CS/COE 0447

Bitwise Operations

wilkie (with content borrowed from:

Jarrett Billingsley

Dr. Bruce Childers)

Bitwise Operations

Being… wise… about… bits??? (Doing stuff to them)

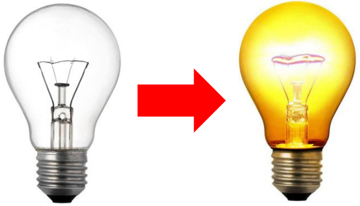
What are "bitwise" operations?

- The "numbers" we use on computers aren't *really* numbers right?
- It's often useful to treat them instead as **a pattern of bits**.
- **Bitwise operations** treat a value as a pattern of bits.

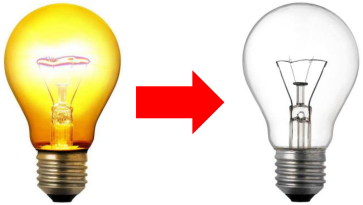
1 0 0 0

The simplest operation: NOT (logical negation)

- If the light is off, turn it on.



- If the light is on, turn it off.

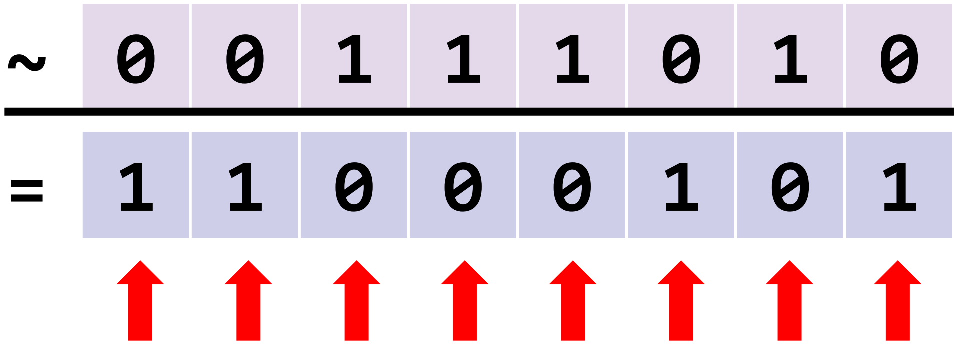


A	Q
0	1
1	0

- We can summarize this in a **truth table**.
- We write NOT as **~A**, or **¬A**, or $\overline{\mathbf{A}}$

Applying NOT to a whole bunch of bits

- if we use the **not** instruction (or ~ in C/Java), this is what happens:



```
~ 0 0 1 1 1 0 1 0
= 1 1 0 0 0 1 0 1
```

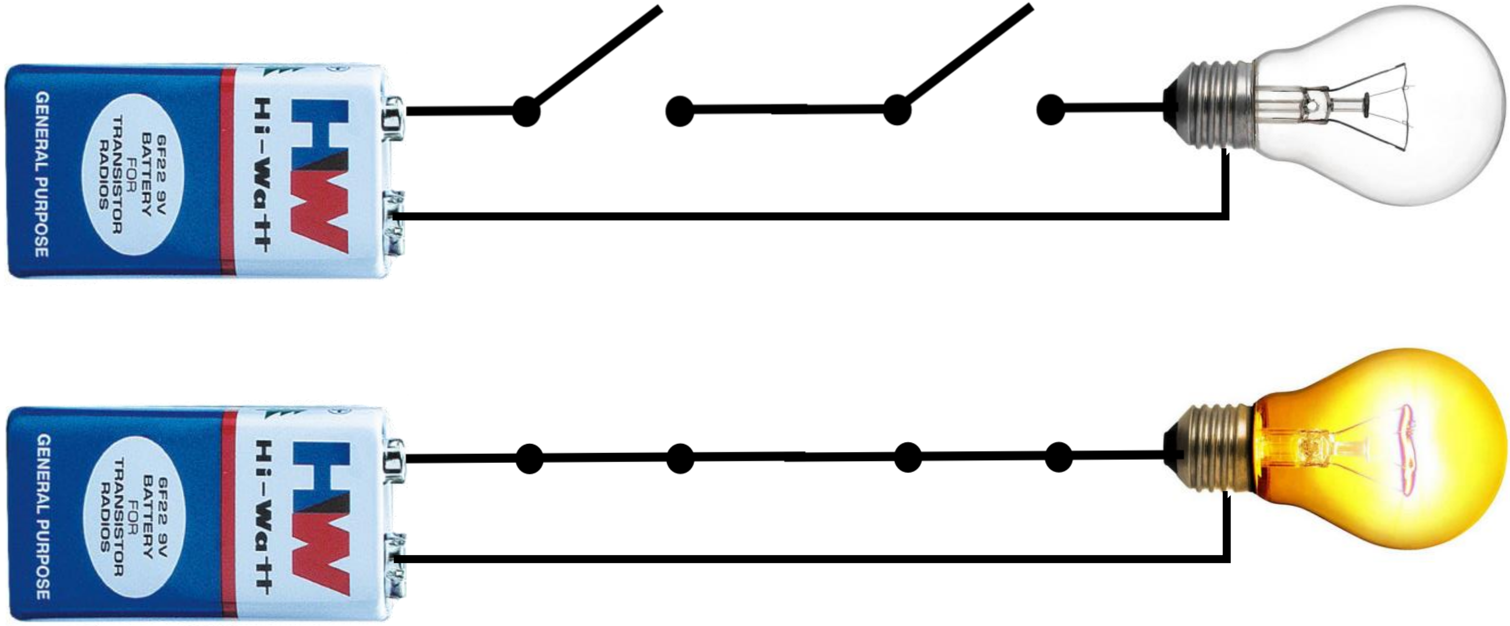
we did 8 **independent** NOT operations

That's it.

only 8 bits shown cause 32 bits on a slide is too much

Let's add some switches

- there are two switches in a row connecting the light to the battery
- **how do we make it light up?**

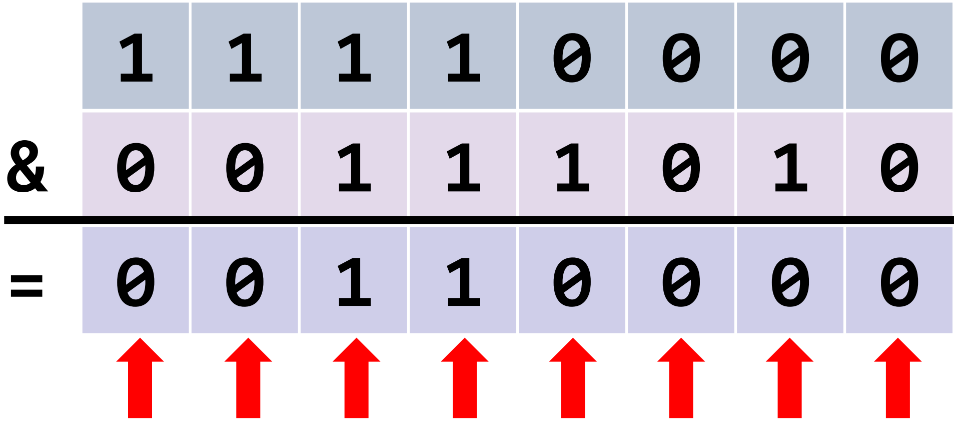

HW
Hi-Watt
6F22 9V BATTERY FOR TRANSISTOR RADIOS
GENERAL PURPOSE
HW
Hi-Watt
6F22 9V BATTERY FOR TRANSISTOR RADIOS
GENERAL PURPOSE

AND (Logical product)

- AND is a **binary (two-operand) operation**.
- It can be written a number of ways:

A&B   A∧B   A·B   AB

- If we use the **and** instruction (or & in C/Java):



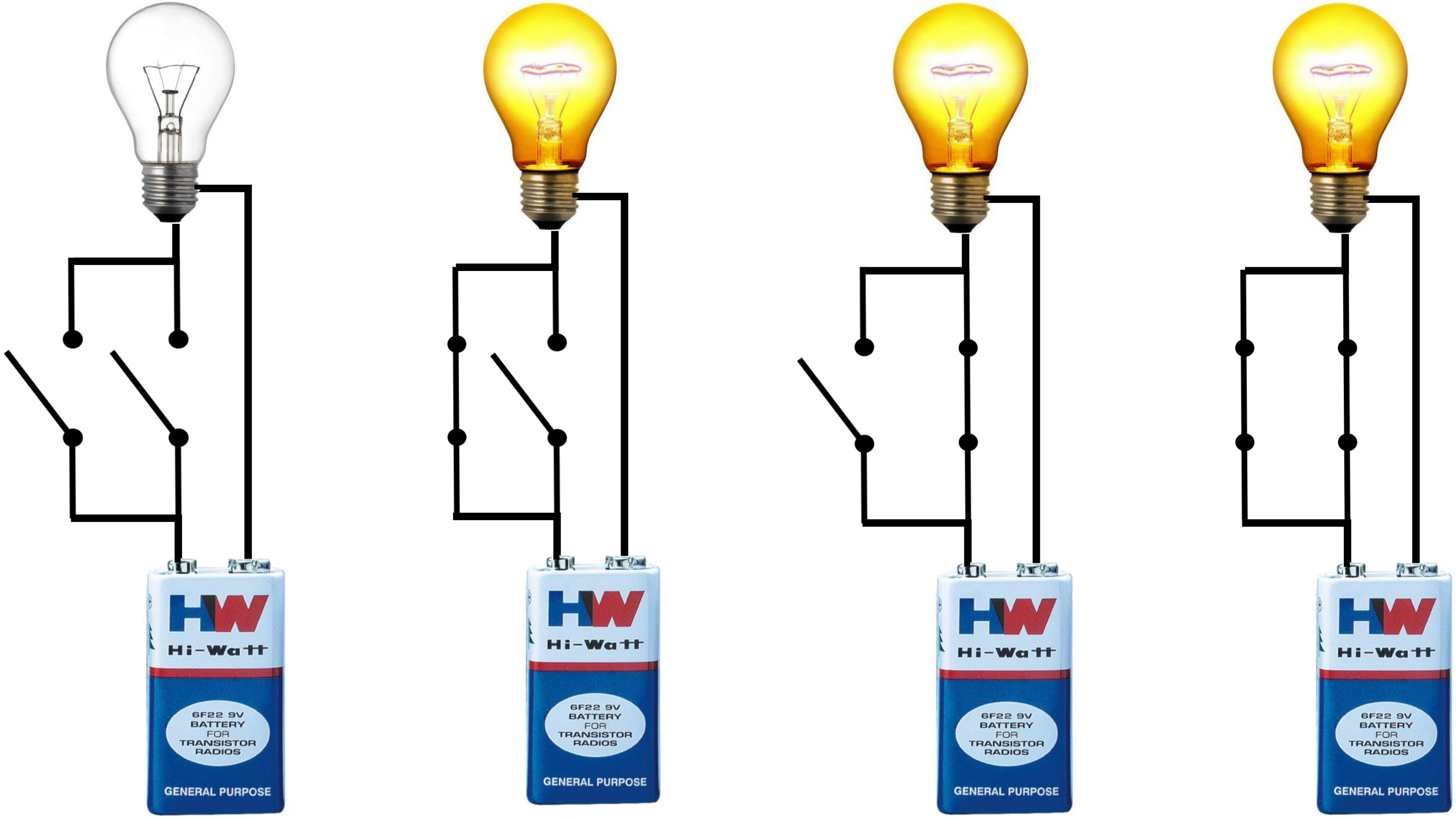
	1	1	1	1	0	0	0	0
&	0	0	1	1	1	0	1	0
=	0	0	1	1	0	0	0	0

we did 8 **independent** AND operations

A	B	Q
0	0	0
0	1	0
1	0	0
1	1	1

"Switching" things up ;))))))))))))))))))))))

- NOW how can we make it light up?



OR (Logical sum...?)

- We might say **"and/or"** in English.
- It can be written a number of ways:

  A|B A∨B A+B

- If we use the **or** instruction (or | in C/Java):

```
  1 1 1 1 0 0 0 0
| 0 0 1 1 1 0 1 0
-----------------
= 1 1 1 1 1 0 1 0
```

We did 8 **independent** OR operations.

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	1

lui, ori...

- if I write `li t0, 0xDEADBEEF` in MIPS, the assembler turns it into:

```
lui at, 0xDEAD
ori t0, at, 0xBEEF
```

- the reason it splits it up is that **there's only enough space in each instruction to fit half of 0xDEADBEEF**
  - we'll learn about instruction encoding later
  - but it suffices to say **each immediate is 16 bits long**
- what the heck are these instructions *doing* tho

MIPS: lui / ori (32-bit immediates)

- **lui** means **load upper immediate.** it puts the immediate value into the **upper 16 bits of the register**, and zeroes out the rest

```
lui at, 0xDEAD
```

- then, **ori** does logical OR of **at** and its **zero-extended** immediate

```
ori t0, at, 0xBEEF
```

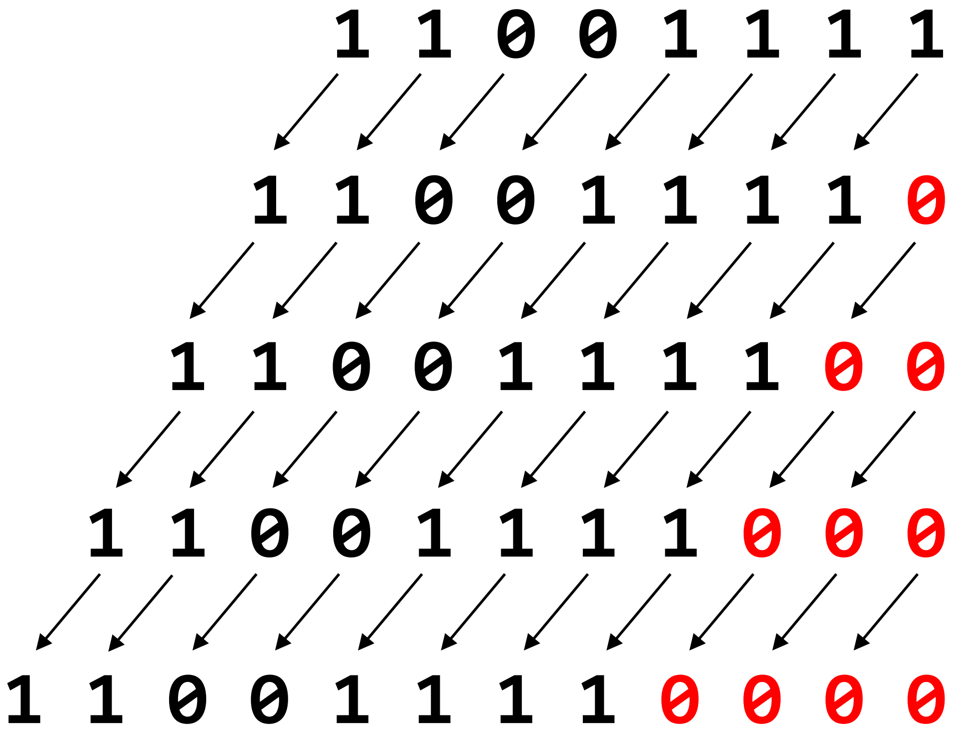
```
  1 1 0 1 1 1 1 0 1 0 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 1 1 1 0 1 1 1 0 1 1 1 1
-------------------------------------------------------------------
  1 1 0 1 1 1 1 0 1 0 1 0 1 1 0 0 1 0 1 1 1 1 1 0 1 1 1 0 1 1 1 1
     D       E       A       D       B       E       E       F
```

Bit Shifting

To the left, to the left! To the right, to the right!

Bit shifting

- Besides AND, OR, and NOT, we can **move bits around**, too.



1 1 0 0 1 1 1 1

if we shift these bits **left by 1…**

1 1 0 0 1 1 1 1 0

we stick a **0** at the bottom

1 1 0 0 1 1 1 1 0 0

again!

1 1 0 0 1 1 1 1 0 0 0

AGAIN!

1 1 0 0 1 1 1 1 0 0 0 0

AGAIN!!!!

Left-shifting in C/Java and MIPS

(animated)

- C and Java use the << operator for left shift

```
B = A << 4; // B = A shifted left 4 bits
```

- MIPS has the **sll** (**S**hift **L**eft **L**ogical) instruction

```
sll t2, t0, 4 # t2 = t0 << 4
```

- If the bottom 4 bits of the result are now 0s...
  - ...what happened to the *top* 4 bits?

```
0011 0000 0000 1111 1100 1101 1100 1111
```

the bit bucket is not a real place

it's a programmer joke ok

in the UK they might say the “Bit Bin”

bc that’s their word for trash

>_> >_> >_> ☺

- We can **shift right, too**

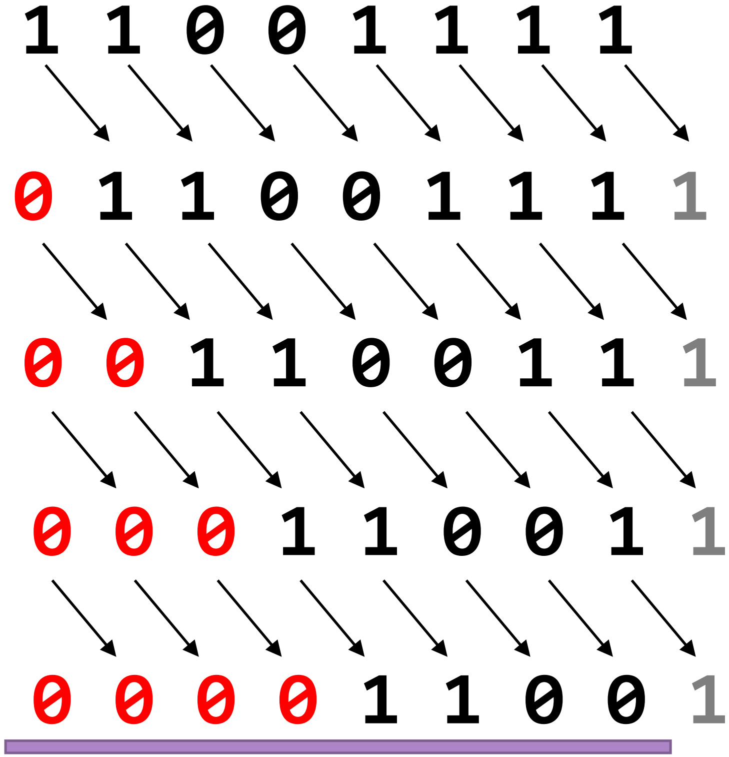
```
0 0 1 1 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 1 1 0 1 1 1 0 0 1 1 1 1
0 0 0 1 1 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 1 1 0 1 1 1 0 0 1 1 1
0 0 0 0 1 1 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 1 1 0 1 1 1 0 0 1 1
0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 1 1 0 1 1 1 0 0 1
0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 1 1 0 1 1 1 0 0
```

- C/Java use >>, MIPS uses **srl** (**S**hift **R**ight **L**ogical)

see what I mean about 32 bits on a slide

MIPS ISA: srl (shift right logical)

- We can **shift right, too (srl)**



```
1 1 0 0 1 1 1 1
0 1 1 0 0 1 1 1 1
0 0 1 1 0 0 1 1 1
0 0 0 1 1 0 0 1 1
0 0 0 0 1 1 0 0 1
```

if we shift these bits **right by 1...**

we stick a **0** at the top

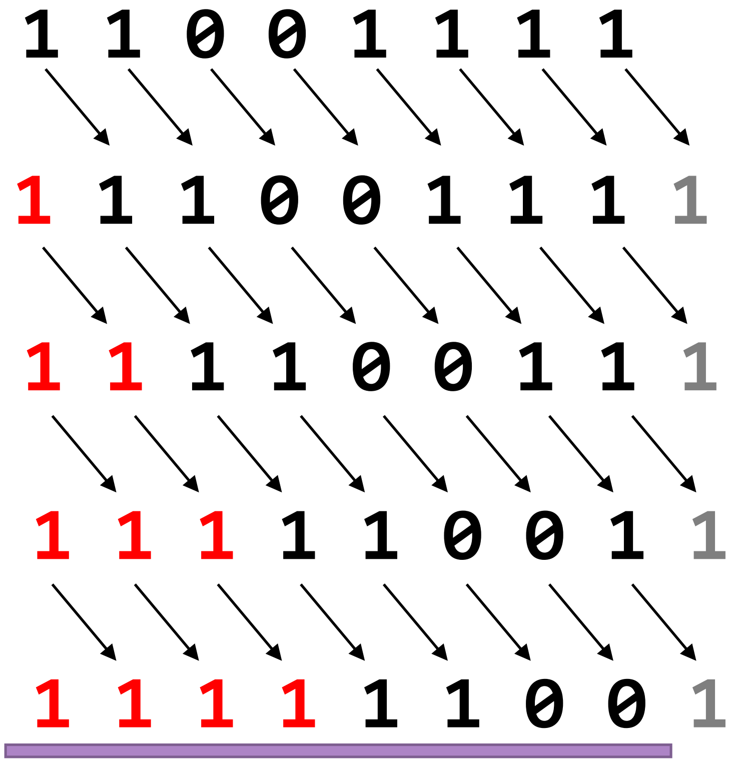
again!

AGAIN!

Wait... what if this was a negative number?

MIPS ISA: srl (shift right arithmetic)

- We can **shift right with sign-extension, too (sra)**



1 1 0 0 1 1 1 1

1 1 1 0 0 1 1 1 1

1 1 1 1 0 0 1 1 1

1 1 1 1 1 0 0 1 1

1 1 1 1 1 1 0 0 1

if we shift these bits **right by 1…**

we copy the **1** at the top (or 0, if MSB was a 0)

again!

AGAIN!

AGAIN!!!!!! (It's still negative!)

Is there a sla instruction?

Huh... that's weird

- Let's start with a value like 5 and shift left and see what happens:

Binary	Decimal
101	5
1010	10
10100	20
101000	40
1010000	80

Why is this happening

Well uh... what if I gave you

49018853

How do you multiply that by 10?

by 100?

by 100000?

Something **very similar** is happening here

a << n == a * 2^n

- **Shifting left by *n* is the same as multiplying by 2^n**
  - You probably learned this as "moving the decimal point"
    - And moving the decimal point *right* is like shifting the digits *left*
- **Shifting is fast and easy on most CPUs**
  - Way faster than multiplication in any case
- Hey... if shifting *left* is the same as multiplying...

a >> n == a / 2^n, ish

- You got it
- **Shifting right by n is like dividing by 2^n**
  - ***sort of.***
- What's 101_2 shifted right by 1?
  - 10_2, which is 2…
    - It's like doing **integer** (or **flooring**) division
- How do we do “actual” multiplication/division?
  - We will get to that next week or so!

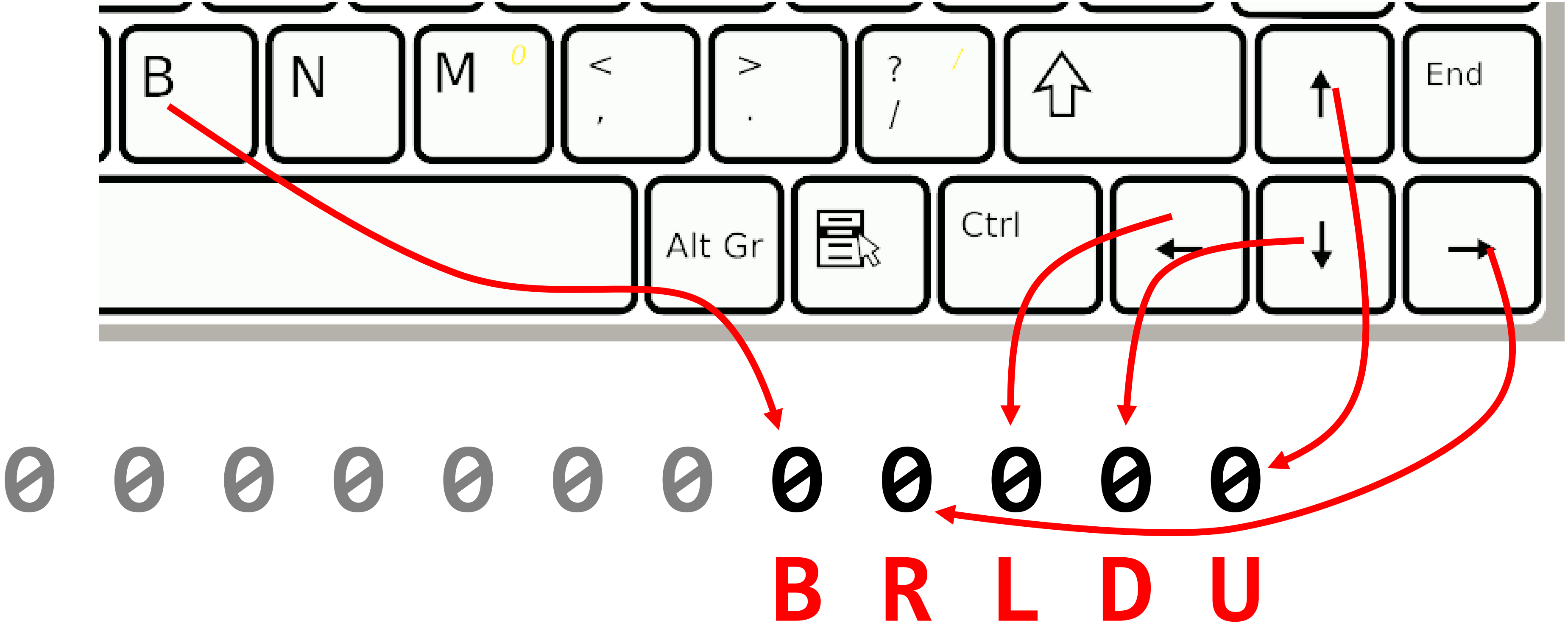
Okaaaaay… so what

Do shifts seem useless? Bitfields, come on out!

clicky clicky

- In the LED Keypad plugin in MARS, input works like this:

`input_get_keys` returns a value in v0...

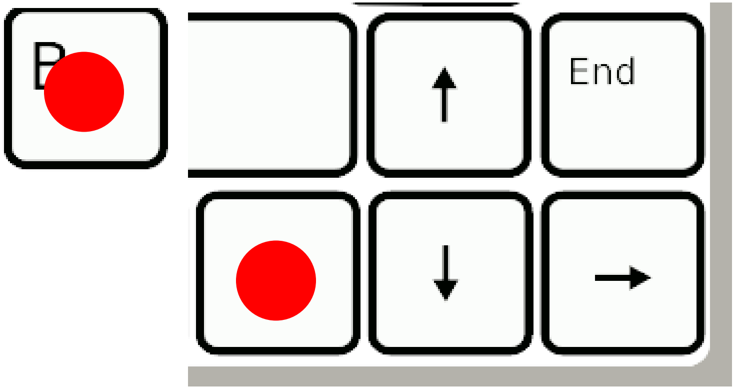


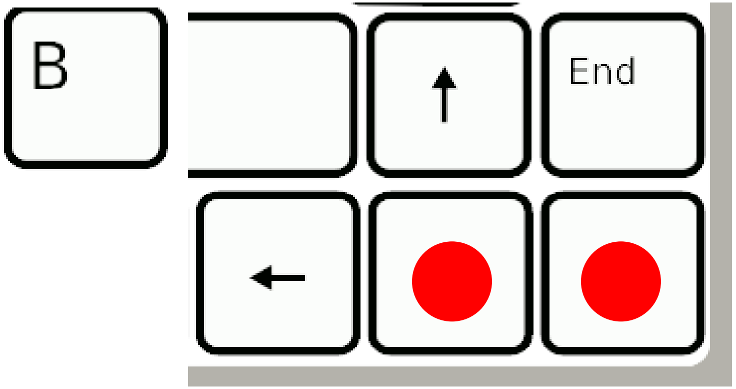
0 0 0 0 0 0 0 0 0 0 0 0

B R L D U

Why do we do this??

- It lets us cram several booleans into a **single** value!
- This technique is known as **bit flags**. We'll see more of these next time!





```
1 0 1 0 0
B R L D U
```

```
0 1 0 1 0
B R L D U
```