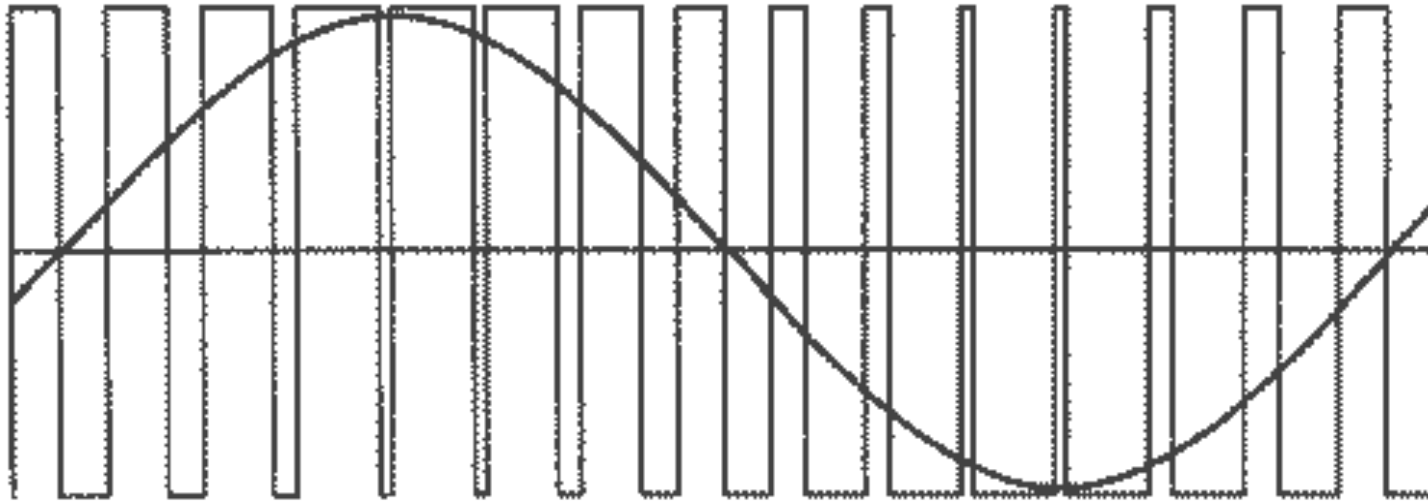
Interacts with electrical components: changing electric field, impedance, capacitance, etc.

- Note: cannot actually send pulse wave

These details are why building high-frequency circuits (e.g., radio, HDMI) requires very careful engineering

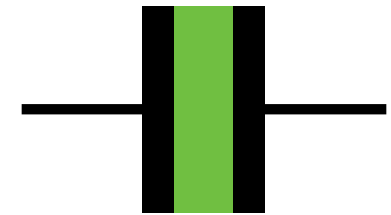
PWM to the Rescue!

Can simulate continuous values with fast enough PWM clocking: need hardware help

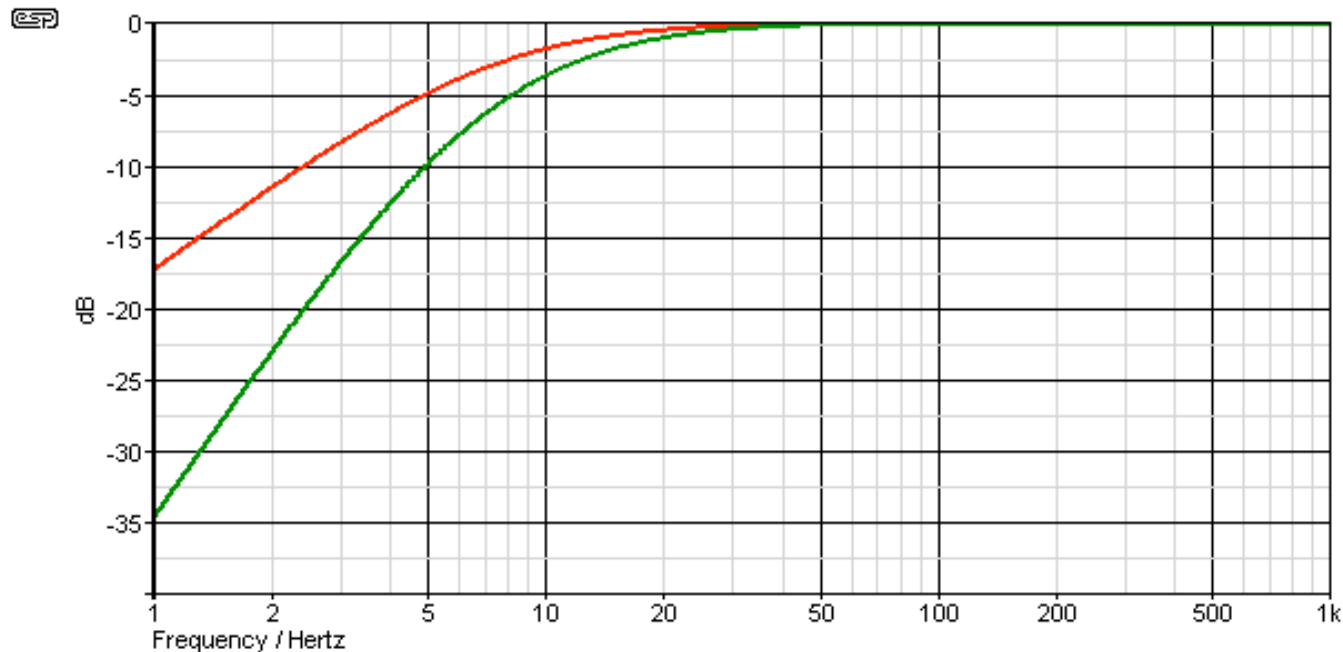


Capacitors

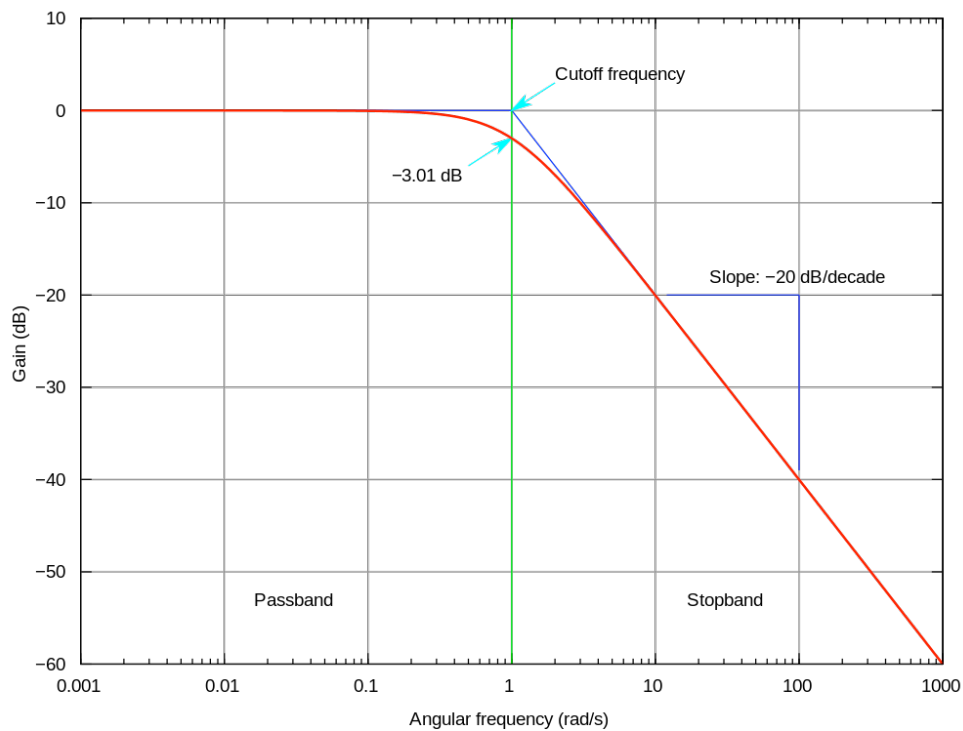
Two (tiny) plates separated by a non-conductive material (dielectric)



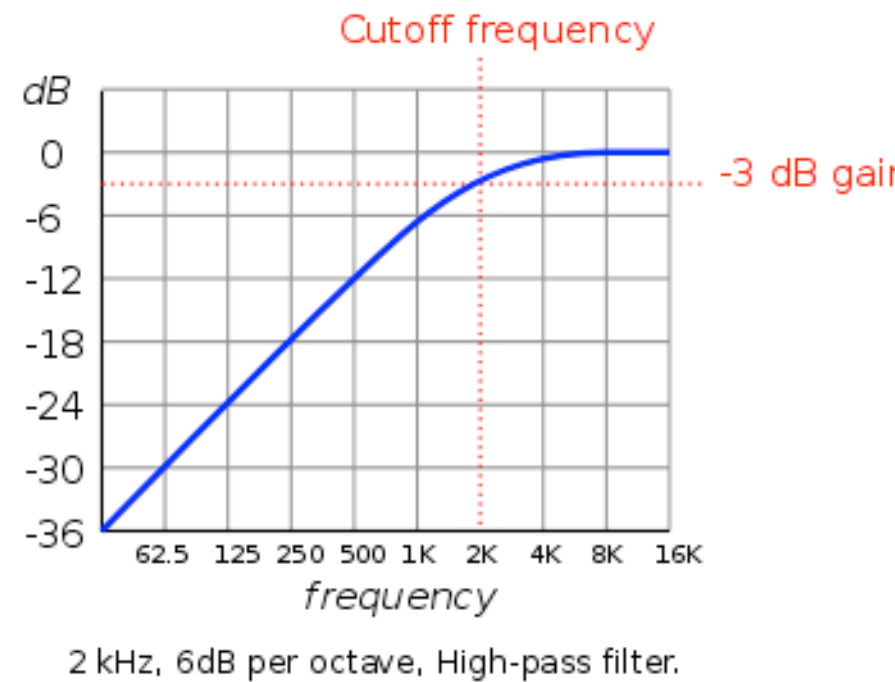
Frequency-selective impedance



Filters



low pass

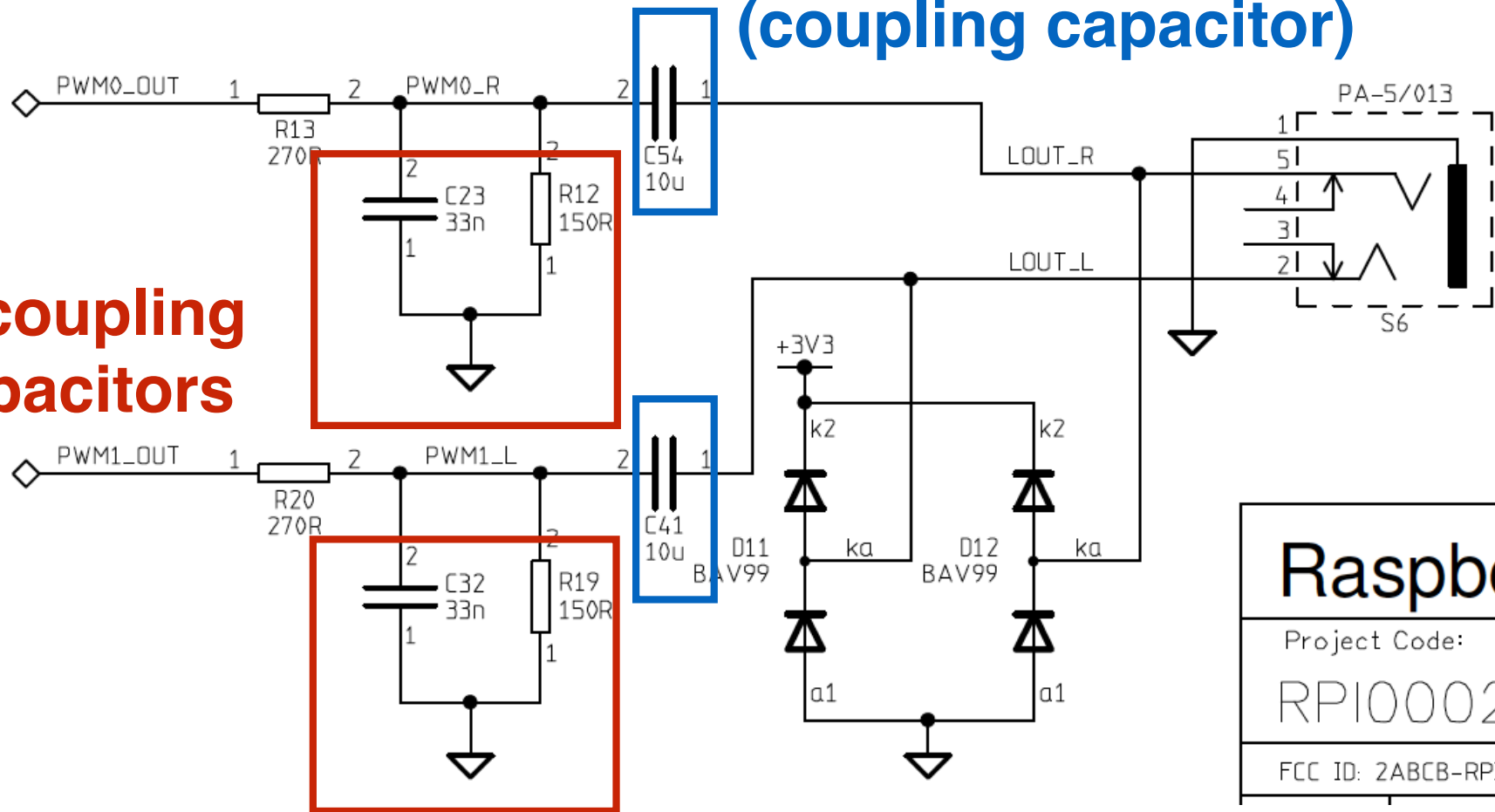


high pass

Raspberry Pi Audio Circuit

high pass filters
(coupling capacitor)

decoupling
capacitors



Raspberry

Project Code:

RPI00027

FCC ID: 2ABCB-RPI21

Hardware PWM Support

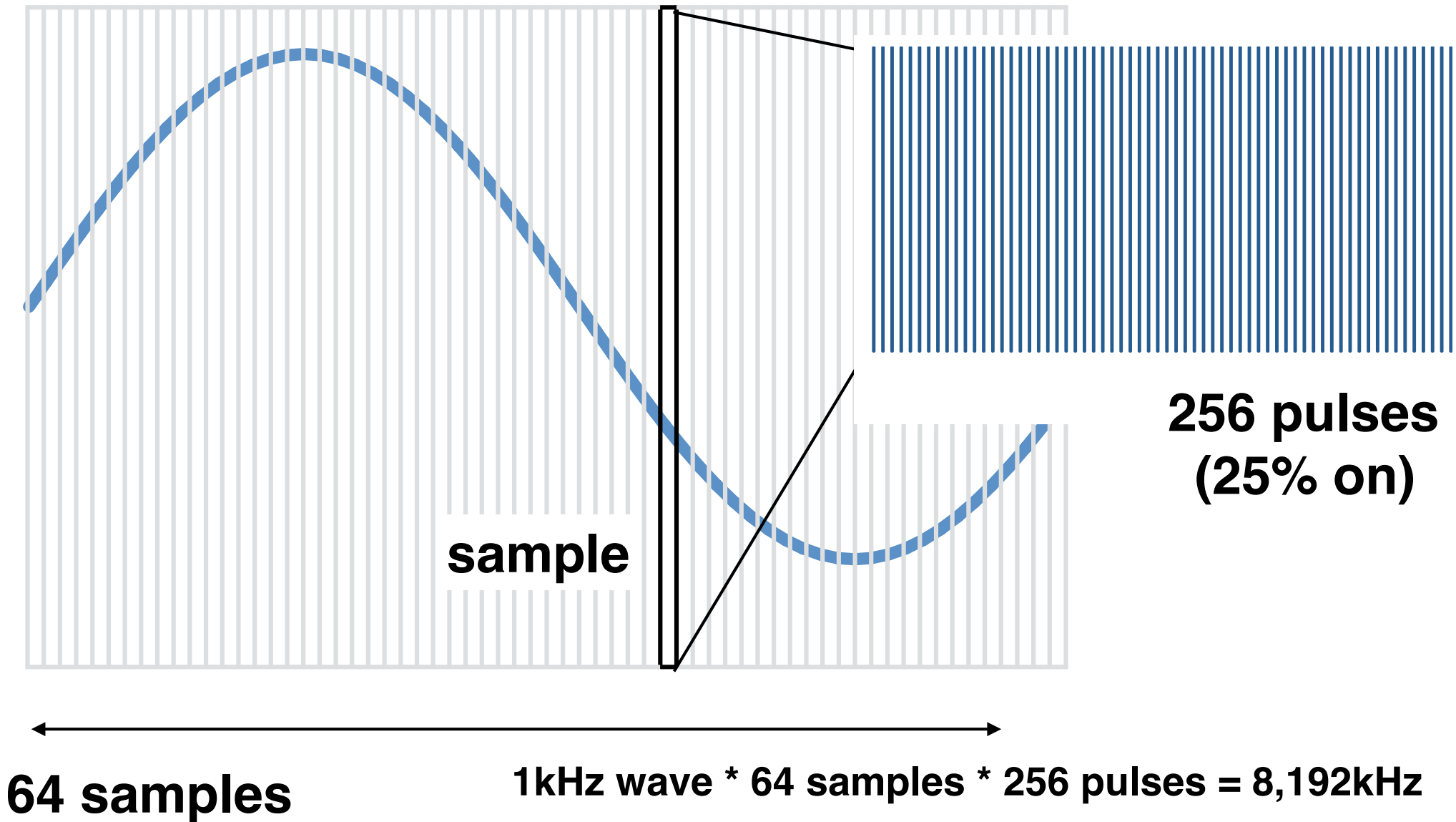
Start with a 19.2MHz clock, divide it to specify the time slots of on/off

E.g., divider of 2.375 = 8,192kHz

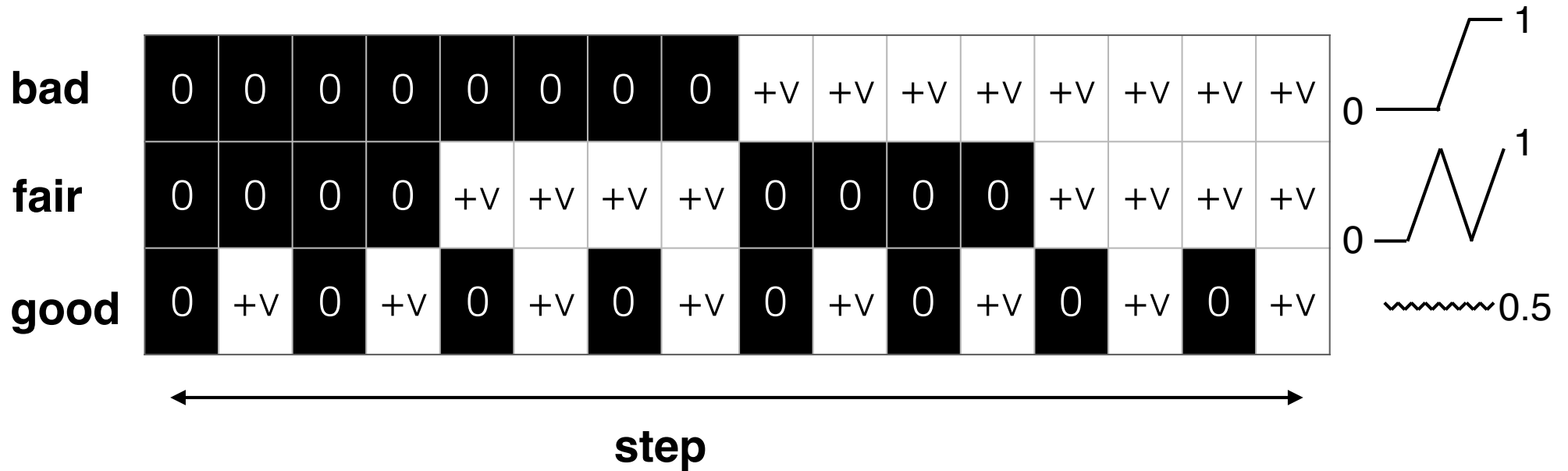
Divide wave into steps (e.g., 64)

Divide each step into train of (e.g., 256) pulses: tell hardware how many pulses should be high

Example: Sine



PWM Clocking of Pulses



What if we want real music?

MIDI

MIDI: Musical Instrument Digital Interface

Simple interface to control musical instruments

Emerged from electronic music and instruments in 1970s

First version described in Keyboard magazine in 1982

A bit of “music”

MIDI

31.25 kbps 8-N-1 serial protocol

Commands are 1 byte, with variable parameters
(c=channel, k=key, v=velocity, l=low bits, m=high bits)

Command	Code	Param	Param
Note on	1001cccc	0kkkkkkkk	0vvvvvvvv
Note off	1000cccc	0kkkkkkkk	0vvvvvvvv
Pitch bender	1110cccc	01111111	0mmmmmmmm

UART (2+ pins)

Bidirectional data transfer, no clock line — “asynchronous”.

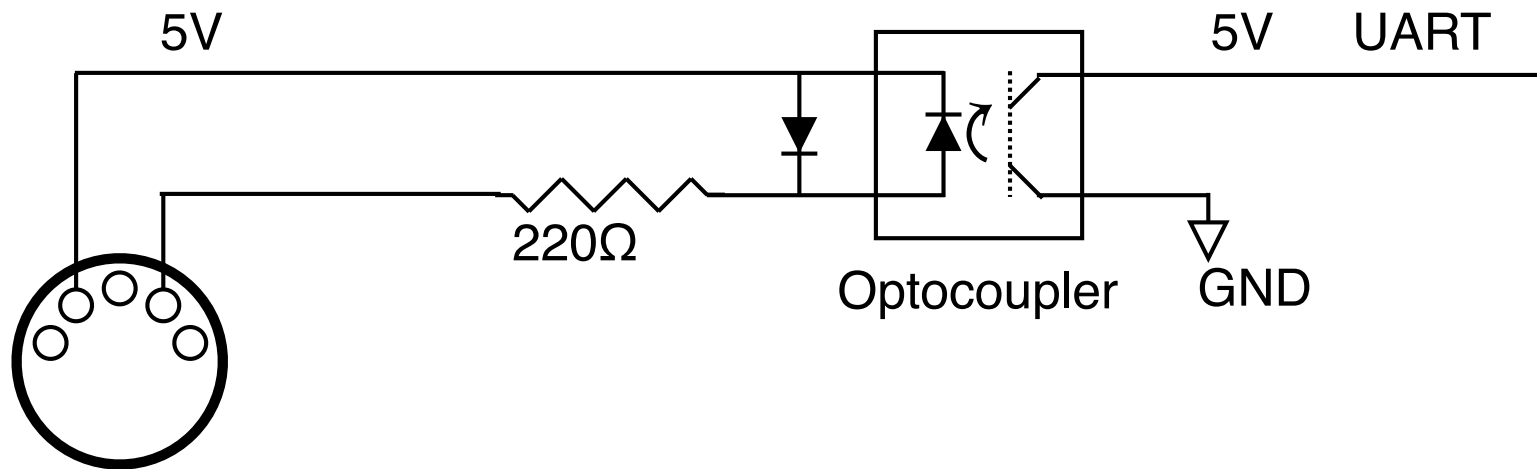
Additional pins for flow control (“I’m ready to send”), old telephony mechanisms.

Start bit, (5 to 9) data bits, (0 or 1) parity bit, (1 or 2) stop bit. 8-N-1:

[illegible]

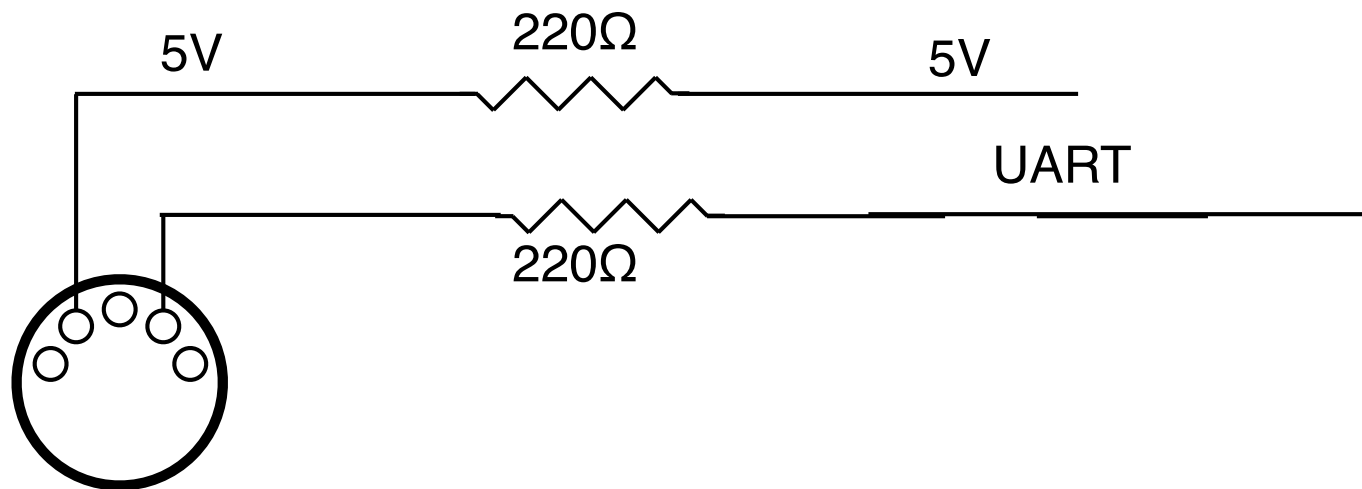
MIDI Circuit

0 is high, 1 is low!



Optocoupler completely isolates circuits electrically:
no noise in instrument

MIDI Hack!



`code/midi`

Raspberry Pi hooked up to KORG volta keys on GPIO pin 25.

UART timing

Inversion