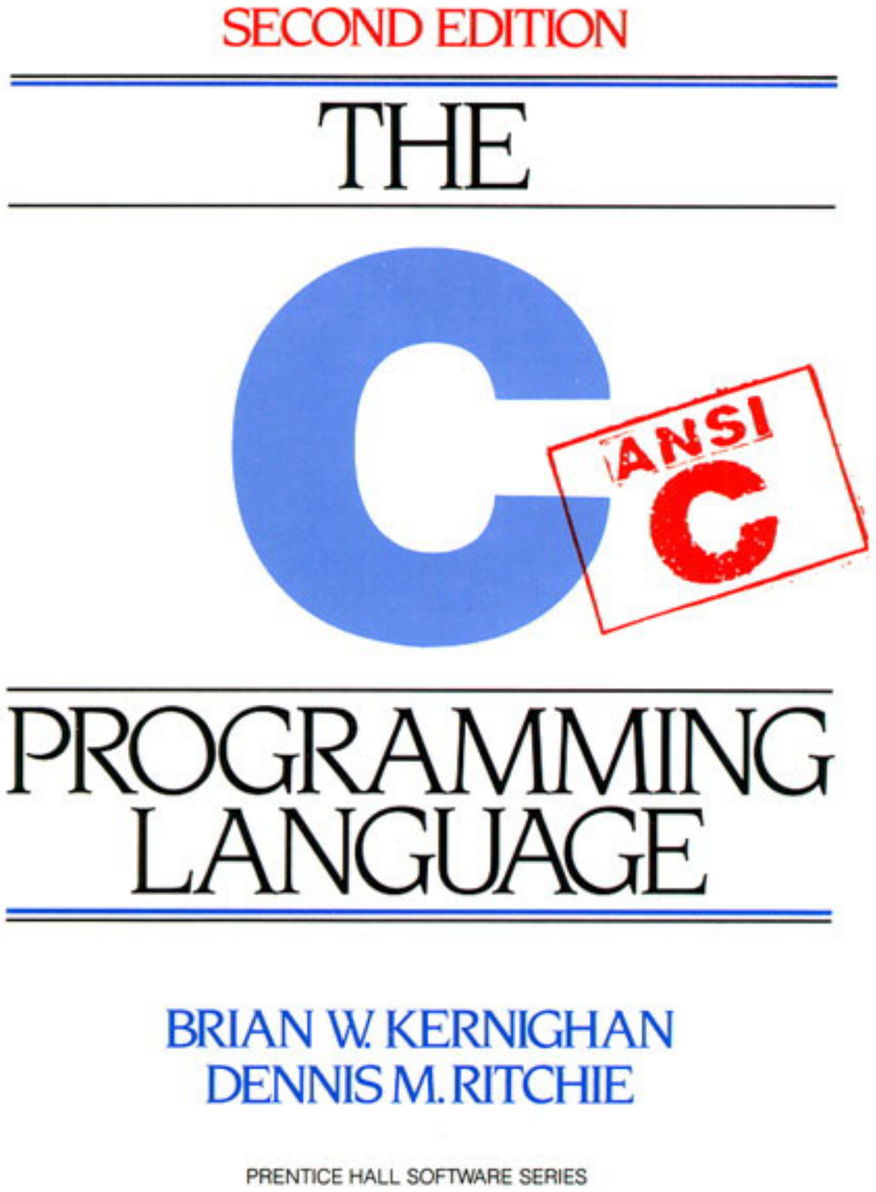
CS 107e
Lecture 4: From ASM to C Part 1

Friday, October 5, 2018

Computer Systems from the Ground Up
Fall 2018
Stanford University
Computer Science Department

Lecturer: Chris Gregg



Logistics

- Assignment 1 deadline on Wednesday, October 10th, 2018 at 11:30 AM (i.e., before Wednesday lab)

Question

If it takes one monkey one minute to eat one banana, how long does it take three monkeys to eat three bananas?

Question

If it takes one monkey one minute to eat one banana, how long does it take three monkeys to eat three bananas?

Answer: one minute

Today's Topics

- ARM condition codes and branch instructions
- Instruction Pipelining
- C language as "high-level" assembly
- How does C relate to ASM?
- What does a compiler do?

Control Flow in Assembly

- Assembly instructions are held in contiguous memory. Let's look at this tiny portion of a program:

```
// configure GPIO 20 for output
ldr r0, FSEL2
mov r1, #1
str r1, [r0]

FSEL2: .word 0x20200008
```

- If we disassemble it, this is what we get:

```
$ arm-none-eabi-objdump -d partial.o

partial.o:  file format ELF32-arm-little

Disassembly of section .text:
$a:
       0 :   04 00 9f e5 ldr r0, [pc, #4]
       4 :   01 10 a0 e3 mov r1, #1
       8 :   00 10 80 e5 str r1, [r0]

FSEL2:
       c:08 00 20 20 .word 0x20200008
```

Notice that the each instruction is four bytes, and they come right after another in memory (0, 4, 8 represent the memory locations). This is called *straight-line* code, because the instructions come one after the other, in a line.

Control Flow in Assembly

- As we do in high level languages, we want to be able to control the flow of programs based on *conditions*. Think about an `if` statement in C/C++/Java:

```
if (a == 42) {
    b = 7;
}
```

- The condition that must be met to go into the `if` statement body is that a is equal to 42.

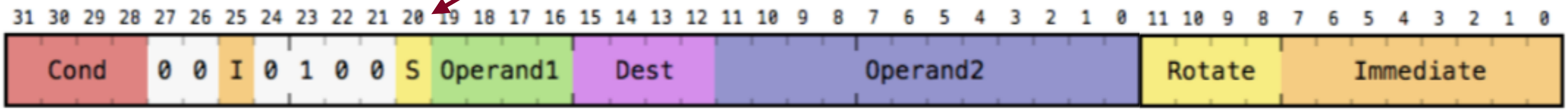
In ARM assembly, we can have an instruction *set the condition flags* based on the result of an operation:

```
add r0, r1, r1    ; does not set condition flags
adds r0, r1, r1   ; sets condition flags
```

The "`s`" on "`adds`" sets the condition flags, which we can then look at.

Control Flow in Assembly

- In an ARM instruction, there is an S bit, and if it is a 1, then the condition flags will be set after the instruction.



- The condition codes that can be set are as follows (set = 1, clear = 0):

Z : set if result is 0, clear otherwise

N : set if result is < 0, clear otherwise (this is the sign flag)

C : set if result generated a carry

V : set if the result had arithmetic overflow

we'll cover carry and overflow later

But here is a great tutorial:
http://teaching.idallen.com/dat2343/10f/notes/040_overflow.txt

Control Flow in Assembly

Code	Suffix	Description	Flags
0000	EQ	Equal / equals zero	Z
0001	NE	Not equal	!Z
0010	CS / HS	Carry set / unsigned higher or same	C
0011	CC / LO	Carry clear / unsigned lower	!C
0100	MI	Minus / negative	N
0101	PL	Plus / positive or zero	!N
0110	VS	Overflow	V
0111	VC	No overflow	!V
1000	HI	Unsigned higher	C and !Z
1001	LS	Unsigned lower or same	!C or Z
1010	GE	Signed greater than or equal	N == V
1011	LT	Signed less than	N != V
1100	GT	Signed greater than	!Z and (N == V)
1101	LE	Signed less than or equal	Z or (N != V)
1110	AL	Always (default)	any

Branching is based on what the status codes are at a given moment. For example:

```
b target  // always branches
beq target // branches if
                zero flag is 1
bne target // branches if
                zero flag is 0
bge target  // read as "branch
                if greater than or
                equal";
                branches if
                N == V
```

Control Flow in Assembly

Code	Suffix	Description	Flags
0000	EQ	Equal / equals zero	Z
0001	NE	Not equal	!Z
0010	CS / HS	Carry set / unsigned higher or same	C
0011	CC / LO	Carry clear / unsigned lower	!C
0100	MI	Minus / negative	N
0101	PL	Plus / positive or zero	!N
0110	VS	Overflow	V
0111	VC	No overflow	!V
1000	HI	Unsigned higher	C and !Z
1001	LS	Unsigned lower or same	!C or Z
1010	GE	Signed greater than or equal	N == V
1011	LT	Signed less than	N != V
1100	GT	Signed greater than	!Z and (N == V)
1101	LE	Signed less than or equal	Z or (N != V)
1110	AL	Always (default)	any

Branches simply read the current condition codes, and take the branch if the condition is satisfied.

We also have two other instructions that simply set the condition flags:

```
cmp r0, r1 // performs
           r0 - r1 and
           sets the
           condition flags

tst r0, r1 // performs
           r0 AND r1 and
           sets the
           condition
           flags
```

Control Flow in Assembly

Code	Suffix	Description
0000	EQ	Equal / equals zero
0001	NE	Not equal
0010	CS / HS	Carry set / unsigned higher or same
0011	CC / LO	Carry clear / unsigned lower
0100	MI	Minus / negative
0101	PL	Plus / positive or zero
0110	VS	Overflow
0111	VC	No overflow
1000	HI	Unsigned higher
1001	LS	Unsigned lower or same
1010	GE	Signed greater than or equal
1011	LT	Signed less than
1100	GT	Signed greater than
1101	LE	Signed less than or equal
1110	AL	Always (default)

Examples: which branches will be taken?

```
        mov     r0, #1
        mov     r1, #2
        mov     r2, #3
        adds        r3, r0, r1
        bge     L1
        adds        r4, r0, r0
L1
        subs        r4, r1, r2 ; subs ra, rb,
                                ;rc puts rb - rc into ra
        bge     L2
        cmp     r0, r1
        beq     L3
L2
        cmp     r2, r3
        beq     L4
L3
        tst     r1, r2
        blt     L5
        add     r5, r4, r3
L4
        tst     r4, r4
        bmi     L6
L5
        add     r0, r1, r1
L6
```

Coding challenge: count the "on" bits in a number

Write assembly program to count the number of "on" bits in a given 8-bit number :

```
    mov   r0, #0x3e
    mov   r1, #0
    ;   count on bits in val in r0
    ;   put result in r1
```

https://salmanarif.bitbucket.io/visual/

Coding challenge: count the "on" bits in a number

Write assembly program to count the number of "on" bits in a given 8-bit number :

```
    mov   r0, #0x3e
    mov   r1, #0
    ;   count on bits in val in r0
    ;   put result in r1
    mov   r2, #1
loop:
    tst   r0, r2
    addne r1, r1, #1
    lsl   r2, r2, #1
    cmp   r2, #0x100
    blt   loop
```

Lots of possible solutions!

https://salmanarif.bitbucket.io/visual/

Coding challenge: count the "on" bits in a number

Write assembly program to count the number of "on" bits in a given 8-bit number :

```
    mov   r0, #0x3e
    mov   r1, #0
    ;   count on bits in val in r0
    ;   put result in r1
    mov   r2, #1
loop:
    tst   r0, r2
    beq   nope
    add   r1, r1, #1
nope:
    lsl   r2, r2, #1
    cmp   r2, #0x100
    blt   loop
```

Here's a different solution

https://salmanarif.bitbucket.io/visual/

Coding challenge: count the "on" bits in a number

Write assembly program to count the number of "on" bits in a given 8-bit number :

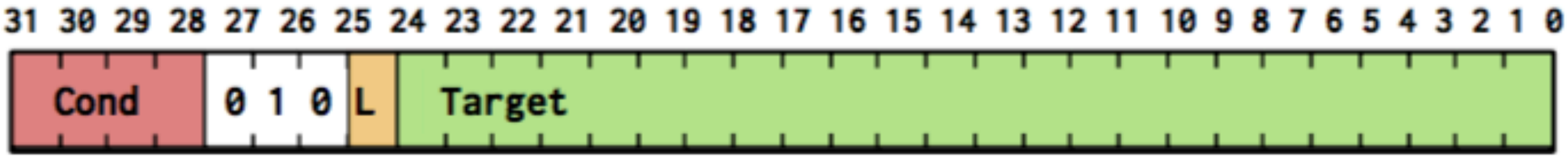
```
    mov   r0, #0x3e
    mov   r1, #0
    ;   count on bits in val in r0
    ;   put result in r1
loop:
    ands r2, r0, #1
    addne r1, r1, #1
    lsrs r0, r0, #1
    bne loop
```

Lots of possible solutions!

This one has fewer instructions!

https://salmanarif.bitbucket.io/visual/

Branch instruction encoding



b (bal) branch always

1110 1010 tttt tttt tttt tttt tttt tttt

beq branch if zero flag set

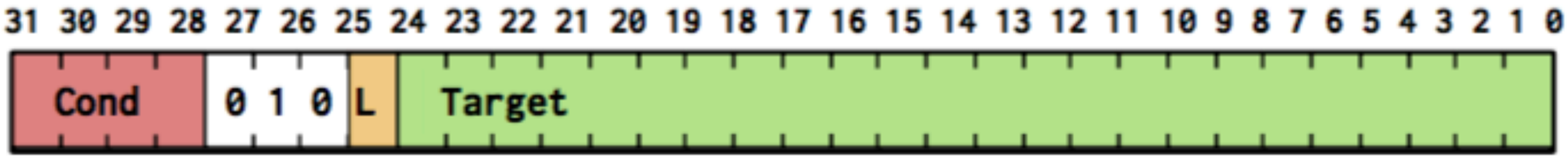
0000 1010 tttt tttt tttt tttt tttt tttt

The branch target is a *PC-relative* offset

The green above encode the offset, counted in 4-byte words.

Q: How far can this reach?

Branch instruction encoding



b (bal) branch always

1110 1010 tttt tttt tttt tttt tttt tttt

beq branch if zero flag set

0000 1010 tttt tttt tttt tttt tttt tttt

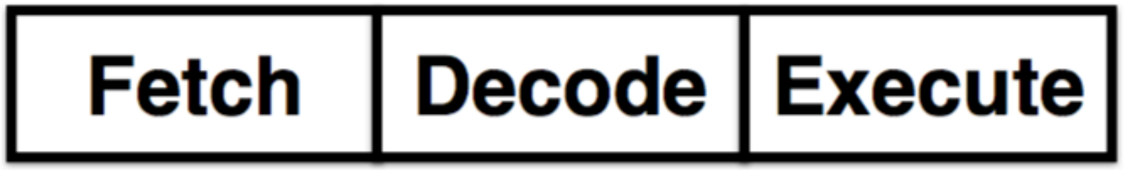
The branch target is a *PC-relative* offset

The green above encode the offset, counted in 4-byte words.

Q: How far can this reach? **24 bits = (1 << 24) = 16,777,216**

16,777,216 * 4 = 67,108,864. Half go forward, half backwards, so there is a range of 67,108,864/2 = 33,554,432 = ±32MB in either direction.

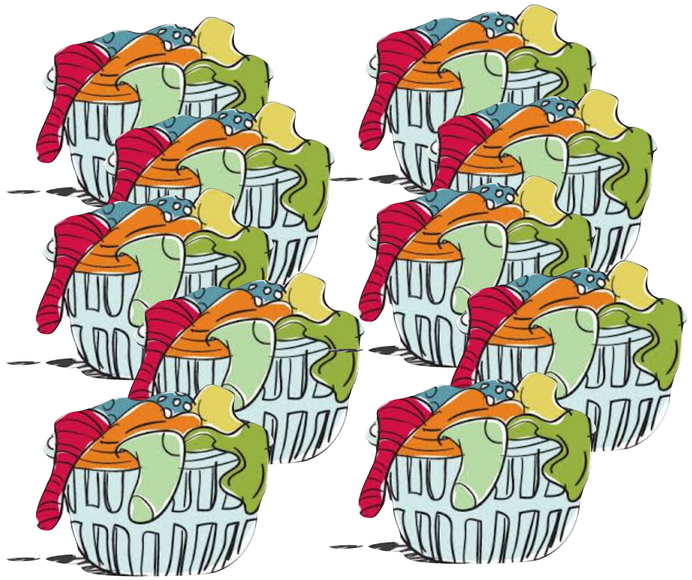
3 Steps Per Instruction

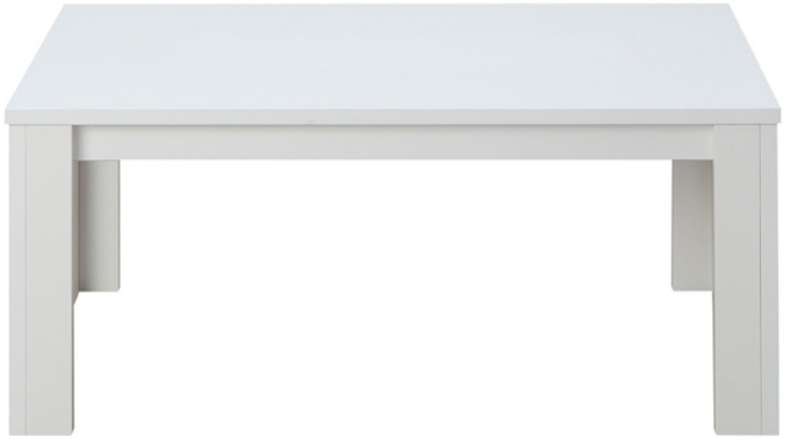


3 instructions takes 9 steps in sequence

Pipelining

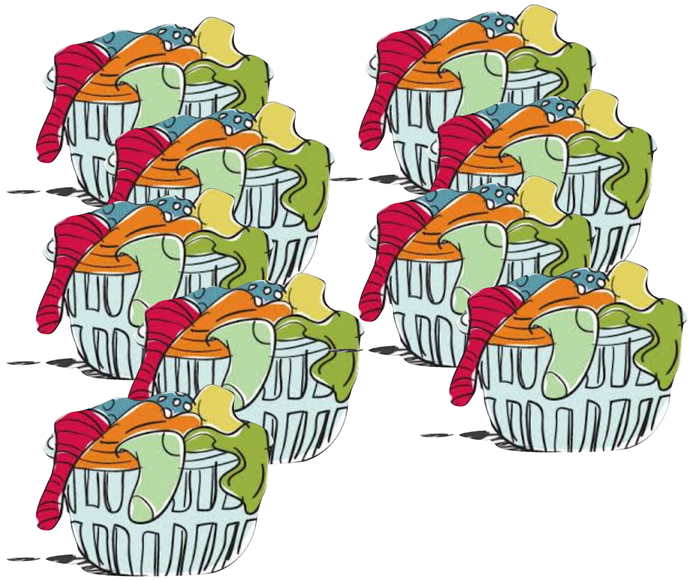
To speed things up, steps are overlapped, or *pipelined*. You can think of pipelining as doing laundry: let's say you have a washer and a dryer. You can do one wash initially, but then you can move the first load to the dryer and wash the next load *at the same time* as you are drying the first load. When those two finish, you can move the first load to the folding table, the second load to the drier, and the third load to the washer. Then you can continuously work on three loads at the same time, assuming they all take the same amount of time.

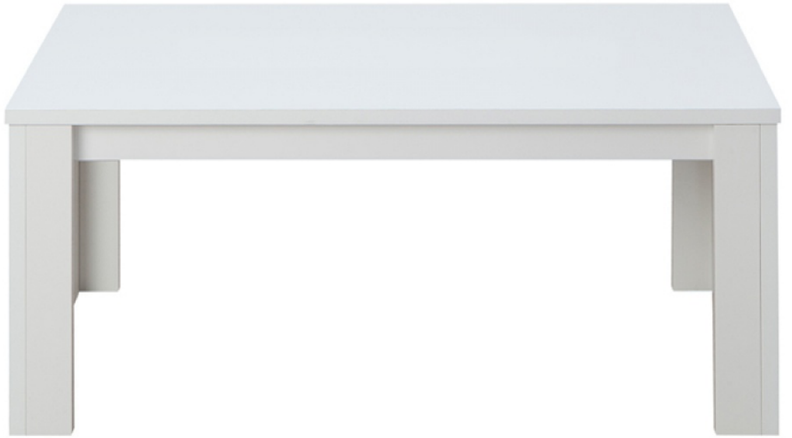




Pipelining

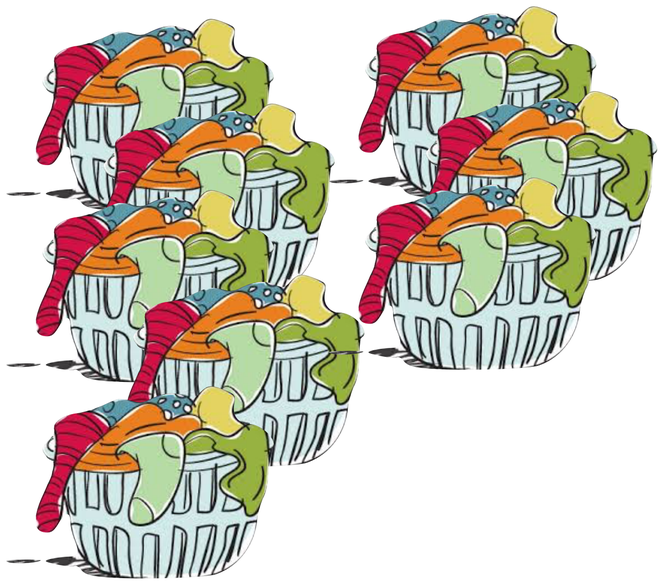
To speed things up, steps are overlapped, or *pipelined*. You can think of pipelining as doing laundry: let's say you have a washer and a dryer. You can do one wash initially, but then you can move the first load to the dryer and wash the next load *at the same time* as you are drying the first load. When those two finish, you can move the first load to the folding table, the second load to the drier, and the third load to the washer. Then you can continuously work on three loads at the same time, assuming they all take the same amount of time.

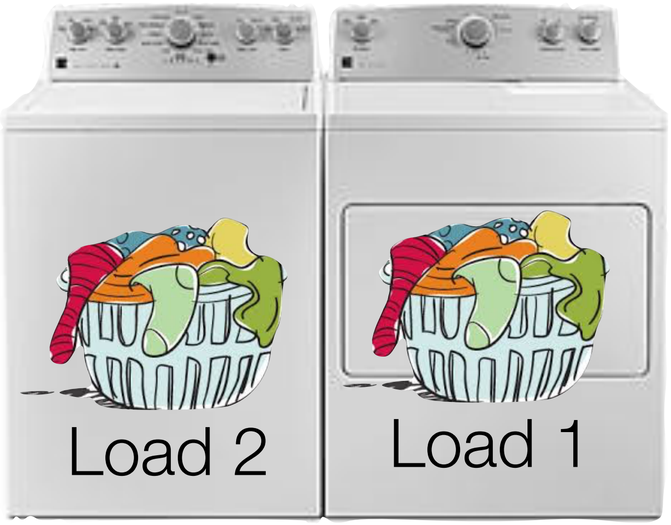


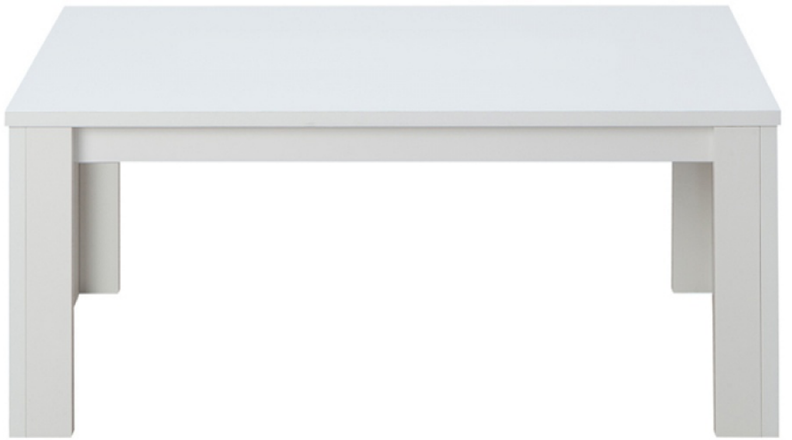


Pipelining

To speed things up, steps are overlapped, or *pipelined*. You can think of pipelining as doing laundry: let's say you have a washer and a dryer. You can do one wash initially, but then you can move the first load to the dryer and wash the next load *at the same time* as you are drying the first load. When those two finish, you can move the first load to the folding table, the second load to the drier, and the third load to the washer. Then you can continuously work on three loads at the same time, assuming they all take the same amount of time.

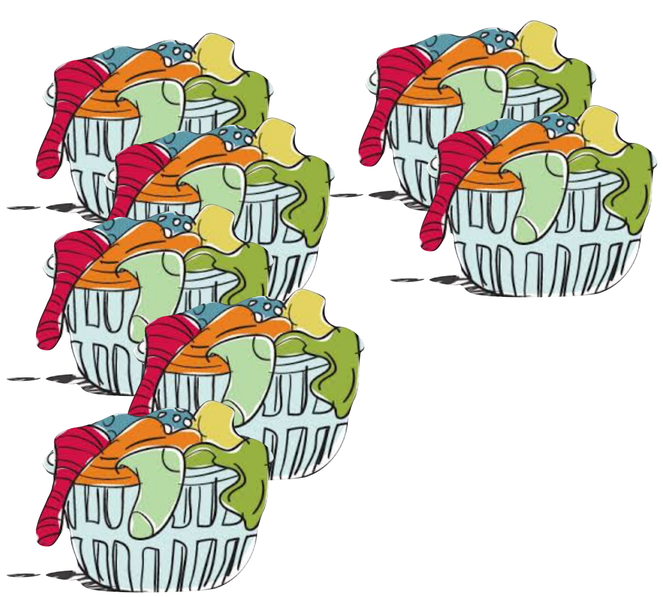


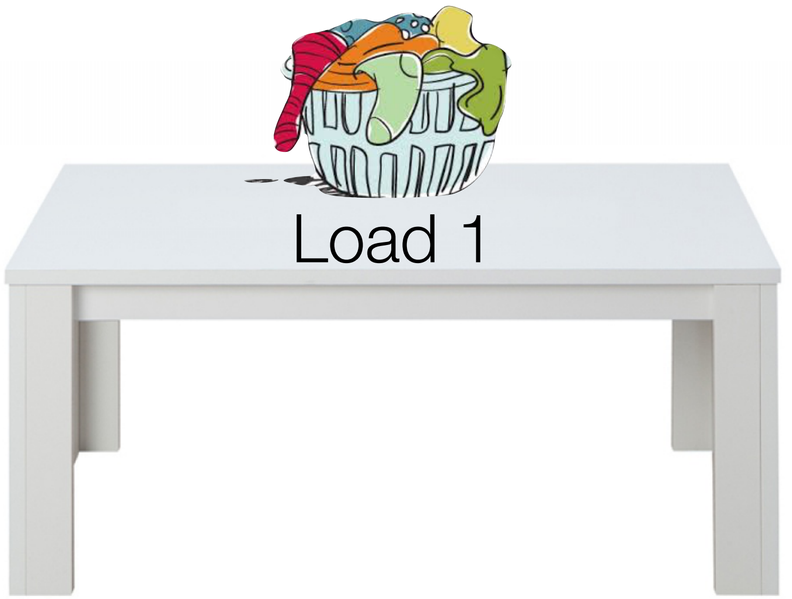




Pipelining

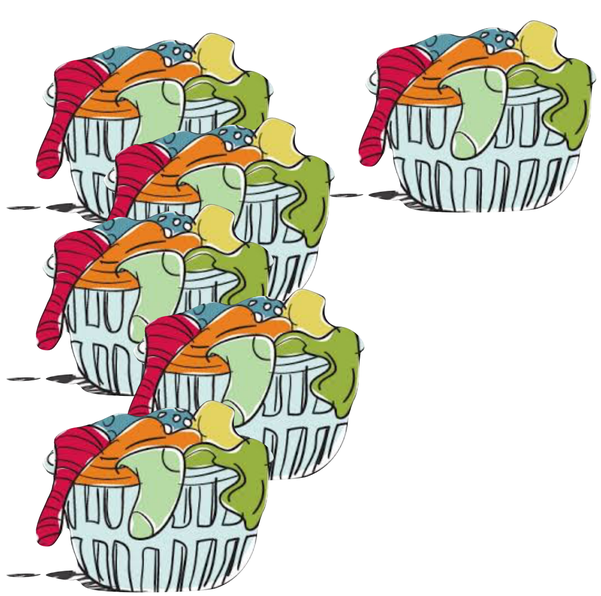
To speed things up, steps are overlapped, or *pipelined*. You can think of pipelining as doing laundry: let's say you have a washer and a dryer. You can do one wash initially, but then you can move the first load to the dryer and wash the next load *at the same time* as you are drying the first load. When those two finish, you can move the first load to the folding table, the second load to the drier, and the third load to the washer. Then you can continuously work on three loads at the same time, assuming they all take the same amount of time.

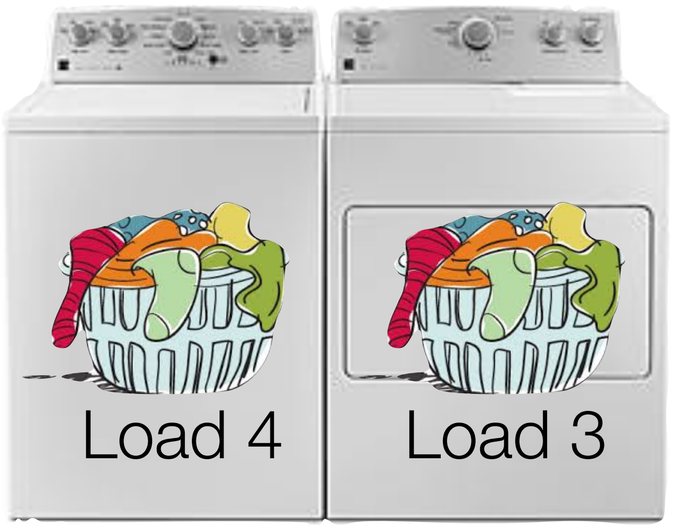


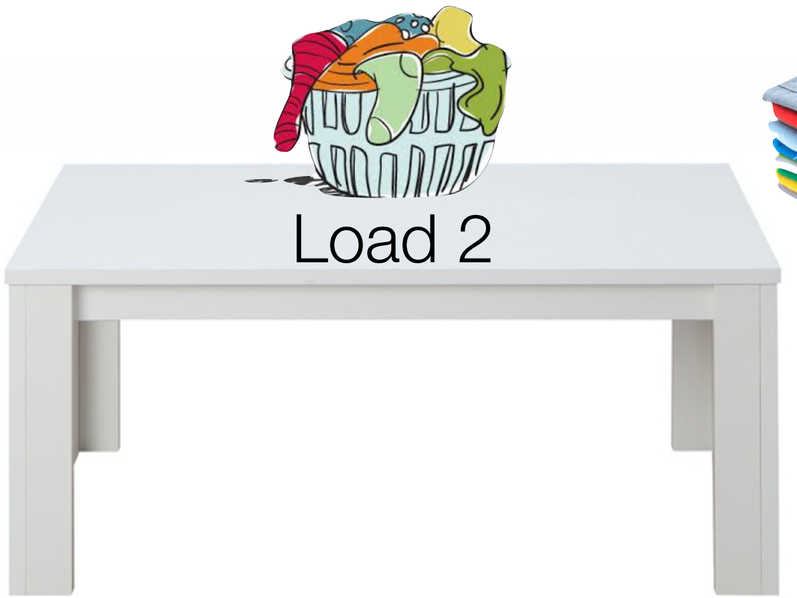


Pipelining

To speed things up, steps are overlapped, or *pipelined*. You can think of pipelining as doing laundry: let's say you have a washer and a dryer. You can do one wash initially, but then you can move the first load to the dryer and wash the next load *at the same time* as you are drying the first load. When those two finish, you can move the first load to the folding table, the second load to the drier, and the third load to the washer. Then you can continuously work on three loads at the same time, assuming they all take the same amount of time.

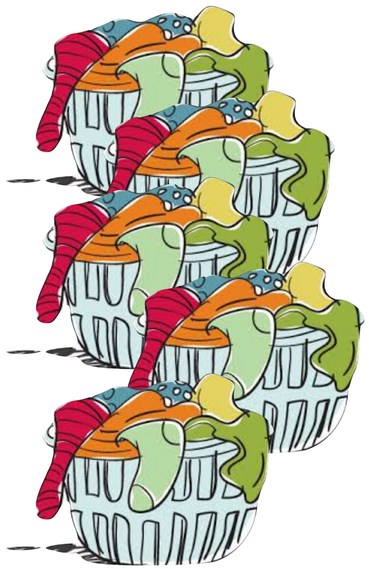


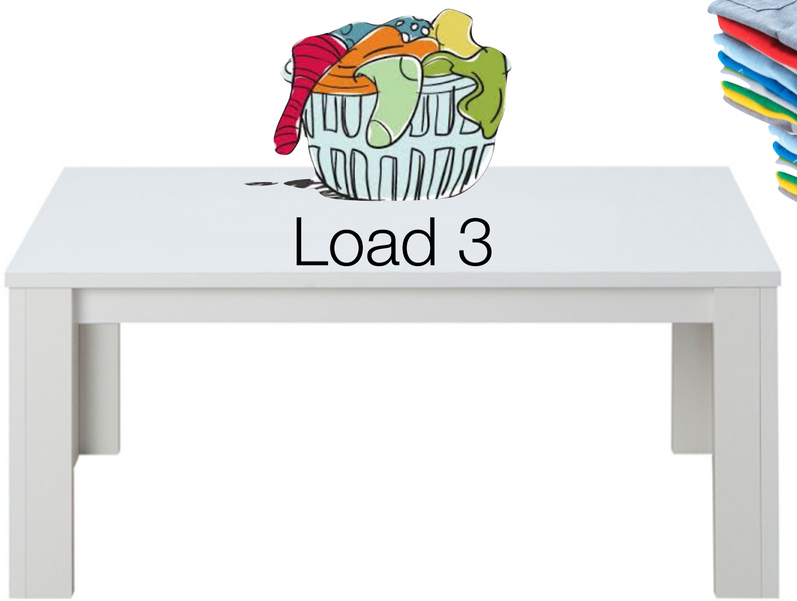




Pipelining

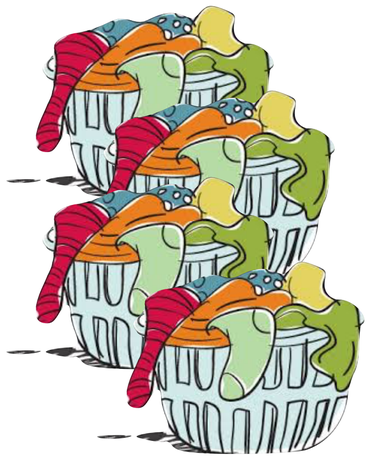
To speed things up, steps are overlapped, or *pipelined*. You can think of pipelining as doing laundry: let's say you have a washer and a dryer. You can do one wash initially, but then you can move the first load to the dryer and wash the next load *at the same time* as you are drying the first load. When those two finish, you can move the first load to the folding table, the second load to the drier, and the third load to the washer. Then you can continuously work on three loads at the same time, assuming they all take the same amount of time.

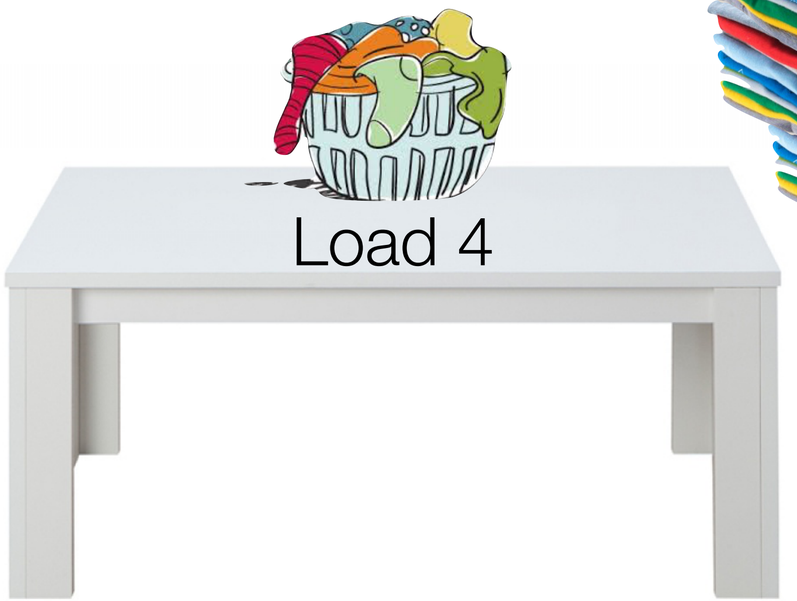




Pipelining

If each step takes 30 minutes, what is the overall *throughput* of a load of laundry? In other words, if you kept doing laundry, what would the average amount of time per load be?





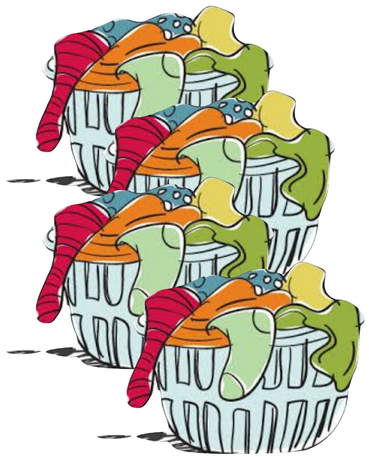
Loads 1,2,3 done

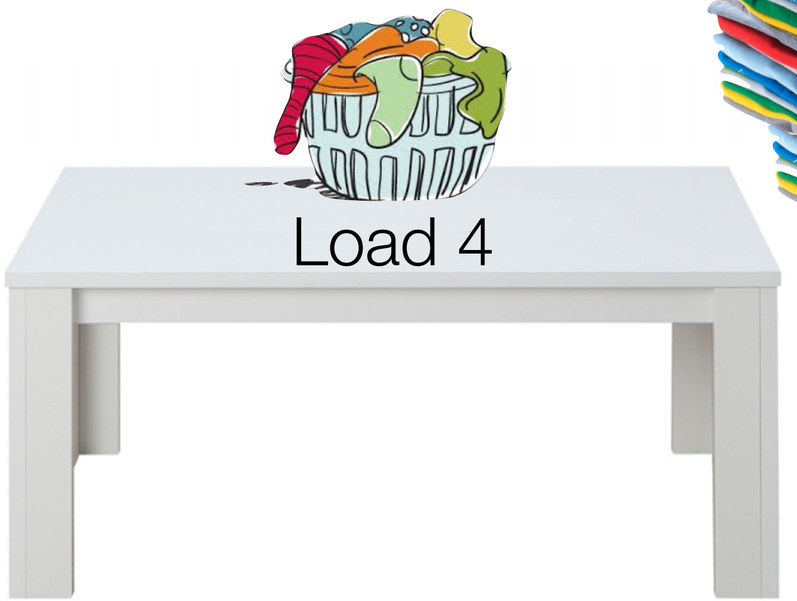
Pipelining

If each step takes 30 minutes, what is the overall *throughput* of a load of laundry? In other words, if you kept doing laundry, what would the average amount of time per load be?

Answer: 30 minutes per load (think banana-eating monkeys)

This is the general idea of pipelining.

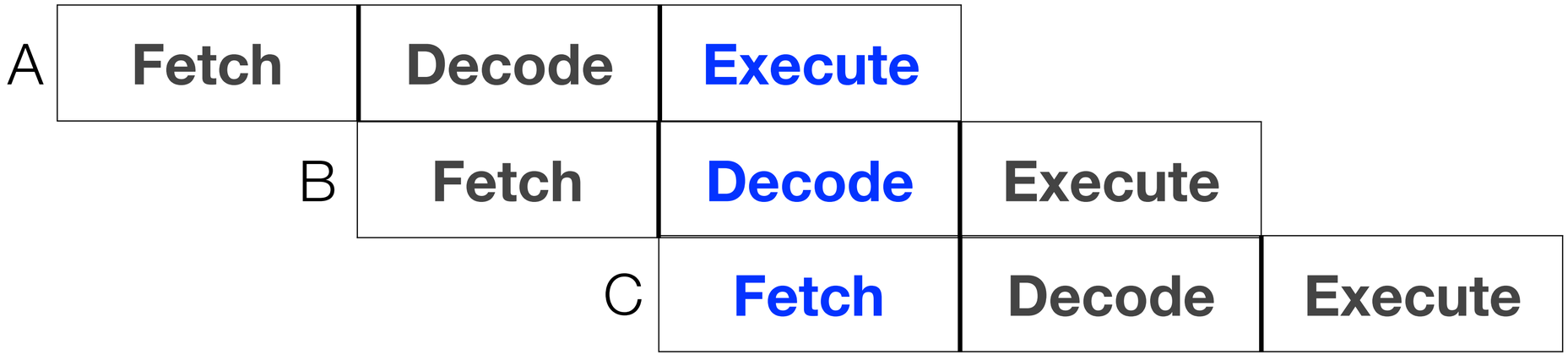




Loads 1,2,3 done

Pipelining

On an ARM processor, instructions are pipelined:



During the cycle that instruction A is executing, the PC has already advanced twice so it holds the address of the instruction being fetched (C). This is two instructions past A (PC + 8).

Pipelining

How does this manifest itself in a branch instruction?

```
$ arm-none-eabi-objdump -d branch_ex2.o
   0: e3a029ff   mov r2, #4177920 ; 0x3fc000

00000004 <loop>:
   4: e2522001   subs  r2, r2, #1
   8: 1affffffd   bne 4 <loop>
```

```
1a       branch if not equal opcode
ffffd    -3 in two's complement offset
         PC is already 8 ahead, or at 16. (offset * 4) is
         the number to add to the PC, so (-3 * 4) + 16
         = address 4 is the next instruction
```

Diversion #1: Why is this so complicated?

Okay, let's pause for a second. The previous slide seems crazy:

```
1a        branch if not equal opcode
fffffd    -3 in two's complement offset
          PC is already 8 ahead, or at 16. (offset * 4) is
          the number to add to the PC, so (-3 * 4) + 16
          = address 4 is the next instruction
```

Why is it so complicated? Design is all about choices, and all about fitting together the puzzle pieces so that things work. To really understand the details, you need to dig into the references, and you need to understand the pieces *from the ground up*. It turns out that the pipelining hardware design choices lead to the fact that the PC is already advanced by 8, for instance. This stuff is complicated because there are a ton of design choices that have to fit together. We want you to appreciate this, even though it can get confusing!

ISA design is an art form!

Some neat things about ARM design

Commonalities across operations

- Register vs. immediate operands
- Use of barrel shifter
- All registers treated same (with a few caveats)
- Predicated execution
- Set condition code (or not)

Diversion #2: Integer Representations

We will talk about two different ways to represent integers, *unsigned* and *signed*:

unsigned: can only represent non-negative numbers

signed: can represent negative, zero, and positive numbers

To be clear: The ARM processor does not care about the representation in binary -- a number is a number. But, we can interpret numbers as unsigned or signed.

Let's take a few minutes to look at some details about integers.

Unsigned Integers

So far, we have talked about converting from decimal to binary and vice-versa, which is a nice one-to-one relationship between the decimal number and its binary representation. Examples:

```
0b0001 = 1
0b0101 = 5
0b1011 = 11
0b1111 = 15
```

The range of an unsigned number is $0 \rightarrow 2^w - 1$, where w is the number of bits in our integer. For example, a 32-bit `int` can represent numbers from 0 to $2^{32} - 1$, or 0 to 4,294,967,295.

Signed Integers: How do we represent them?

What if we want to represent negative numbers? We have choices!

One way we could encode a negative number is simply to designate some bit as a "sign" bit, and then interpret the rest of the number as a regular binary number and then apply the sign. For instance, for a four-bit number:

0 001 = 1	1 001 = -1
0 010 = 2	1 010 = -2
0 011 = 3	1 011 = -3
0 100 = 4	1 100 = -4
0 101 = 5	1 101 = -5
0 110 = 6	1 110 = -6
0 111 = 7	1 111 = -7

This might be okay...but we've only represented 14 of our 16 available numbers...

Signed Integers: How do we represent them?

0 001 = 1	1 001 = -1
0 010 = 2	1 010 = -2
0 011 = 3	1 011 = -3
0 100 = 4	1 100 = -4
0 101 = 5	1 101 = -5
0 110 = 6	1 110 = -6
0 111 = 7	1 111 = -7

What about 0 000 and 1 000? What should they represent?

Well...this is a bit tricky!

Signed Integers: How do we represent them?

0 001 = 1	1 001 = -1
0 010 = 2	1 010 = -2
0 011 = 3	1 011 = -3
0 100 = 4	1 100 = -4
0 101 = 5	1 101 = -5
0 110 = 6	1 110 = -6
0 111 = 7	1 111 = -7

What about 0 000 and 1 000? What should they represent?

Well...this is a bit tricky!

Let's look at the bit patterns: 0 000 1 000

Should we make the 0 000 just represent decimal 0? What about 1 000? We could make it 0 as well, or maybe -8, or maybe even 8, but none of the choices are nice.

Signed Integers: How do we represent them?

0 001 = 1	1 001 = -1
0 010 = 2	1 010 = -2
0 011 = 3	1 011 = -3
0 100 = 4	1 100 = -4
0 101 = 5	1 101 = -5
0 110 = 6	1 110 = -6
0 111 = 7	1 111 = -7

What about 0 000 and 1 000? What should they represent?

Well...this is a bit tricky!

Let's look at the bit patterns: 0 000 1 000

Should we make the 0 000 just represent decimal 0? What about 1 000? We could make it 0 as well, or maybe -8, or maybe even 8, but none of the choices are nice.

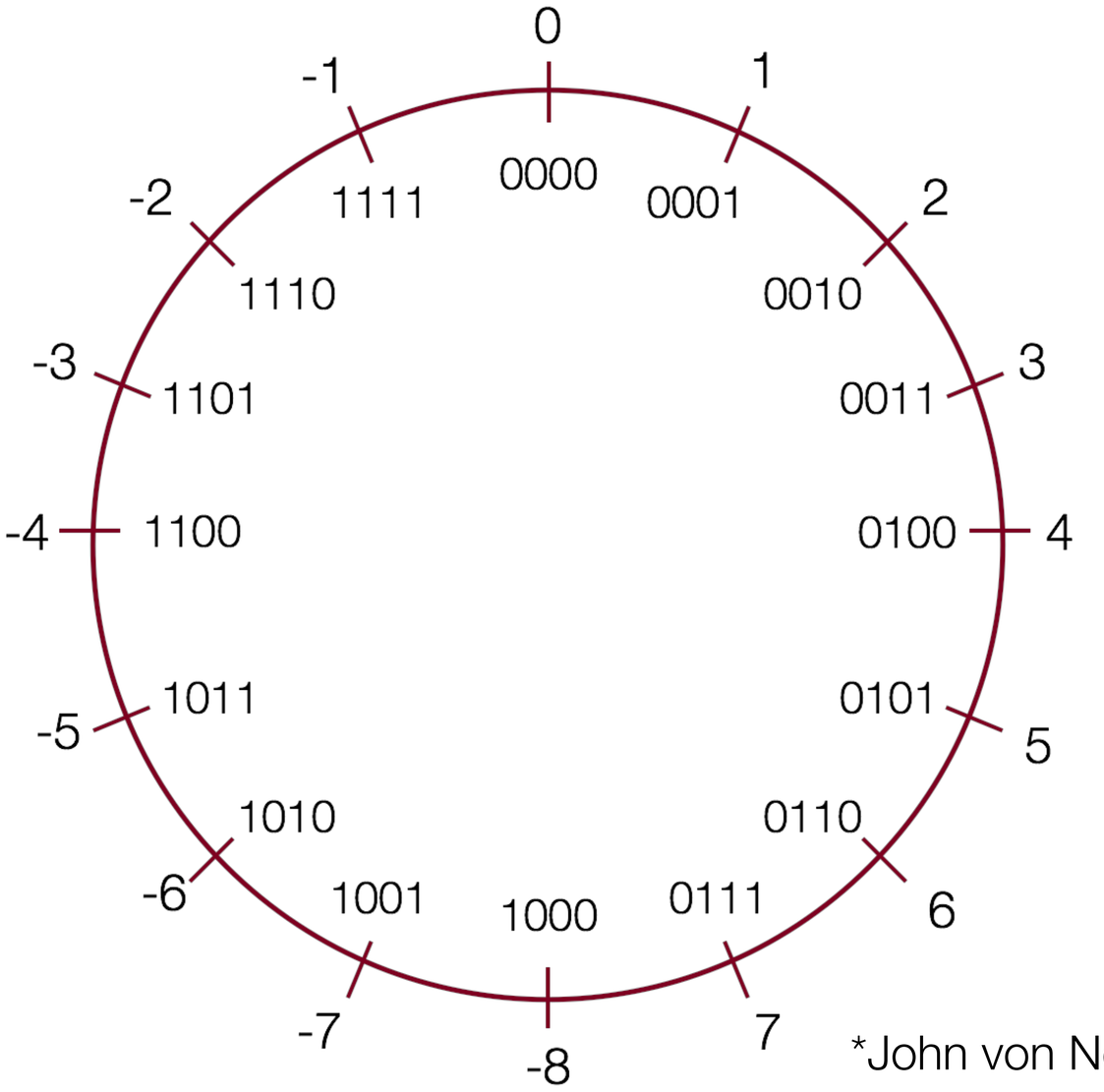
Fine. Let's just make 0 000 to be equal to decimal 0. How does arithmetic work? Well…to add two numbers, you need to know the sign, then you might have to subtract (borrow and carry, etc.), and the sign might change…this is going to get ugly!

Signed Integers: How do we represent them?

There is a better way!

Signed Integers: How do we represent them?

Behold: the "two's complement" circle:



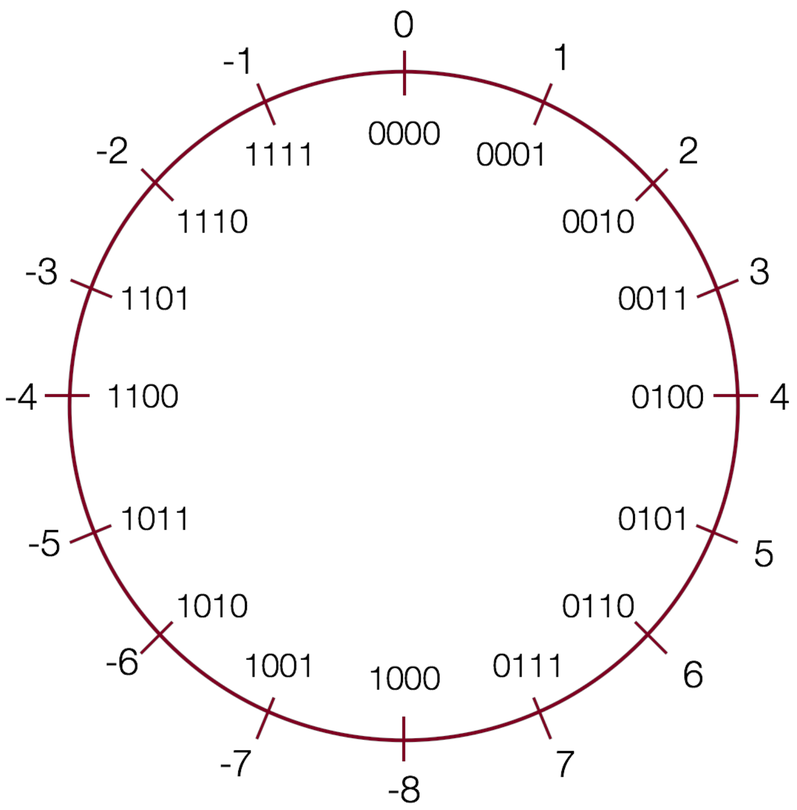
In the early days of computing*, two's complement was determined to be an excellent way to store binary numbers.

In two's complement notation, positive numbers are represented as themselves (phew), and negative numbers are represented as the *two's complement* of themselves (definition to follow).

This leads to some amazing arithmetic properties!

*John von Neumann suggested it in 1945, for the EDVAC computer.

Two's Complement



A two's-complement number system encodes positive and negative numbers in a binary number representation. The weight of each bit is a power of two, except for the most significant bit, whose weight is the negative of the corresponding power of two.

Definition: For vector $\vec{x} = [x_{w-1}, x_{w-2}, \ldots, x_0]$ of an w-bit integer $x_{w-1}x_{w-2}\ldots x_0$ is given by the following formula:

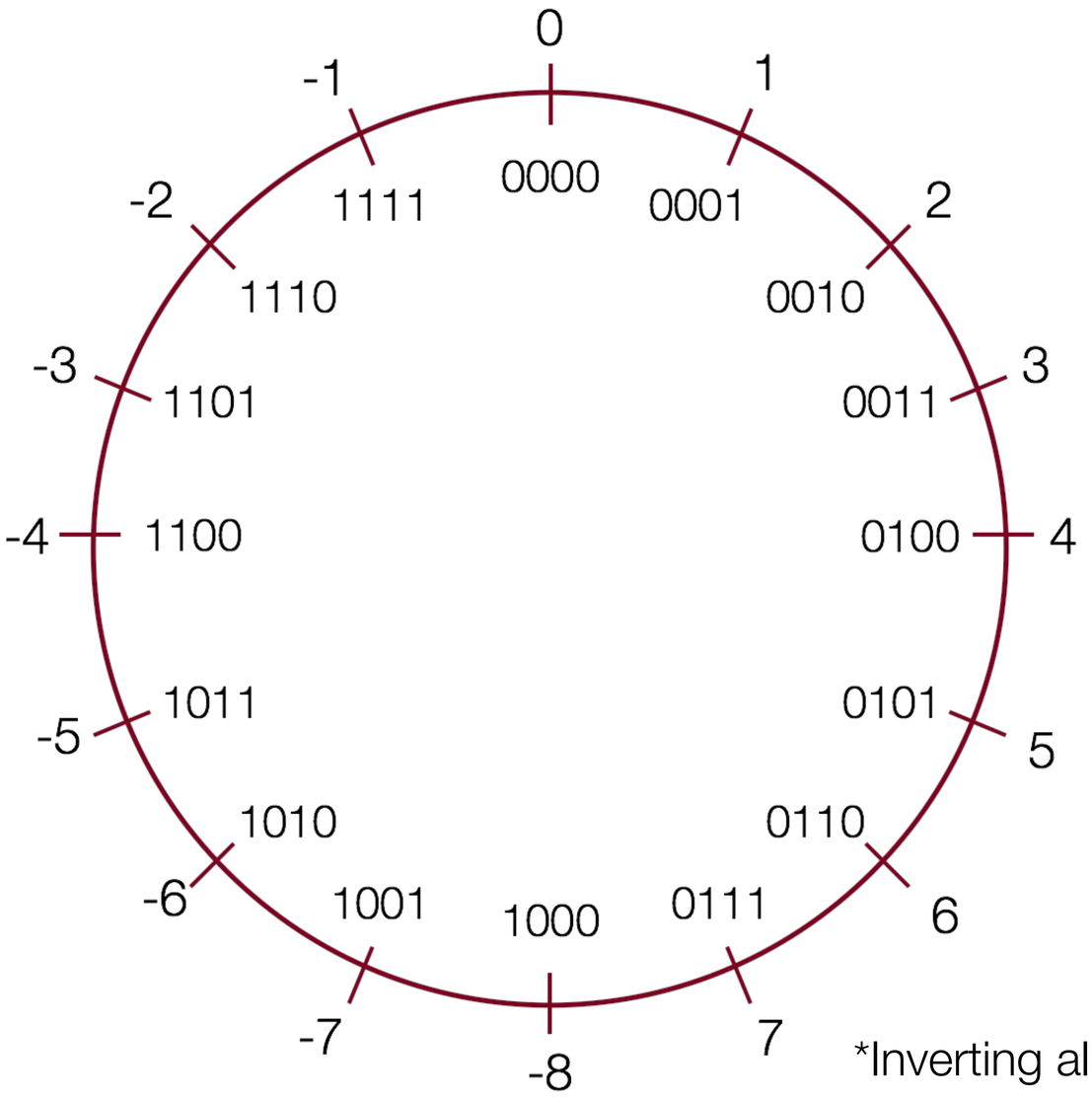
$$B2T_w(\vec{x}) = -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i.$$

$B2T_w$ means "Binary to Two's complement function"

In practice, a negative number in two's complement is obtained by inverting all the bits of its positive counterpart*, and then adding 1.

*Inverting all the bits of a number is its "one's complement"

Two's Complement



In practice, a negative number in two's complement is obtained by inverting all the bits of its positive counterpart*, and then adding 1, or: `x = ~x + 1`

Example: The number 2 is represented as normal in binary: 0010

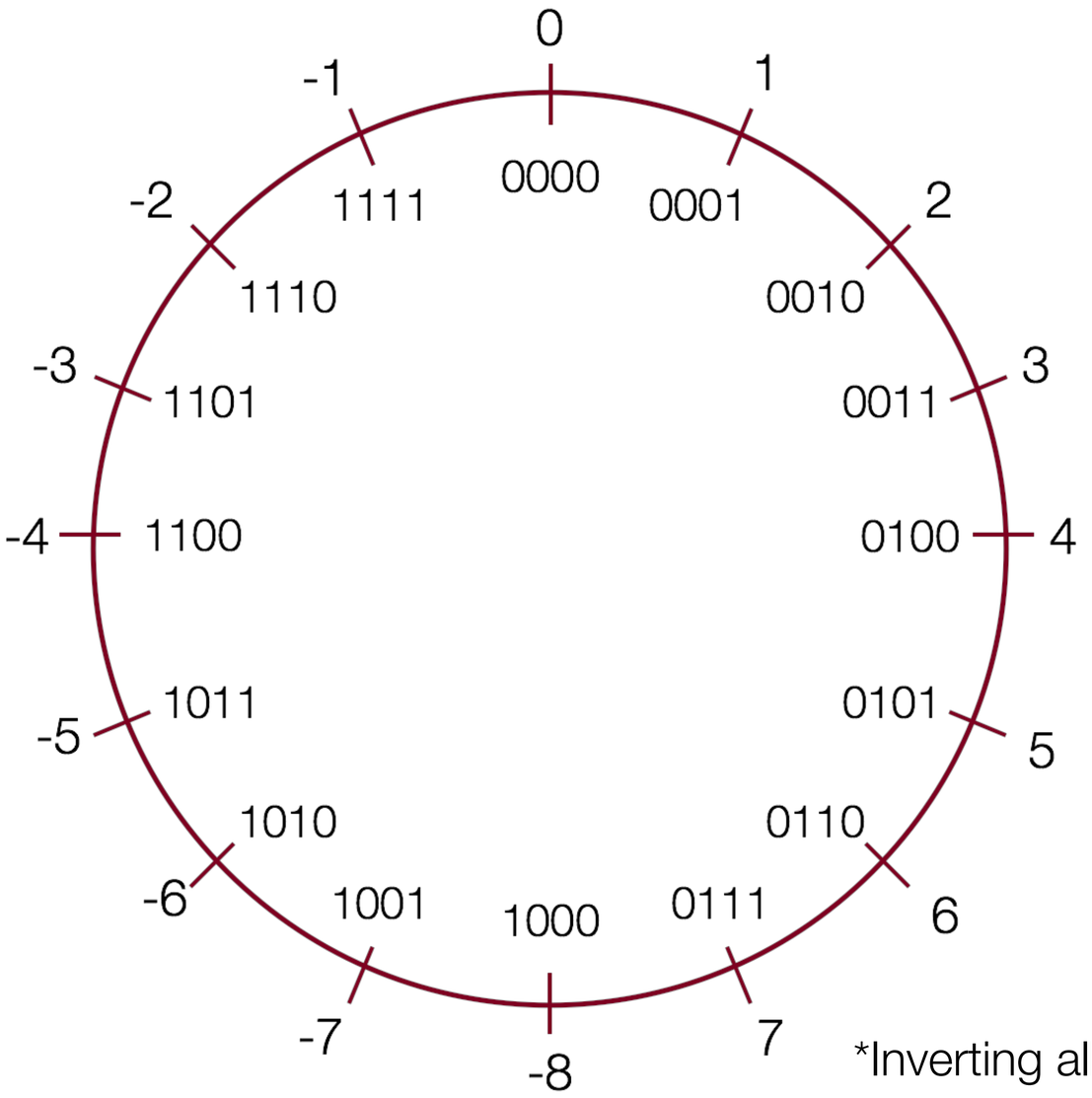
-2 is represented by inverting the bits, and adding 1:

0010 ☞ 1101

```
 1101
+   1
------
 1110
```

*Inverting all the bits of a number is its "one's complement"

Two's Complement



Trick: to convert a positive number to its negative in two's complement, start from the right of the number, and write down all the digits until you get to a 1. Then invert the rest of the digits:

Example: The number 2 is represented as normal in binary: 0010

Going from the right, write down numbers until you get to a 1:

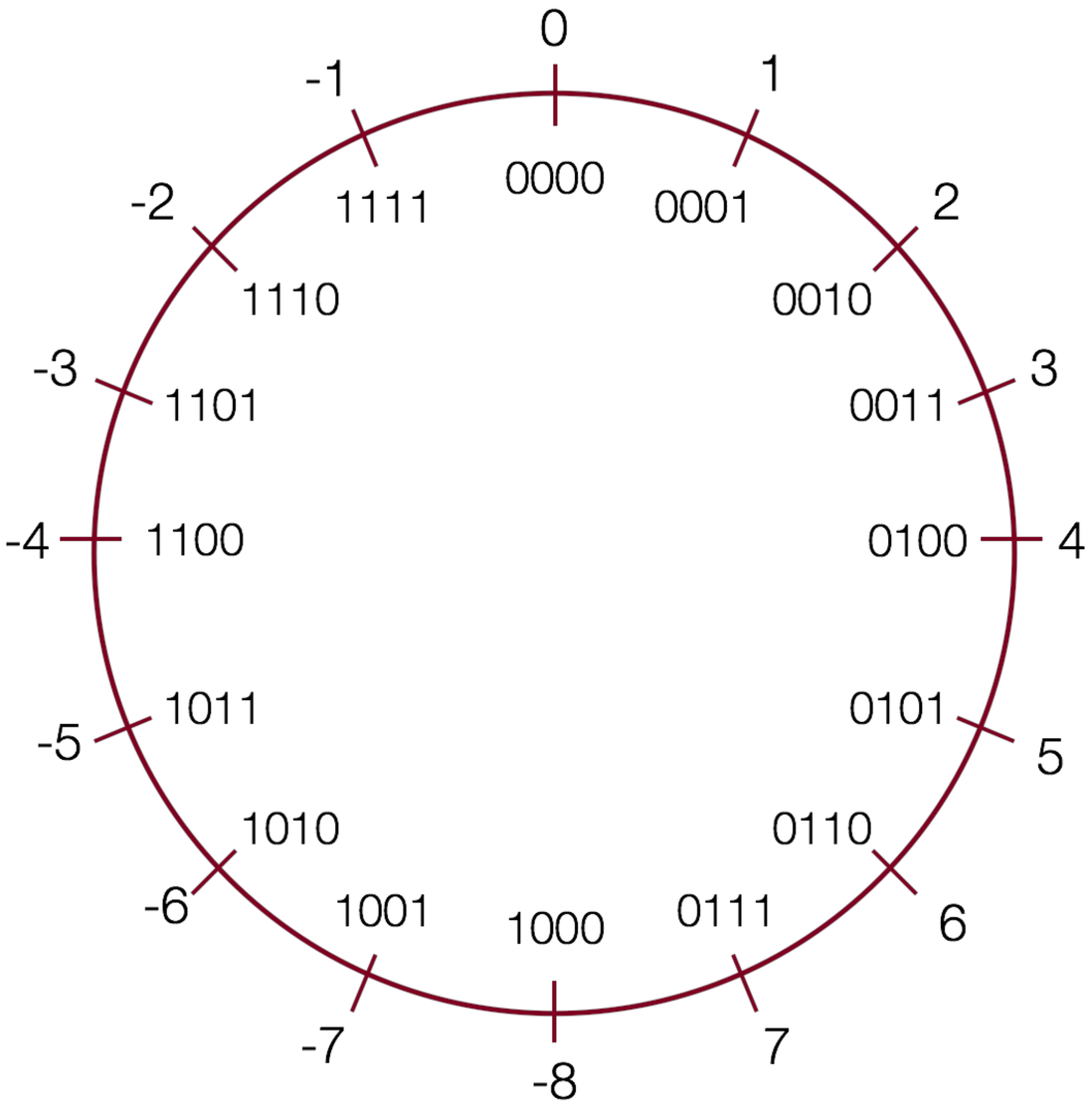
10

Then invert the rest of the digits:

1110

*Inverting all the bits of a number is its "one's complement"

Two's Complement



To convert a negative number to a positive number, perform the same steps!

Example: The number -5 is represented in two's complements as: 1011

5 is represented by inverting the bits, and adding 1:

1011 ☞ 0100

$$\begin{array}{r} 0100 \\ +\quad 1 \\ \hline 0101 \end{array}$$

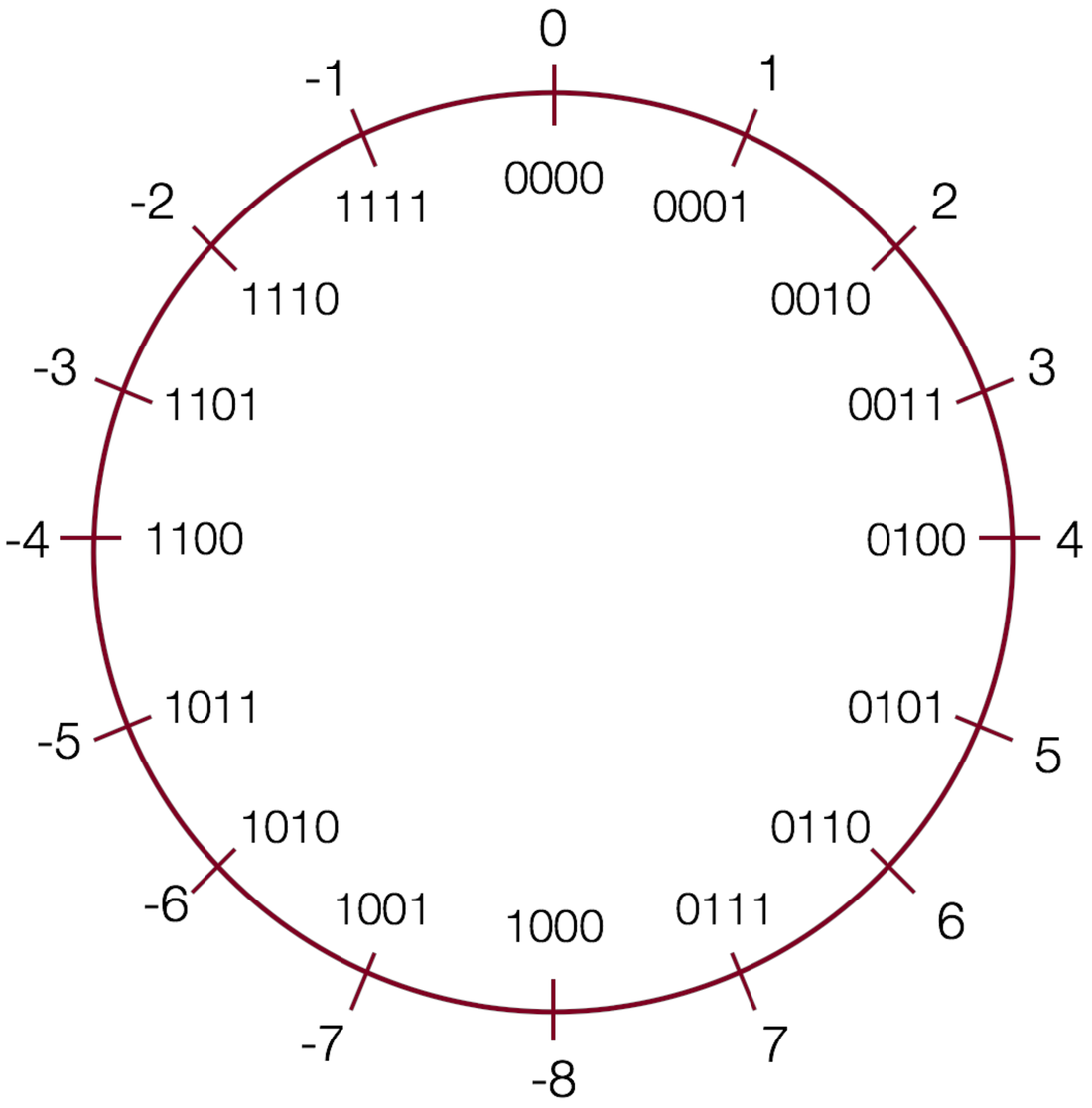
Shortcut: start from the right, and write down numbers until you get to a 1:

1

Now invert all the rest of the digits:

0101

Two's Complement: Neat Properties



There are a number of useful properties associated with two's complement numbers:

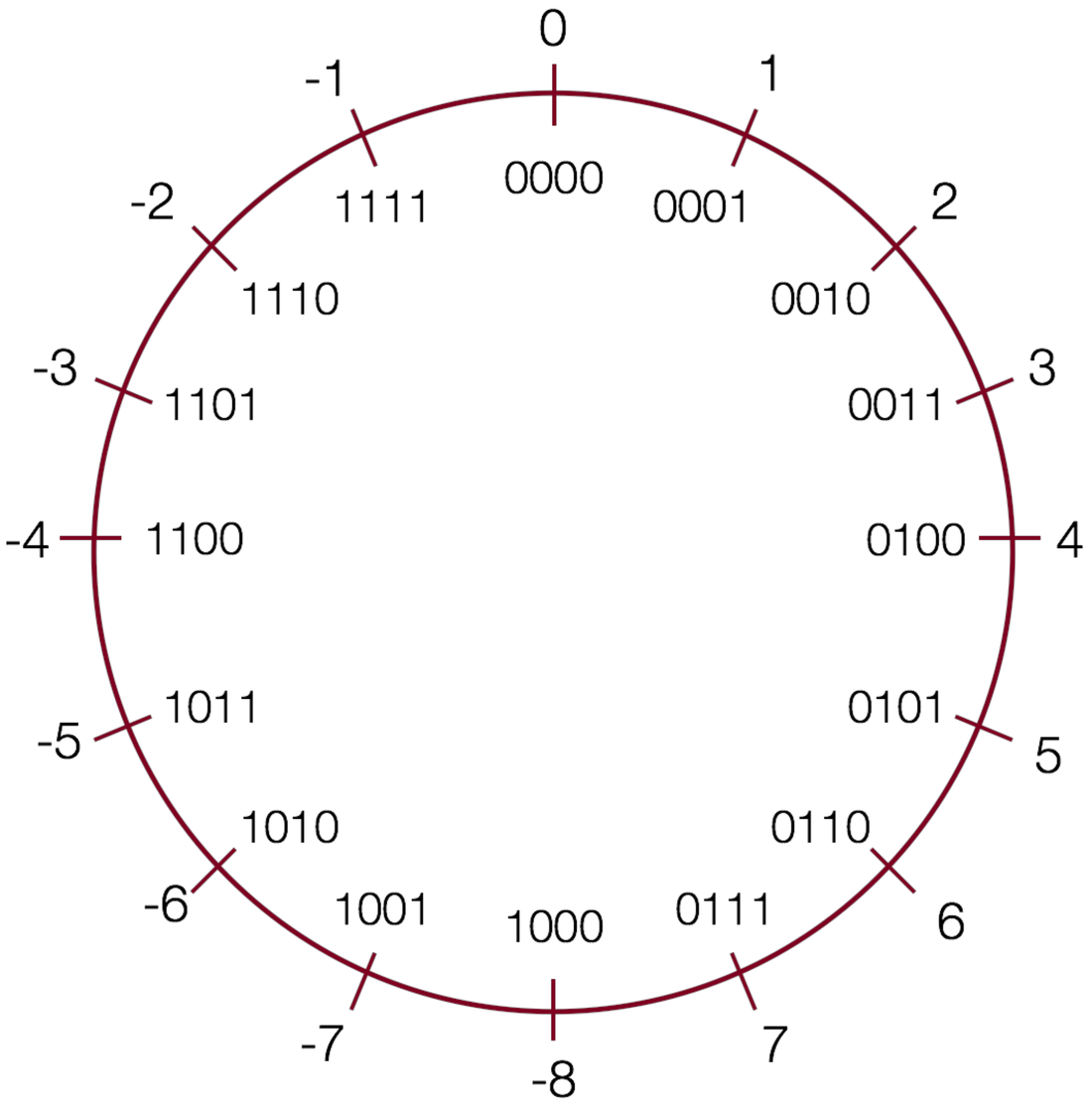
1. There is only one zero (yay!)
2. The highest order bit (left-most) is 1 for negative, 0 for positive (so it is easy to tell if a number is negative)
3. Adding two numbers is just…adding! Example:
   2 + -5 = -3

0010 ☞ 2

+1011 ☞ -5

1101 ☞ -3 decimal (wow!)

Two's Complement: Neat Properties



More useful properties:

4. Subtracting two numbers is simply performing the two's complement on one of them and then adding.
   Example:
   4 - 5 = -1

0100 ☞ 4, 0101 ☞ 5

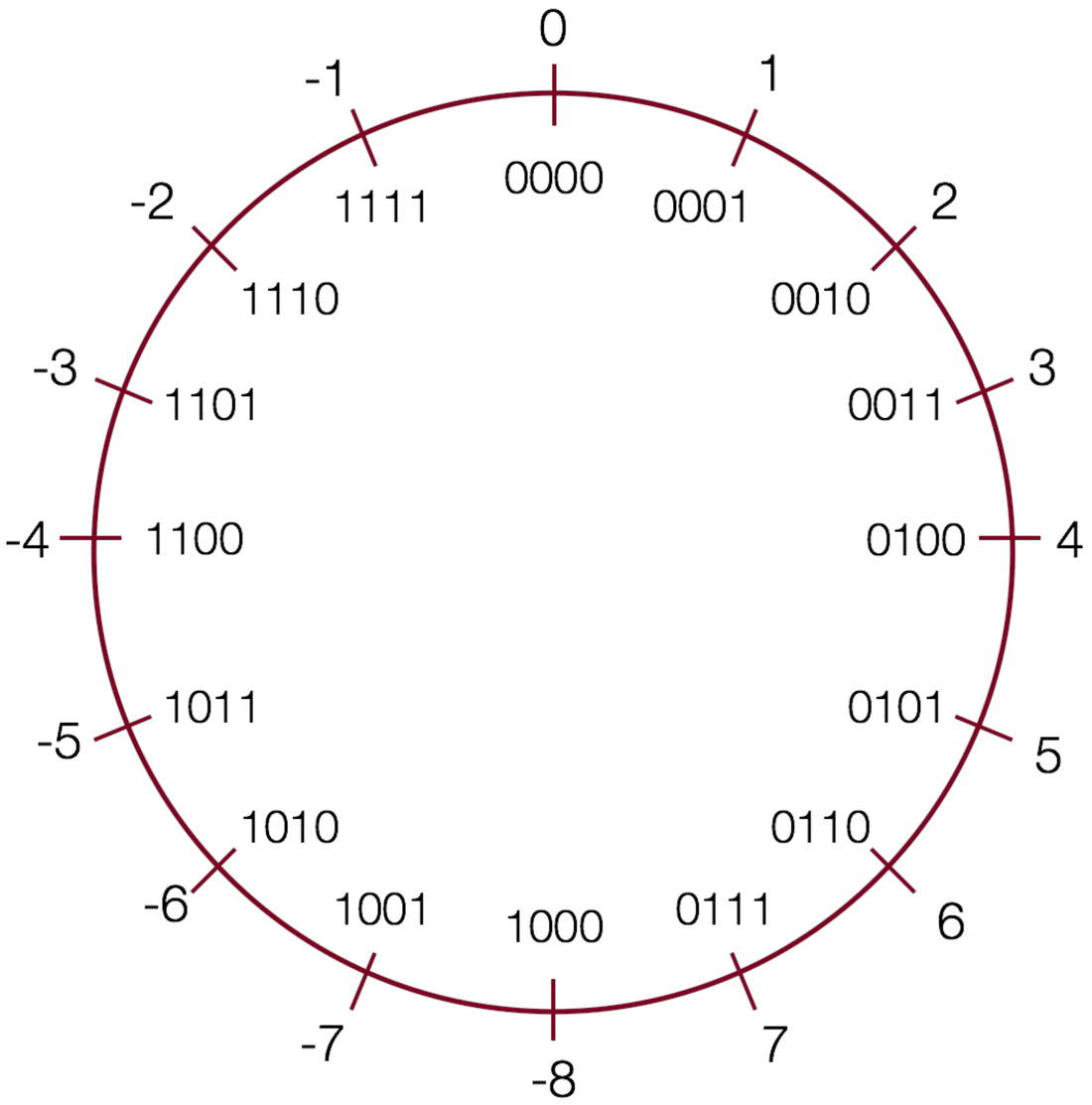
Find the two's complement of 5: 1011
add:

0100 ☞ 4

<u>+1011</u> ☞ -5

1111 ☞ -1 decimal

Two's Complement: Neat Properties



More useful properties:

5. Multiplication of two's complement works just by multiplying (throw away overflow digits).

Example: -2 * -3 = 6

1110 ☞ -2

x1101 ☞ -3

1110

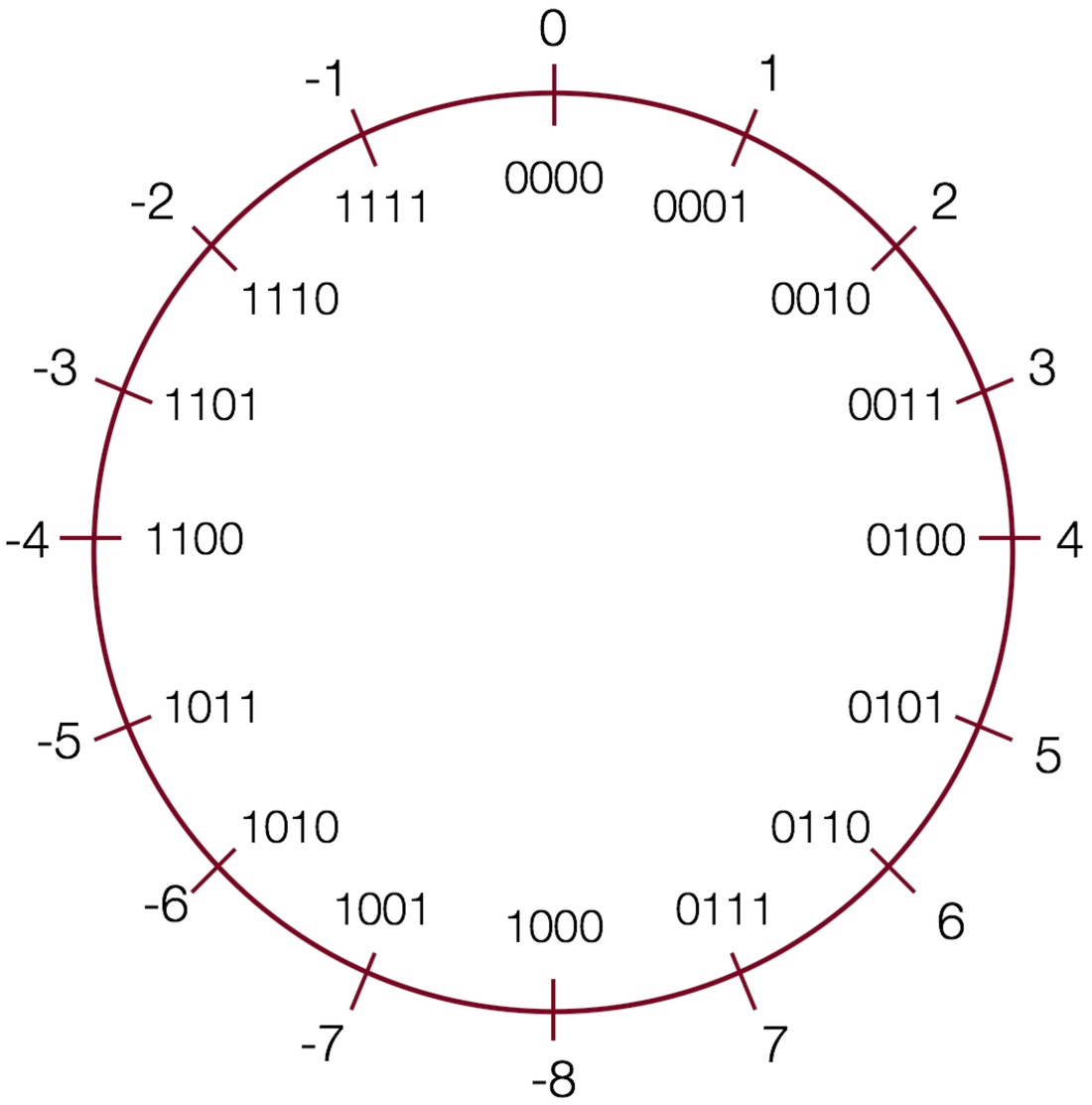
0000

1110

+1110

~~1011~~**0110** ☞ 6

Practice



Convert the following 4-bit numbers from positive to negative, or from negative to positive using two's complement notation:

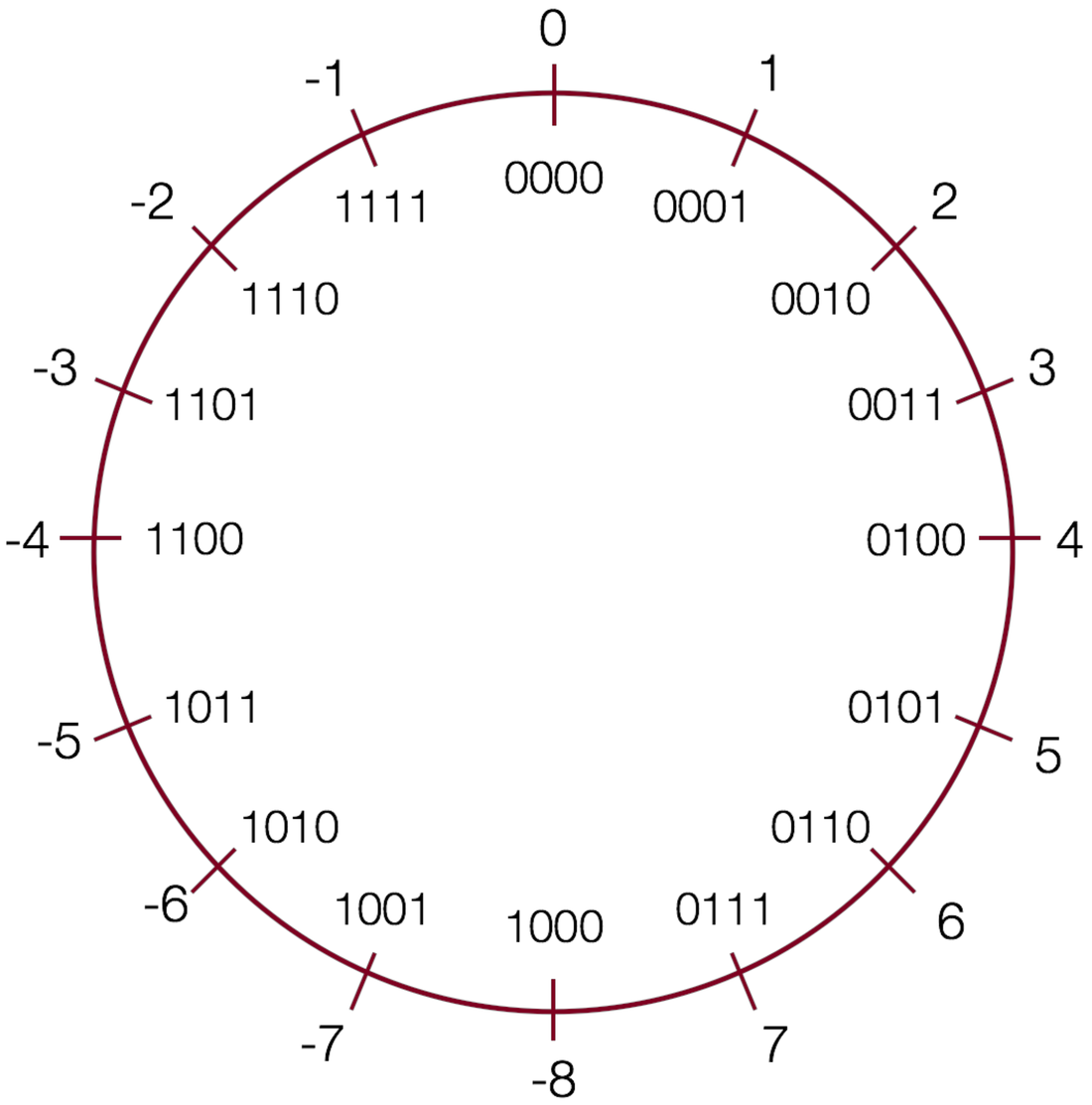
a. -4 (1100) ☞

b. 7 (0111) ☞

c. 3 (0011) ☞

d. -8 (1000) ☞

Practice



Convert the following 4-bit numbers from positive to negative, or from negative to positive using two's complement notation:

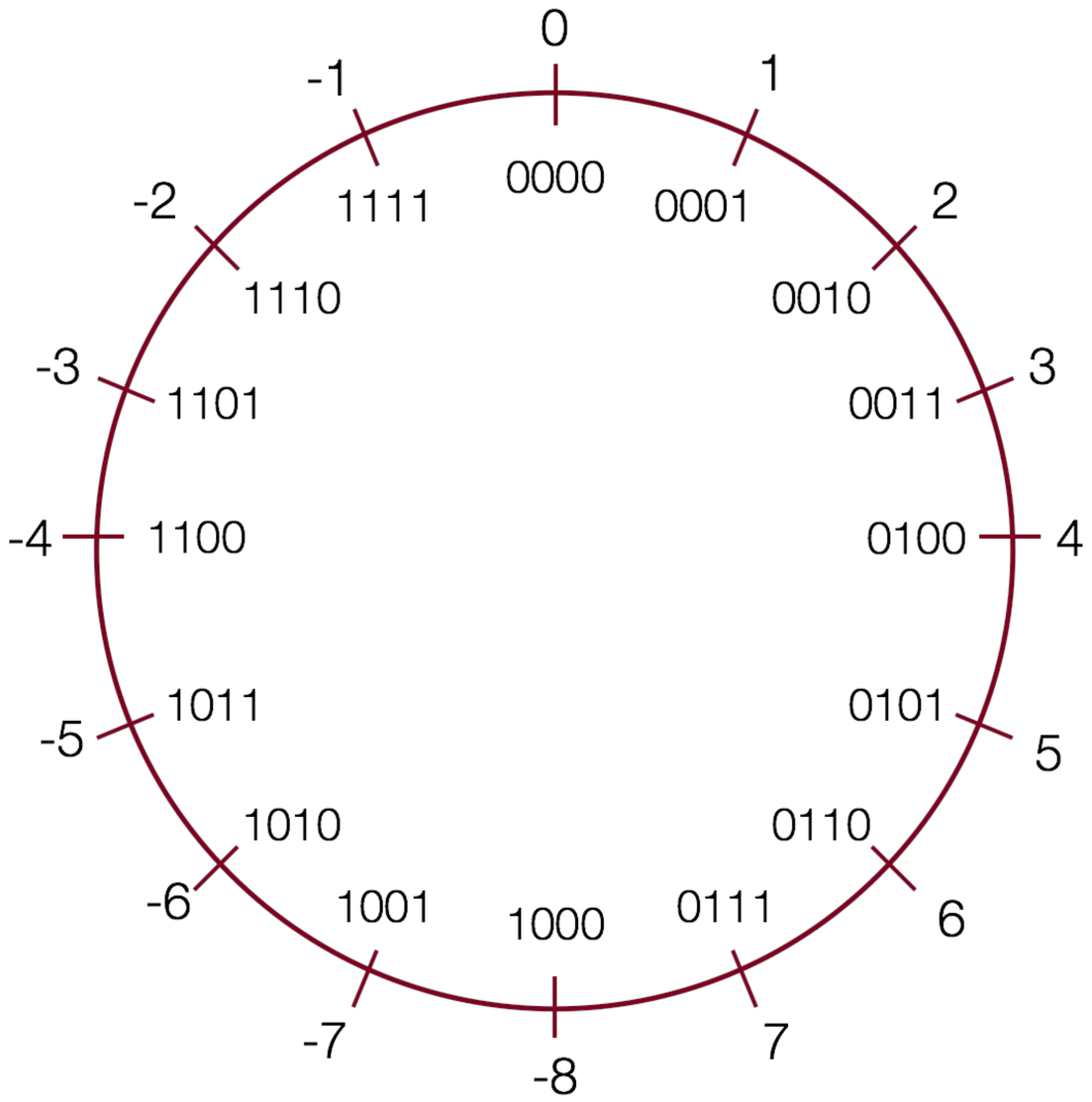
a. -4 (1100) ☞ 0100

b. 7 (0111) ☞ 1001

c. 3 (0011) ☞ 1101

d. -8 (1000) ☞ 1000 (?! If you look at the chart, +8 cannot be represented in two's complement with 4 bits!)

Practice



Convert the following 8-bit numbers from positive to negative, or from negative to positive using two's complement notation:

a. -4 (11111100) ☞ 00000100

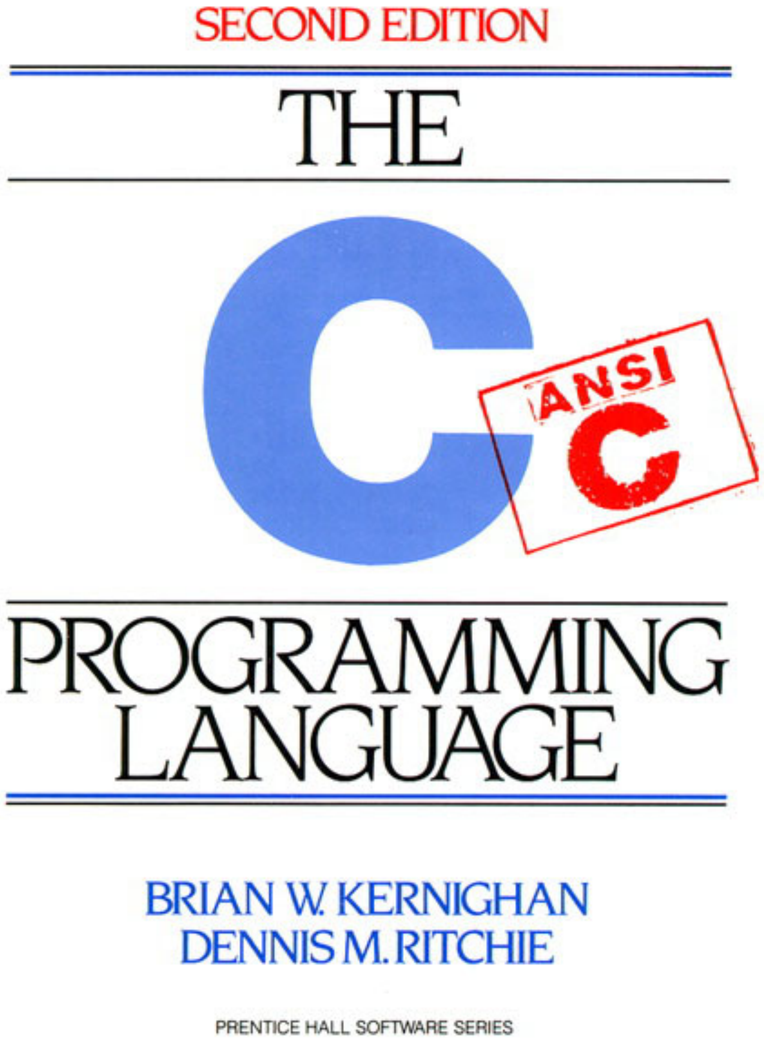
b. 27 (00011011) ☞ 11100101

c. -127 (10000001) ☞ 01111111

d. 1 (00000001) ☞ 11111111

C

Dennis Ritchie



C

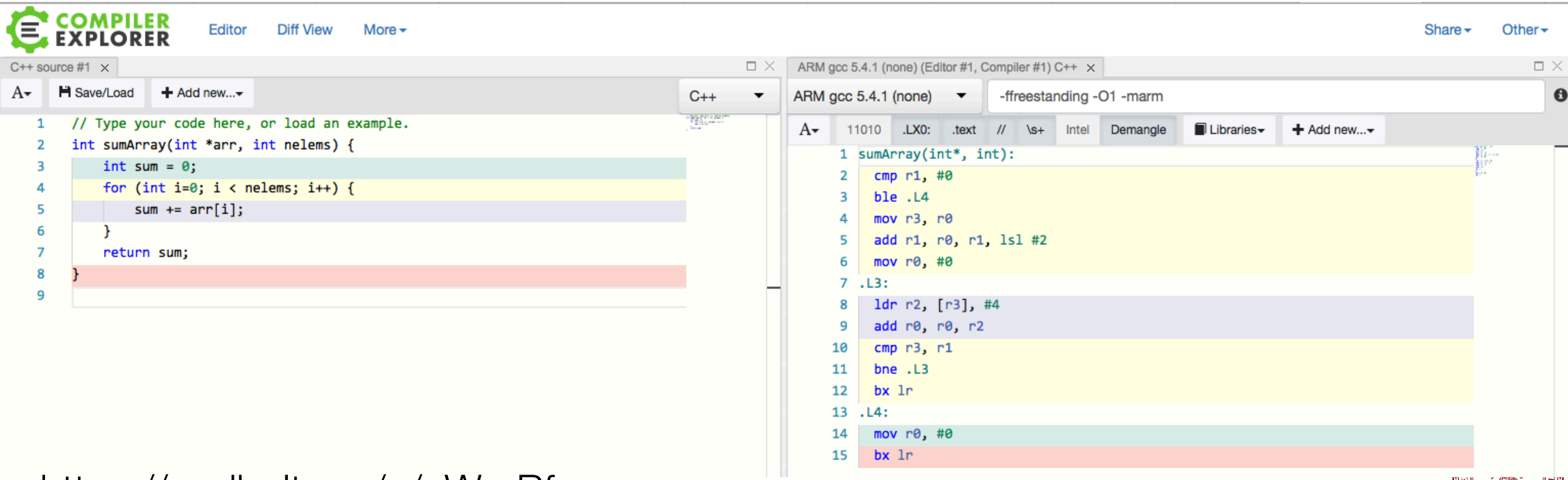
Ken Thompson built UNIX using C

The C programming language, invented in 1972 (!) is a high level language that makes it easy to work at a low level. It allows open access to the underlying hardware, yet it is still generally portable across machines, and compilers have been built for virtually every computer going back to the 1970s.

C

C language features closely model the ISA of many machines: data types, arithmetic / logical operators, control flow, access to memory, etc.

The *Compiler Explorer* is a cool interactive tool to see the translation from C to assembly:



https://godbolt.org/g/oWrqRf

Let's convert some assembly to C!

```
.equ DELAY, 0x3F0000

    ldr r0, FSEL2
    mov r1, #1
    str r1, [r0]
    mov r1, #(1<<20)

loop:
    ldr r0, SET0
    str r1, [r0]

    mov r2, #DELAY
    wait1:
        subs r2, #1
        bne wait1

    ldr r0, CLR0
    str r1, [r0]

    mov r2, #DELAY
    wait2:
        subs r2, #1
        bne wait2

    b loop

FSEL2: .word 0x20200008
SET0:  .word 0x2020001C
CLR0:  .word 0x20200028
```

Let's convert some assembly to C!

```
.equ DELAY, 0x3F0000

    ldr r0, FSEL2
    mov r1, #1
    str r1, [r0]
    mov r1, #(1<<20)

loop:
    ldr r0, SET0
    str r1, [r0]

    mov r2, #DELAY
    wait1:
        subs r2, #1
        bne wait1

    ldr r0, CLR0
    str r1, [r0]

    mov r2, #DELAY
    wait2:
        subs r2, #1
        bne wait2

    b loop

FSEL2: .word 0x20200008
SET0:  .word 0x2020001C
CLR0:  .word 0x20200028
```

```
unsigned int *FSEL2 =  (unsigned int *)0x20200008;
unsigned int *SET0  =  (unsigned int *)0x2020001c;
unsigned int *CLR0  =  (unsigned int *)0x20200028;

#define DELAY 0x3f0000

void main(void)
{
   *FSEL2 = 1;

   while (1) {
        *SET0 = 1 << 20;
        for (int c = DELAY; c != 0; c--) ;
        *CLR0 = 1 << 20;
        for (int c = DELAY; c != 0; c--) ;
   }
}
```

Let's convert some assembly to C!

```
.equ DELAY, 0x3F0000

    ldr r0, FSEL2
    mov r1, #1
    str r1, [r0]
    mov r1, #(1<<20)

loop:
    ldr r0, SET0
    str r1, [r0]

    mov r2, #DELAY
    wait1:
        subs r2, #1
        bne wait1

    ldr r0, CLR0
    str r1, [r0]

    mov r2, #DELAY
    wait2:
        subs r2, #1
        bne wait2

    b loop

FSEL2: .word 0x20200008
SET0:  .word 0x2020001C
CLR0:  .word 0x20200028
```

Globals

```
unsigned int *FSEL2 =  (unsigned int *)0x20200008;
unsigned int *SET0  =  (unsigned int *)0x2020001c;
unsigned int *CLR0  =  (unsigned int *)0x20200028;

#define DELAY 0x3f0000

void main(void)
{
   *FSEL2 = 1;

   while (1) {
        *SET0 = 1 << 20;
        for (int c = DELAY; c != 0; c--) ;
        *CLR0 = 1 << 20;
        for (int c = DELAY; c != 0; c--) ;
   }
}
```

Let's convert some assembly to C!

```
.equ DELAY, 0x3F0000

    ldr r0, FSEL2
    mov r1, #1
    str r1, [r0]
    mov r1, #(1<<20)

loop:
    ldr r0, SET0
    str r1, [r0]

    mov r2, #DELAY
    wait1:
        subs r2, #1
        bne wait1

    ldr r0, CLR0
    str r1, [r0]

    mov r2, #DELAY
    wait2:
        subs r2, #1
        bne wait2

    b loop

FSEL2: .word 0x20200008
SET0:  .word 0x2020001C
CLR0:  .word 0x20200028
```

Constant

```
unsigned int *FSEL2 =  (unsigned int *)0x20200008;
unsigned int *SET0  =  (unsigned int *)0x2020001c;
unsigned int *CLR0  =  (unsigned int *)0x20200028;

#define DELAY 0x3f0000

void main(void)
{
   *FSEL2 = 1;

   while (1) {
        *SET0 = 1 << 20;
        for (int c = DELAY; c != 0; c--) ;
        *CLR0 = 1 << 20;
        for (int c = DELAY; c != 0; c--) ;
   }
}
```

Let's convert some assembly to C!

```
.equ DELAY, 0x3F0000

    ldr r0, FSEL2
    mov r1, #1
    str r1, [r0]
    mov r1, #(1<<20)

loop:
    ldr r0, SET0
    str r1, [r0]

    mov r2, #DELAY
    wait1:
        subs r2, #1
        bne wait1

    ldr r0, CLR0
    str r1, [r0]

    mov r2, #DELAY
    wait2:
        subs r2, #1
        bne wait2

    b loop

FSEL2: .word 0x20200008
SET0:  .word 0x2020001C
CLR0:  .word 0x20200028
```

Assignment to a memory address

```
unsigned int *FSEL2 =  (unsigned int *)0x20200008;
unsigned int *SET0  =  (unsigned int *)0x2020001c;
unsigned int *CLR0  =  (unsigned int *)0x20200028;

#define DELAY 0x3f0000

void main(void)
{
    *FSEL2 = 1;

    while (1) {
        *SET0 = 1 << 20;
        for (int c = DELAY; c != 0; c--) ;
        *CLR0 = 1 << 20;
        for (int c = DELAY; c != 0; c--) ;
    }
}
```

Let's convert some assembly to C!

```
.equ DELAY, 0x3F0000

    ldr r0, FSEL2
    mov r1, #1
    str r1, [r0]
    mov r1, #(1<<20)

loop:
    ldr r0, SET0
    str r1, [r0]

    mov r2, #DELAY
    wait1:
        subs r2, #1
        bne wait1

    ldr r0, CLR0
    str r1, [r0]

    mov r2, #DELAY
    wait2:
        subs r2, #1
        bne wait2

    b loop

FSEL2: .word 0x20200008
SET0:  .word 0x2020001C
CLR0:  .word 0x20200028
```

More assignments

```
unsigned int *FSEL2 =  (unsigned int *)0x20200008;
unsigned int *SET0  =  (unsigned int *)0x2020001c;
unsigned int *CLR0  =  (unsigned int *)0x20200028;

#define DELAY 0x3f0000

void main(void)
{
    *FSEL2 = 1;

    while (1) {
        *SET0 = 1 << 20;
        for (int c = DELAY; c != 0; c--) ;
        *CLR0 = 1 << 20;
        for (int c = DELAY; c != 0; c--) ;
    }
}
```

Let's convert some assembly to C!

```
.equ DELAY, 0x3F0000

    ldr r0, FSEL2
    mov r1, #1
    str r1, [r0]
    mov r1, #(1<<20)

loop:
    ldr r0, SET0
    str r1, [r0]

    mov r2, #DELAY
    wait1:
        subs r2, #1
        bne wait1

    ldr r0, CLR0
    str r1, [r0]

    mov r2, #DELAY
    wait2:
        subs r2, #1
        bne wait2

    b loop

FSEL2: .word 0x20200008
SET0:  .word 0x2020001C
CLR0:  .word 0x20200028
```

Inner loops (note that in C there is no body!)

```
unsigned int *FSEL2 =  (unsigned int *)0x20200008;
unsigned int *SET0  =  (unsigned int *)0x2020001c;
unsigned int *CLR0  =  (unsigned int *)0x20200028;

#define DELAY 0x3f0000

void main(void)
{
    *FSEL2 = 1;

    while (1) {
        *SET0 = 1 << 20;
        for (int c = DELAY; c != 0; c--) ;
        *CLR0 = 1 << 20;
        for (int c = DELAY; c != 0; c--) ;
    }
}
```

Let's convert some assembly to C!

```
.equ DELAY, 0x3F0000

    ldr r0, FSEL2
    mov r1, #1
    str r1, [r0]
    mov r1, #(1<<20)

loop:
    ldr r0, SET0
    str r1, [r0]

    mov r2, #DELAY
    wait1:
        subs r2, #1
        bne wait1

    ldr r0, CLR0
    str r1, [r0]

    mov r2, #DELAY
    wait2:
        subs r2, #1
        bne wait2

    b loop

FSEL2: .word 0x20200008
SET0:  .word 0x2020001C
CLR0:  .word 0x20200028
```

Outer loop (goes forever)

```
unsigned int *FSEL2 =  (unsigned int *)0x20200008;
unsigned int *SET0  =  (unsigned int *)0x2020001c;
unsigned int *CLR0  =  (unsigned int *)0x20200028;

#define DELAY 0x3f0000

void main(void)
{
    *FSEL2 = 1;

    while (1) {
        *SET0 = 1 << 20;
        for (int c = DELAY; c != 0; c--) ;
        *CLR0 = 1 << 20;
        for (int c = DELAY; c != 0; c--) ;
    }
}
```

How do we get the C onto the Pi?

We use a *Makefile*, which has the commands in it (more on Makefiles later!):

```
$ make
arm-none-eabi-gcc -g -Wall -Og -std=c99 -ffreestanding  -c cblink.c -o cblink.o
arm-none-eabi-gcc -nostdlib -T memmap cblink.o -o cblink.elf
arm-none-eabi-objcopy cblink.elf -O binary cblink.bin
rm cblink.elf

$ rpm-install.py cblink.bin
```

Can we see what the C code produces? Yes!

```
.equ DELAY, 0x3F0000

    ldr r0, FSEL2
    mov r1, #1
    str r1, [r0]
    mov r1, #(1<<20)

loop:
    ldr r0, SET0
    str r1, [r0]

    mov r2, #DELAY
    wait1:
        subs r2, #1
        bne wait1

    ldr r0, CLR0
    str r1, [r0]

    mov r2, #DELAY
    wait2:
        subs r2, #1
        bne wait2

    b loop

FSEL2: .word 0x20200008
SET0:  .word 0x2020001C
CLR0:  .word 0x20200028
```

Use objdump on the .elf file:

```
$ arm-none-eabi-objdump -D cblink.elf

cblink.elf:     file format elf32-littlearm
Disassembly of section .text:

00008000 <main>:
    8000:   e59f3050    ldr   r3, [pc, #80]     ; 8058 <main+0x58>
    8004:   e5933000    ldr   r3, [r3]
    8008:   e3a02001    mov   r2, #1
    800c:   e5832000    str   r2, [r3]
    8010:   e59f0040    ldr   r0, [pc, #64]     ; 8058 <main+0x58>
    8014:   e3a01601    mov   r1, #1048576      ; 0x100000
    8018:   e3a0283f    mov   r2, #4128768      ; 0x3f0000
    801c:   e5903004    ldr   r3, [r0, #4]
    8020:   e5831000    str   r1, [r3]
    8024:   e1a03002    mov   r3, r2
    8028:   ea000000    b     8030 <main+0x30>
    802c:   e2433001    sub   r3, r3, #1
    8030:   e3530000    cmp   r3, #0
    8034:   1afffffc    bne   802c <main+0x2c>
    8038:   e5903008    ldr   r3, [r0, #8]
    803c:   e5831000    str   r1, [r3]
    8040:   e1a03002    mov   r3, r2
    8044:   ea000000    b     804c <main+0x4c>
    8048:   e2433001    sub   r3, r3, #1
    804c:   e3530000    cmp   r3, #0
    8050:   1afffffc    bne   8048 <main+0x48>
    8054:   eafffff0    b     801c <main+0x1c>
    8058:   0000805c    andeq r8, r0, ip, asr r0

Disassembly of section .data:

0000805c <FSEL2>:
    805c:   20200008    eorcs r0, r0, r8

00008060 <SET0>:
    8060:   2020001c    eorcs r0, r0, ip, lsl r0

00008064 <CLR0>:
    8064:   20200028    eorcs r0, r0, r8, lsr #32
```

Aside: Larson Scanner Different Speeds

Some students have written their Larson scanners and found out that the same delay amount produces different times whether you are going forwards or backwards!

It turns out that the reason why is particular to *the address* of instructions in the loop, in a particular and interesting way!

Notice the addresses in the program to the right (in hex): 8000, 8004, 8008, etc. They are at locations divisible by 4 (because each instruction is 4 bytes long)

```
$ arm-none-eabi-objdump -D cblink.elf

cblink.elf:     file format elf32-littlearm
Disassembly of section .text:

00008000 <main>:
    8000:   e59f3050    ldr   r3, [pc, #80]     ; 8058 <main+0x58>
    8004:   e5933000    ldr   r3, [r3]
    8008:   e3a02001    mov   r2, #1
    800c:   e5832000    str   r2, [r3]
    8010:   e59f0040    ldr   r0, [pc, #64]     ; 8058 <main+0x58>
    8014:   e3a01601    mov   r1, #1048576      ; 0x100000
    8018:   e3a0283f    mov   r2, #4128768      ; 0x3f0000
    801c:   e5903004    ldr   r3, [r0, #4]
    8020:   e5831000    str   r1, [r3]
    8024:   e1a03002    mov   r3, r2
    8028:   ea000000    b     8030 <main+0x30>
    802c:   e2433001    sub   r3, r3, #1
    8030:   e3530000    cmp   r3, #0
    8034:   1afffffc    bne   802c <main+0x2c>
    8038:   e5903008    ldr   r3, [r0, #8]
    803c:   e5831000    str   r1, [r3]
    8040:   e1a03002    mov   r3, r2
    8044:   ea000000    b     804c <main+0x4c>
    8048:   e2433001    sub   r3, r3, #1
    804c:   e3530000    cmp   r3, #0
    8050:   1afffffc    bne   8048 <main+0x48>
    8054:   eafffff0    b     801c <main+0x1c>
    8058:   0000805c    andeq r8, r0, ip, asr r0

Disassembly of section .data:

0000805c <FSEL2>:
    805c:   20200008    eorcs r0, r0, r8

00008060 <SET0>:
    8060:   2020001c    eorcs r0, r0, ip, lsl r0

00008064 <CLR0>:
    8064:   20200028    eorcs r0, r0, r8, lsr #32
```

Aside: Larson Scanner Different Speeds

It turns out that the processor is actually slower in some cases if some instructions are on a boundary that is not divisible by 16 instead of 4!

Why? Well, it has to do with how fast instructions can be pulled out of memory, and what is called "caching" — instructions are brought out of main memory in bunches into a faster memory (the cache), and if the code that you want to run happens to be offset from the chunks that are pulled out, the processor will spend more time pulling out chunks of addresses, slowing things down.

Let's look at blink.s and make some modifications!

```
$ arm-none-eabi-objdump -D cblink.elf

cblink.elf:     file format elf32-littlearm
Disassembly of section .text:

00008000 <main>:
    8000:   e59f3050    ldr   r3, [pc, #80]     ; 8058 <main+0x58>
    8004:   e5933000    ldr   r3, [r3]
    8008:   e3a02001    mov   r2, #1
    800c:   e5832000    str   r2, [r3]
    8010:   e59f0040    ldr   r0, [pc, #64]     ; 8058 <main+0x58>
    8014:   e3a01601    mov   r1, #1048576      ; 0x100000
    8018:   e3a0283f    mov   r2, #4128768      ; 0x3f0000
    801c:   e5903004    ldr   r3, [r0, #4]
    8020:   e5831000    str   r1, [r3]
    8024:   e1a03002    mov   r3, r2
    8028:   ea000000    b     8030 <main+0x30>
    802c:   e2433001    sub   r3, r3, #1
    8030:   e3530000    cmp   r3, #0
    8034:   1afffffc    bne   802c <main+0x2c>
    8038:   e5903008    ldr   r3, [r0, #8]
    803c:   e5831000    str   r1, [r3]
    8040:   e1a03002    mov   r3, r2
    8044:   ea000000    b     804c <main+0x4c>
    8048:   e2433001    sub   r3, r3, #1
    804c:   e3530000    cmp   r3, #0
    8050:   1afffffc    bne   8048 <main+0x48>
    8054:   eafffff0    b     801c <main+0x1c>
    8058:   0000805c    andeq r8, r0, ip, asr r0

Disassembly of section .data:

0000805c <FSEL2>:
    805c:   20200008    eorcs r0, r0, r8

00008060 <SET0>:
    8060:   2020001c    eorcs r0, r0, ip, lsl r0

00008064 <CLR0>:
    8064:   20200028    eorcs r0, r0, r8, lsr #32
```

Know Your Tools!

- Assembler (as)
  - Transform assembly code (text) into object code (binary machine instructions)
  - This is a mechanical translation, with few surprises

- Compiler (gcc)
  - Transform C code (text) into object code (likely first translated into asm then into object form, C ☞ asm ☞ object)
  - This is a *complex* translation, with a large potential for optimization

Know Your Tools!

- When coding directly in assembly, *the instructions you see are the instructions you get,* with no surprises!

- For C source, you may need to drop down to see what the compiler has generated to be sure of what you're getting.

- What transformations are *legal*?
- What transformations are *desirable*?

Extra Slides

Extra Slides

The Compiler sometimes needs hints when you are dealing with real hardware!

Extra Slides

Calculating the number of instructions per second of your Pi

Based on the blinking LED, you can get a good idea of how many instructions your Pi is running under the current configuration:

1. Time ten on/off flashes (8 seconds on my Pi). **`8sec/10 flashes`**
2. Look at the number of instructions in a on/off loop:

```
wait1:
    subs r2, #1
    bne wait1
```

`2 instructions`

3. Each loop is executed **`0x3F0000`** (decimal: **`4128768`**) times, and there are two loops, so there are a total of **`2 inst. * 4128768 * 2 = 16515072 instructions per flash.`**
4. Therefore:

`16515072 inst/flash * 10 flashes / 8 sec =` **`20,643,840`** `instructions per second`

Extra Slides

```
16515072 inst/flash * 10 flashes / 8 sec = 20,643,840 instructions per second
```

The Raspberry Pi has a clock frequency of 700MHz
Why are our Pis only running 20M instructions per second?

There are a number of factors:

1. Caches are turned off, and memory is slow. All instructions must be read from memory, so this makes things much slower. Memory reads are 50x slower than caches, and 100x slower than registers.
2. There are other hardware factors that can change the instructions per second count, too.

Code	Suffix	Description	Flags
0000	EQ	Equal / equals zero	Z
0001	NE	Not equal	!Z
0010	CS / HS	Carry set / unsigned higher or same	C
0011	CC / LO	Carry clear / unsigned lower	!C
0100	MI	Minus / negative	N
0101	PL	Plus / positive or zero	!N
0110	VS	Overflow	V
0111	VC	No overflow	!V
1000	HI	Unsigned higher	C and !Z
1001	LS	Unsigned lower or same	!C or Z
1010	GE	Signed greater than or equal	N == V
1011	LT	Signed less than	N != V
1100	GT	Signed greater than	!Z and (N == V)
1101	LE	Signed less than or equal	Z or (N != V)
1110	AL	Always (default)	any