CS/COE 0447

Introduction to:
Introduction to
Computer Architecture

wilkie (with content borrowed from:

Jarrett Billingsley

Dr. Bruce Childers)

Memory

Where data calls “home”

What is the memory?

- the **system memory** is a piece of *temporary* storage hardware
  - it's smaller and faster than the *persistent* storage. (disk, etc)
    - maybe in the future it won't be temporary, or the line between system memory and persistent storage will go away...
- it's where the **programs and data** that the computer is currently executing and using reside
  - all the variables, all the functions, all the open files etc.
  - the CPU **can only run programs from system memory!**

Memory: Looking at Bytes

- the memory is **a big one-dimensional array of bytes**
- what do these bytes mean?
  - ¯_(ツ)_/¯
- every byte value has an **address**
  - this is its "array index"
  - addresses start at 0, like arrays in C/Java
    - gee wonder where they got the idea
- when each byte has its own address, we call it a **byte-addressable machine**
  - not many *non*-byte-addressable machines these days
  - A non-byte-addressable machine would have addresses that refer to **words**, etc. (Getting data in-between words may be difficult… it is called misaligned data. It’s a yuck problem.)
  - Thankfully, we are using a byte-addressable architecture

Addr	Val
0	00
1	30
2	04
3	00
4	DE
5	C0
6	EF
7	BE
8	6C
9	34
A	00
B	01

Memory: It's Finite (but how much?)

- each address refers to *one* byte. if your addresses are *n* bits long… **how many bytes** can your memory have?
  - **2^n B**
- machines with 32-bit addresses can access 2^{32} B = **4GiB** of memory
  - with 64-bit addresses… **16EiB** lol
- kibi, mebi, gibi, tebi, pebi, exbi are powers of 2
  - **kiB** = $2^{10,}$ **MiB** = 2^{20}, **GiB** = 2^{30} etc.
- kilo, mega, giga, tera, peta, exa are *ostensibly* powers of 10
  - **kB** = 10^3, **MB** = 10^6, **GB** = 10^9 etc.
- **but most people still say "kilo, mega" to mean the powers of 2**
  - really only hard drive manufacturers use the "power of 10"
    - 1TB hard drive is 10^{12}B… $10^{12} \div 2^{40}$ = **909 GiB**
      - now you know why it's like that

Memory: Looking at Words

- for most things, we want to use **words**
  - the "comfortable" integer size for the CPU
  - on this version of MIPS, it's **32b** (**4B**)
- but our memory only holds bytes... wat do
- **combine multiple bytes into larger values**
  - the CPU can handle this for us
  - but importantly, *the data is still just bytes*
- when we talk about values bigger than a byte...
  - the **address** is **the address of their first byte**
    - the byte at the *smallest* address
  - so what are the addresses of the three words here?

Addr	Val
0	00
1	30
2	04
3	00
4	DE
5	C0
6	EF
7	BE
8	6C
9	34
A	00
B	01
C	02

Endianness

Because We Like to Make Numbers More Confusing, Honestly

A Matter of Perspective

- Let's say there's a word at address 4... made of 4 bytes (32-bits)
- *Wh...what word do those 4 bytes represent?*

If we think of addresses *increasing downward...* ...is it **`0xDEC0EFBE`**?

Addr	Val
...	...
4	DE
5	C0
6	EF
7	BE
...	...

Addr	Val
...	...
7	BE
6	EF
5	C0
4	DE
...	...

if we think of addresses *increasing upward...* ...is it **`0xBEEFC0DE`**?

The Two Endians

- When interpreting a *sequence of bytes* as larger values, **endianness** is the rule used to decide **what order to put the bytes in**

big-endian means the **first byte** is the "big end:" the **most significant** byte

0	1	2	3
DE	C0	EF	BE

little-endian means the **first byte** is the "little end:" the **least significant** byte

0xDEC0EFBE

0xBEEFC0DE

first byte = **byte at smallest address**

nothing to do with ***value*** of bytes, only ***order***

Which is "better" ?

- it doesn't matter.* **as long as you're consistent,** it's fine
- for political (x86) reasons, most computers today are little-endian
- but endianness pops up whenever you have sequences of bytes:
  - like in files
  - or networks
  - or hardware buses
  - or… memory!
- which one is MIPS?
  - it's *bi-endian,* meaning it can be configured to work either way
  - **but** MARS uses the **endianness of the computer it's running on**
    - so **little-endian** for virtually everyone
      - cause x86
        - *ugh*

What Doesn't Endianness Affect?

× the arrangement of the bits ***within a byte***
- it just changes **meaning of order of the bytes**
  - note the bytes are still `DE, C0` etc.

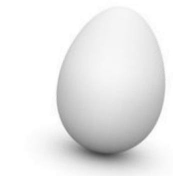
× **1-byte** values, arrays of bytes, **ASCII** strings…
- *single bytes* don't care about endianness at all

× the ordering of bytes **inside the CPU**
- there's no need for e.g. "big-endian" arithmetic
- the CPU works with whole *words*

- endianness only comes up when moving data:
  - larger than single bytes
  - between **the CPU and memory**
  - or between **multiple computers**

```
0xBEEFC0DE
0xED0CFEEB
0xDEC0EFBE
l l e H o
H e l l o
```

The Messy Origin of Endianness



- There was once a nation called Lilliput.
- Once, a future king cut themselves when breaking their boiled egg big-end first.
- So the emperor outlawed such a practice and eggs can only be cut my their little end first.
- Many rebellions, civil wars, and executions happened on both sides as each fought to win the ability to eat their eggs whichever way. The big-endians vs. the little-endians.
- It's war/political satire from "Gulliver's Travels" by Jonathan Swift. Used again as satire to ***complain about how ridiculous computer endianness is.*** *~~ Infinite sighs ~~*

Variables, Loads, and Stores

What are loads? Stores? And why Variables are Not Scary in asm.

Memory Addresses

- *everything* in memory has an **address**
  - the position in memory **where it begins**
    - where its **first byte** is
  - this applies to variables, functions, objects, arrays etc.
- a super important concept:

every variable really has **two parts:**
an **address** and a **value**

- if you want to put a variable in memory…
  - first you need to figure out **what address to put it in**
  - this *extremely tedious* task is handled by assemblers
    - whew

Declaring Global Variables

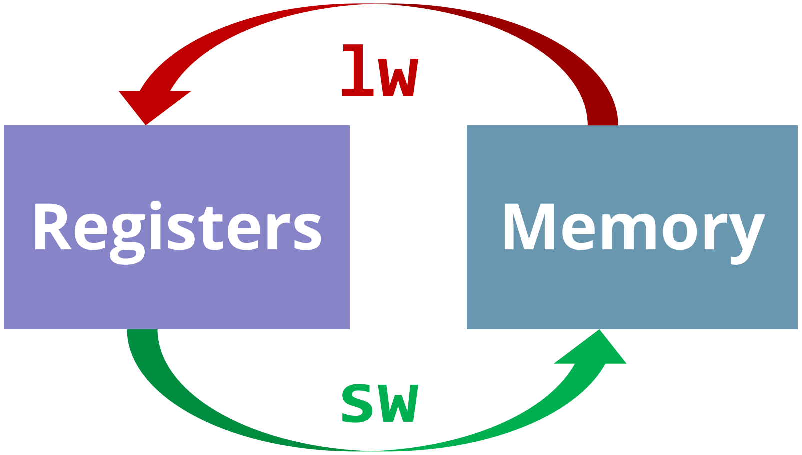
- we can declare a global variable like this (at the top of a file):

```
.data
    x: .word 4
```

- the Java equivalent would be `static int x = 4;`
- `.data` says "I'm gonna declare variables"
  - you can declare as many as you want!
  - to go back to writing code, use `.text`
- if we assemble this little program and make sure **Tools > Show Labels Window** is checked, what do you see?
  - the assembler *gave the variable* that address
  - it'll do that for every variable

Load-Store Architectures

- in some architectures, *many* instructions can access memory
  - x86-64: `add [rsp-8], rcx`
    - adds the contents of `rcx` to the value at address `rsp-8`
- in a **load-store** architecture, **all** memory accesses are done with two kinds of instructions: loads and stores (like in MIPS)

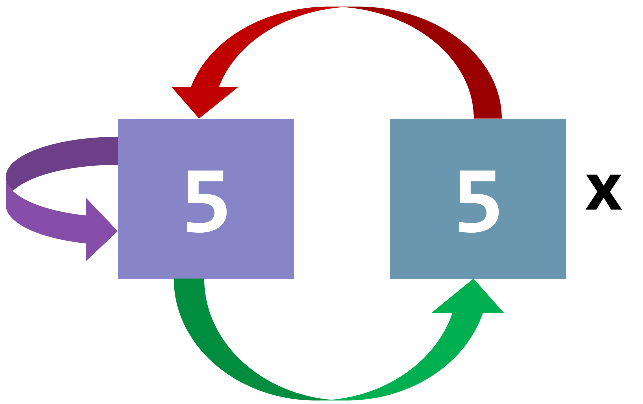


loads copy data **from** memory **into** CPU registers

stores copy data **from** CPU registers **into** memory

Operating On Variables in Memory

- we want to increment a variable that is **in memory**
  - where do values **have to be for the CPU to operate on them?**
  - what do we want the **overall outcome to be?**
- so, what **three steps** are needed to increment that variable?
  1. **load** the value from memory into a register
  2. **add 1** to the value in the register
  3. **store** the value back into memory
- **every variable access** works like this!!!
  - High Level Languages (HLLs) just hide this from you



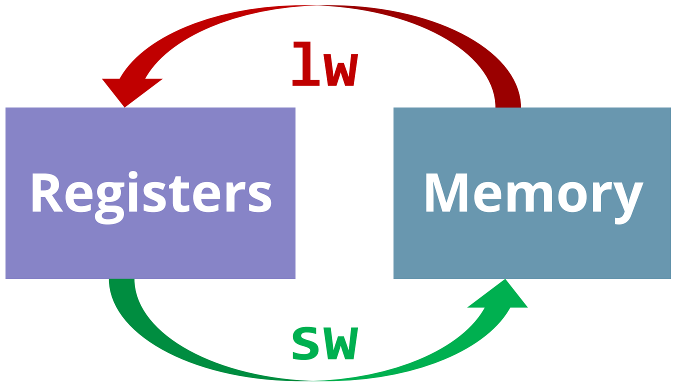
Accessing Memory in MIPS

MIPS ISA: load/store words

- You can load and store entire 32-bit words with **lw** and **sw**
- The instructions look like this (variable names not important):

```
lw t1, x # loads from variable x into t1
sw t1, x # stores from t1 into variable x
```

- In MIPS, **stores are written with the destination on the *right*. !?**
  - Well, you can remember it with this diagram...
  - The memory is "on the right" for both loads and stores



Read, Modify, Write

- you now know enough to increment x!
- first we **load x into a register**
- then...
- and then...

```
lw  t0, x
add t0, t0, 1
sw  t0, x
```

- Let's see what values are in **t0** and memory after this program runs

Variables: That’s really it.

- Variables in asm aren't *THAT* scary
- *Please* don't be afraid of them
- You just gotta remember to store if you wanna change em

- And remember… you are the CPU:
  - Loads: Read from memory
  - Stores: Write to memory

Smaller Values

When your 32-bit cup doth not overfloweth… wait, no, that’s not right.

MIPS ISA: load/store bytes/half-words

- some values are tiny
- to load/store **bytes**, we use **lb/sb**
- to load/store 16-bit (**half-word**) values, we use **lh/sh**
- These mostly look and work just like **lw/sw,** like:

```
lb t0, tiny # loads a byte into t0
sb t0, tiny # stores a byte into tiny
```

- I said mostly… recall: how big are registers?
  - So, what should go in those extra 16/24 bits then?
    - ???

Can I Get an Extension?

- Sometimes you need to *widen* a number with fewer bits to more
- **zero extension** is easy: **put 0s at the beginning.**

1001_2 → *to 8 bits* → $0000\ 1001_2$

- But there are also **signed numbers** which we didn't talk about yet… hmm

Signed Numbers (sign-magnitude)

- Seems like a good time to think about "negative" values.
  - These are numbers that have nothing good to say.

- Binary numbers have bits which are either 0 or 1.
  - Well, yeah...

- So what if we used one bit to designate "positive" or "negative"
  - Called **sign-magnitude** encoding:

```
10100010 = -34

00010110 =  22 (normal)
```

Signed Numbers (problems)

$$\underbrace{\color{red}{1}0000000}_{} = \underbrace{\color{red}{-}0}_{}$$

$$\underbrace{\color{red}{0}0000000}_{} = \underbrace{0}_{}$$

- Waaaaait a second.
  - What is negative zero???

- This encoding allows two different zeros.
  - This means we can represent how many different values (8-bit)?
    - 2^8 – 1 (minus the one redundant value) = 255 (-127 … 0 … 127)

- Sign-magnitude is a little naïve… let's try a different approach…

Signed Numbers (1's Complement)

- Let's borrow a technique from accounting and mechanical calculators: **flip the dang bits**.

11010100 = -00101011 = -43

00100110 = 00100110 = 38

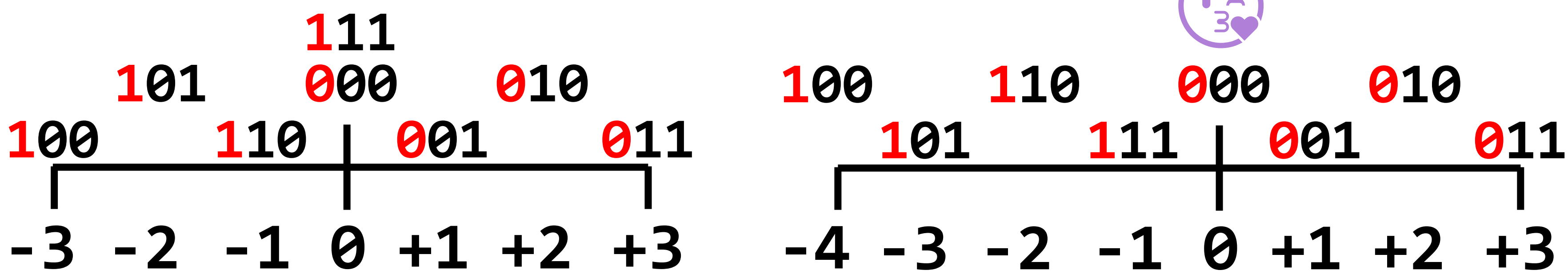
00000000 = 00000000 = 0

11111111 = -00000000 = -0 😠

- OH COME ON (actually, this *is* better because math is easier)
  - But this is really *isn't* used that much.

Signed Numbers (2's Complement)

- This one, I promise, is juuuuust right.
  - But it's a little strange!
- We'll just make SURE there is only one zero:



1's Complement   **2's Complement**

- So, we flip the bits… (1's complement) and add one.
  - Adding one makes sure our -0 is used for -1 instead!
- Sure, it's a little lopsided, but, hey, we get an extra number.
  - But, hmm, but -4 **doesn't have a valid positive number**.
    - That's the trade-off, but it's for the best.

Signed Numbers (2's Complement)

- Let's look at the **same bit patterns** as before:

```
11010100=-00101011=-(43+1)=-44
00100110= 00100110= 38
00000000= 00000000= 0
11111111=-00000000=-(0+1) =-1
```

- **If the MSB is 1**: Flip! Add one!
- **Otherwise**: Do nothing! It's the same!

Signed Numbers (2's Complement)

- What happens when we add zeros to a positive number:

```
        00100110 = 38
0000000000100110 = 38
```

- What happens when we add ones to a negative number:

```
         10100110 = -90
 1111111110100110 =
-0000000001011001 = -90
```

Dang that's cool!

Can I Get an Extension? (Reprise)

- Sometimes you need to *widen* a number with fewer bits to more
- **zero extension** is easy: **put 0s at the beginning.**

1001_2 ➔ *to 8 bits* ➔ $0000\ 1001_2$

- But there are also **signed numbers** which we didn't talk about yet
  - The **top bit (MSB)** of signed numbers determines **the sign (+/-)**
- **sign extension** puts ***copies of the sign bit*** at the beginning

1001_2 ➔ *to 8 bits* ➔ $1111\ 1001_2$

0010_2 ➔ *to 8 bits* ➔ $0000\ 0010_2$

EXPAND

- If you load a **byte...**

`10010000`

31			0
00000000	00000000	00000000	00000000

If the byte is **signed...** what *should* it become?

31			0
11111111	11111111	11111111	10010000

`lb` does sign extension.

If the byte is **unsigned...** what *should* it become?

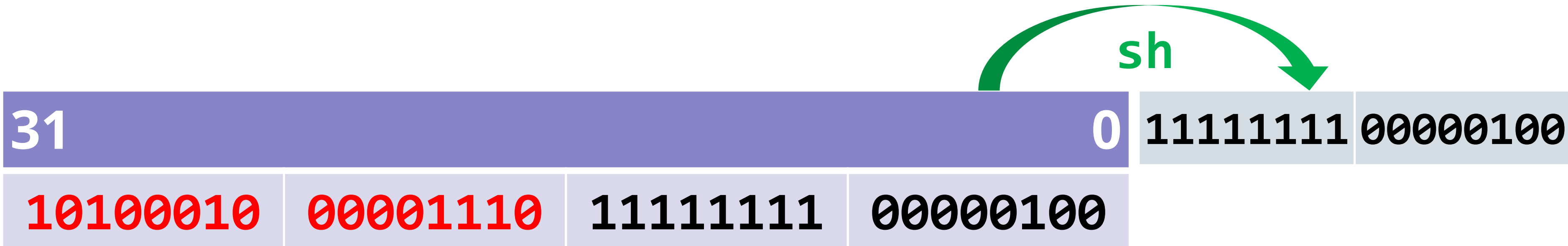
31			0
00000000	00000000	00000000	10010000

`lbu` does zero extension.

lbu (load byte unsigned) is USUALLY what you want to use!

Truncation

- if we go the other way, **the upper part of the value is cut off.**



- the sign issue doesn't exist when storing, cause we're going from a *larger* number of bits to a *smaller* number
  - therefore, **there are no sbu/shu instructions**