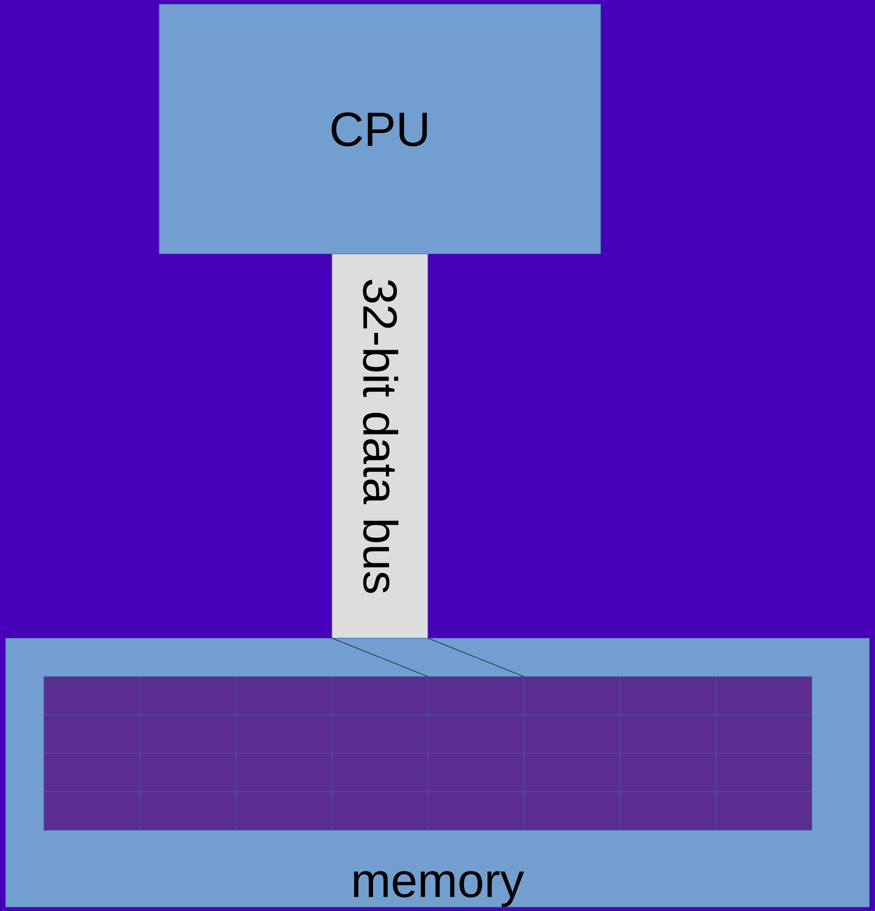
Alignment and Endianness

ECE 362
https://engineering.purdue.edu/ece362/

LDR Rt, [Rn, #imm]: offset is shifted

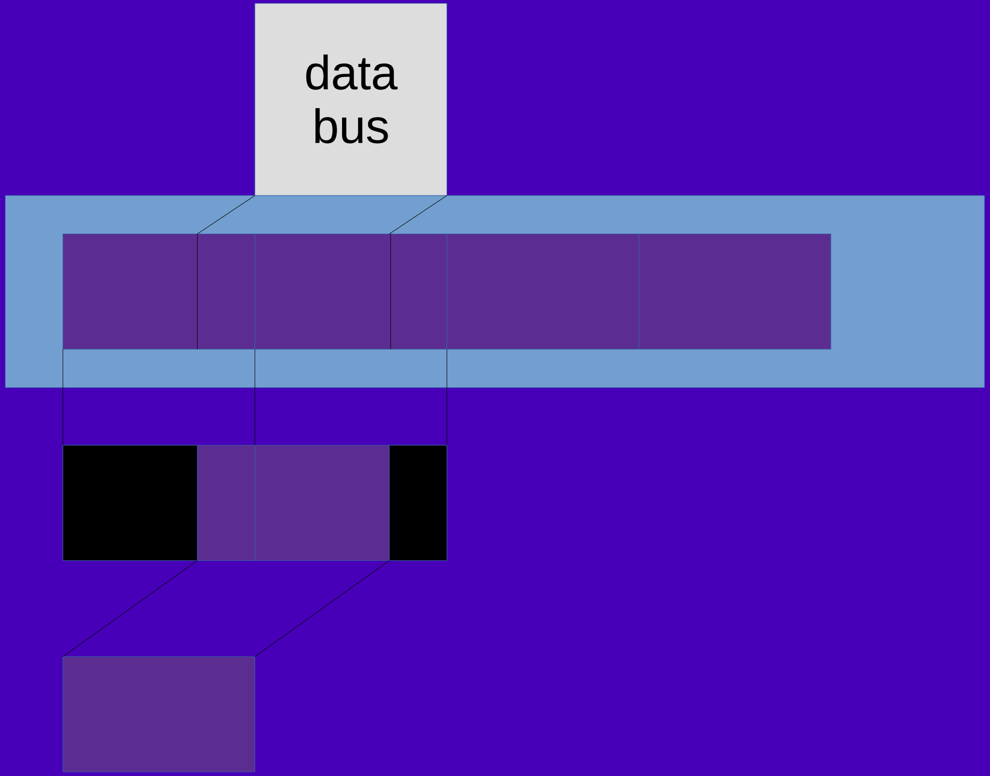
- #imm must be divisible by 4.
- Contents of Rn must be divisible by 4.
- Rn + #imm must be divisible by 4.
  - Necessary for **alignment**.
- imm is encoded into instruction as a 5-bit value that can represent multiples of 4 from 0 – 124.

Why is this “alignment” needed?



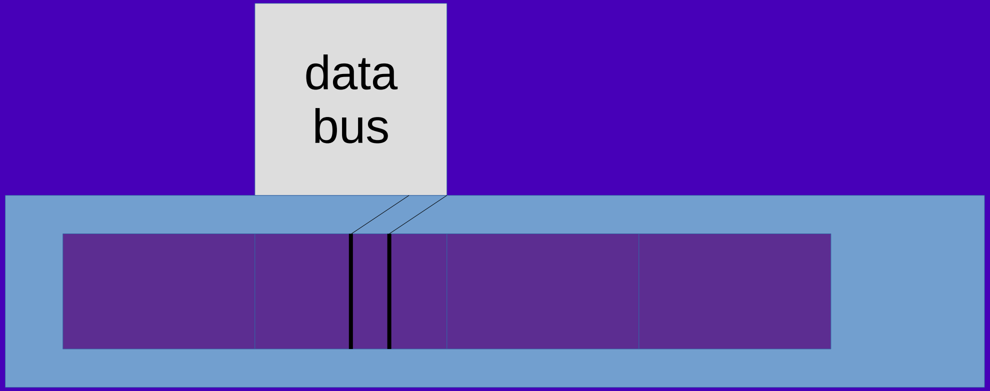
- The Cortex-M0 CPU has a 32-bit internal data bus.
- Memory words are also 32-bits wide.
- Accessing a 32-bit memory word is simple.

Why is this “alignment” needed?



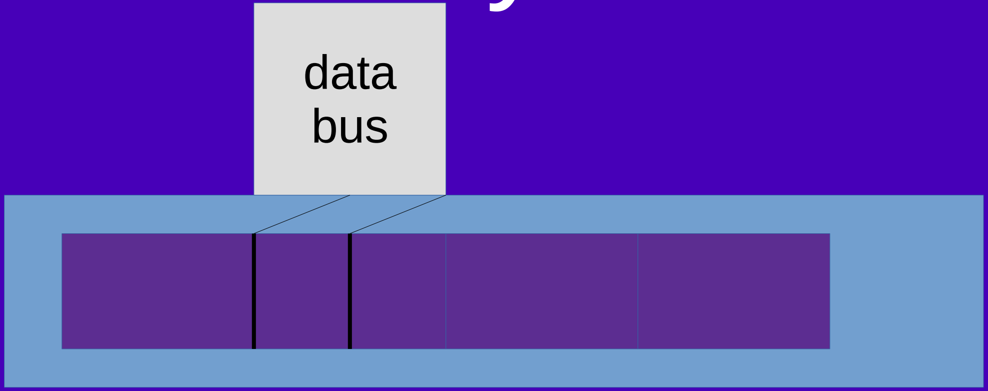
- Reading memory across a 32-bit boundary is difficult:
  - Read one word.
  - Mask off the part we don’t need.
  - Read next word.
  - Mask off the part we don’t need.
  - Combine and shift.
- Cortex-M0 doesn’t have hardware to do this.
- Writes are even harder.

Byte reads/writes allowed.



- Cortex-M0 can read/ write byte in any location in memory.
    - Just enough shift/mask hardware to do this.

Halfword reads/writes allowed only on even boundaries.



- Cortex-M0 can read/write halfword (2-bytes) in any even location in memory.
  - Just enough shift/mask hardware to do this.

Nothing mysterious about alignment

- Early RISC CPUs had alignment constraints.
  - Easy to cause problems.
  - Parsing, network protocol handling often lead to cases where you want to do unaligned access.
- CPUs that support unaligned access are easier to program.
  - Not doing so led to lost sales.
  - Most modern CPUs support unaligned memory accesses.
    - Including higher-end ARM CPUs.
- Cortex-M0 is *architected* to not support unaligned accesses.
  - The *instructions* and *registers* won’t support arbitrary unaligned accesses.
  - Consider the SP register...

Consider the following code.

```
// sp is unmodified
//
mov   r7, sp
adds r7, #1
mov   sp, r7
```

- Can the SP register contain an odd number?
  - No.
  - The two least significant bits of SP are hard-wired to 0.
  - The SP value is unchanged by these instructions.

Where did term ‘endian’ originate?

- Jonathan Swift. Irish satirist.
  - 30 Nov 1667 – 19 Oct 1754
  - *A Modest Proposal* (1729)
  - *Gulliver’s Travels* (1726)
    - Lilliputians and Blefuscudians were at war because:
      - Lilliputians broke breakfast eggs on the little end (little endians)
      - Blefuscudians broke breakfast eggs on the big end (big endians)

What do we mean by "endian"?

- This is not something to get worked up about.
- This is not something to get worked up about.
- Consider the following code:

```
ldr     r1, =0x20000000 // Start address of SRAM
ldr     r0, =0xfedcba98 // An interesting 4-byte number
str     r0, [r1]        // Store interesting number in SRAM.
ldr     r2, [r1]        // Read the 4-byte integer
ldrb    r3, [r1,#0]     // Read the first byte of it.
ldrb    r4, [r1,#3]     // Read the fourth byte of it.
```

- What does R2 contain?
- How about R3?
- How about R4?

Endianness is only a factor when looking at bytes of memory

- ARM Cortex-M0 stores the "little end" (the least significant byte) of a 4-byte integer first (at the lowest address) in memory.
  - That's called "little endian."
  - R3 would contain the least significant byte 0x98.
  - R4 would contain the most significant byte 0xfe.
  - R2 is ALWAYS read back the way it is written. (How could this not be so?)
- Normally, we look at memory like this.
  - And this seems backward for the way the western world looks at tables.

```
          0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
20000000 98 ba dc fe 00 00 00 00 00 00 00 00 00 00 00 00
20000010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

So let’s change the way we look at memory...

- We could just as easily look at memory like this.
  - Then the integers would look right, yes?
  - Problem solved!

```
 f  e  d  c  b  a  9  8  7  6  5  4  3  2  1  0
00 00 00 00 00 00 00 00 00 00 00 00 fe dc ba 98 20000000
00 00 0a 21 64 6c 72 6f 57 20 2c 6f 6c 6c 65 48 20000010
```

Unfortunately, strings of characters are always big-endian...

- Awwww....

```
                   f  e  d  c  b  a  9  8  7  6  5  4  3  2  1  0
................ 00 00 00 00 00 00 00 00 00 00 00 00 fe dc ba 98 20000000
...!dlroW ,olleH 00 00 0a 21 64 6c 72 6f 57 20 2c 6f 6c 6c 65 48 20000010
```

This is usually the best you can do.

- At some point, you get a sense for when you need to reverse the bytes of a number.
  - It only happens when looking at memory one byte at a time.

```
          0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
20000000 98 ba dc fe 00 00 00 00 00 00 00 00 00 00 00 00 ................
20000010 48 65 6c 6c 6f 20 57 6f 72 6c 64 21 0a 00 00 00 Hello, World!....
```

ARM Cortex-M0 endian instrs

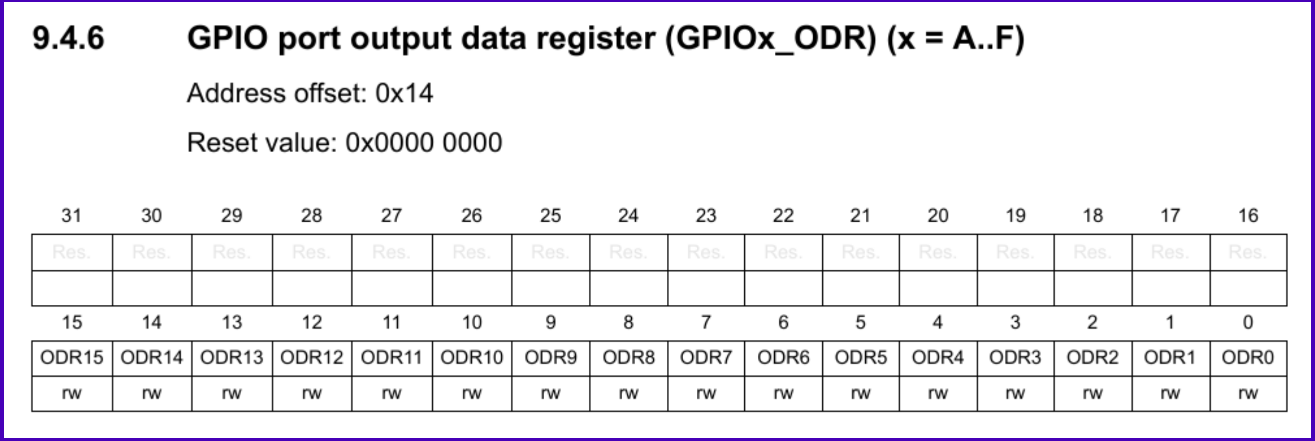
- There are instructions to reverse the byte order of a word:
  - REV: this will do it
  - REV16: not what you want
  - REVSH: also not what you want

Almost Everything in the World is little-endian.

- Get used to it.
- The ARM Cortex-M0 *architecture* is actually specified as bi-endian.
  - This is mostly a marketing claim.
  - It is seldom used in big-endian mode.
  - Instructions are, nevertheless, ALWAYS little-endian.
- Network protocols (Ethernet, IP, UDP, TCP, etc) use big-endian byte order.
  - For the four-byte IP address 128.46.154.76, 128 is sent first.
  - Applications must regularly swap bytes to communicate via the network.
- Bits in a byte are almost always specified big-endian.
  - Except serial protocols like RS-232, Ethernet, USB.
  - I^2C is a communication protocol that uses big-endian bit format.
- ECE 362 used big-endian CPUs for >35 years until 2018.

Is little-endian better in any way?

- One small way: I/O register access.
- Consider the GPIO ODR register:



9.4.6 GPIO port output data register (GPIOx_ODR) (x = A..F)

Address offset: 0x14

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

- What happens if we do an STRB to this register?

Section 9.4 of the FRM says:

- "The peripheral register can be written in word, half word or byte mode."
- If we write a byte to 0x4800 0014, that will be the lowest significant byte (bits 7 – 0).
- Some I/O registers are defined as a single byte of valid data in a 32-bit word. (e.g. the DAC)
  - If we were using big-endian order, we'd have to be careful.
  - Because its little-endian, we don't have to think about it.