CS/COE 0447

Negative Numbers, Arrays and Array Addressing

wilkie (with content borrowed from:

Jarrett Billingsley

Dr. Bruce Childers)

Announcements

- Lab 1 was due! Last Sunday!
  - If you didn't turn it in, first, that was silly of you.
  - Because then you earned a Zero!
  - But the lowest two labs will be dropped (because we all have lives, and sometimes homework doesn't fit into that)
  - Grades *should* be posted on Course Web (Blackboard)

- The first midterm does seem like it is gonna be around where it is in the schedule.
  - I'll let you know what the topics are and some study material two weeks prior.
  - So, don't ask me what is on it now ☺

- The first project will be assigned within the next two weeks
  - It will be due AFTER the first midterm

Lightning Recap

- "Loads" move from? To? "Stores" move from? To?
- You are the CPU!
  - That would be a good Halloween costume, right?

I'm invoking my right to use "Hipster wearing blank t-shirt" stock photos for educational purposes.

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.**

$$1001_2 \rightarrow \textit{to 8 bits} \rightarrow \color{red}{0000}\ 1001_2$$

- 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)

$$\underbrace{\texttt{1}\underbrace{\texttt{0000000}}\ } = \texttt{-}\underbrace{\texttt{0}}$$

$$\underbrace{\texttt{0}\underbrace{\texttt{0000000}}\ } = \underbrace{\texttt{0}}$$

- 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)

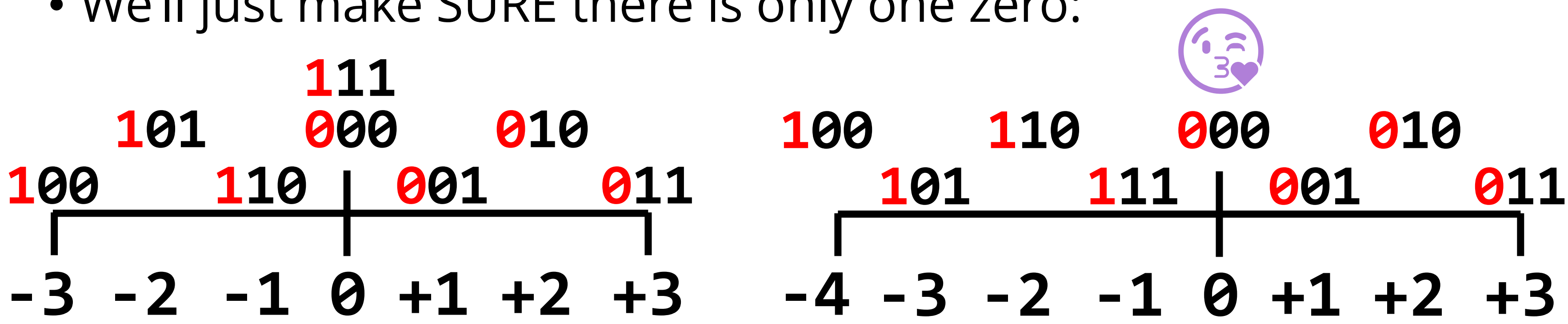
- Let's borrow a technique from accounting and mechanical calculators: **flip the dang bits**.

```
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
000
```

```
        10100110 =  ?
-(01011001+1) =  ?
       -01011010 = -90
```

- What hap… …mber:

```
           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.**

$\texttt{1001}_2$ ➔ *to 8 bits* ➔ $\texttt{0000 1001}_2$

- 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

$\texttt{1001}_2$ ➔ *to 8 bits* ➔ $\texttt{1111 1001}_2$

$\texttt{0010}_2$ ➔ *to 8 bits* ➔ $\texttt{0000 0010}_2$

EXPAND

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

31			0
00000000	00000000	00000000	00000000

10010000

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.**

sh

31 ... 0					
10100010	00001110	11111111	00000100	11111111	00000100

- 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**

Arrays

Rice is great if you're really hungry and want to eat two thousand of something. – Mitch Hedberg, presumably talking about Arrays

Strings, Arrays, etc

- When we wanted to store 4-byte values…
  - We split them up across consecutive bytes
- What about a string?
  - How is a string *represented?*
  - How many bytes is a string?
    - Might be thousands or millions of characters
- *Any* array might be that big too!
- The solution to *storing* it in memory is the same
  - But **how do you *access* these big things**
    - They don't fit in registers!
      - sssssssoooooooooooooooooooooooooooo…

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

What is an Array... ?

- If we did this in C or Java:

```
byte[] arr = {1, 2, 3, 4, 5, ...};
```

- In memory it might look like this
- What memory address is **`arr[0]`** at?
  - What about **`arr[1]`**?
  - What about **`arr[2]`**?
  - What about **`arr[3]`**?
- If an array starts at memory address **A**...
  - ...then item at index **i** is at address...
  - **A + i ?**
    - Only if the array is **holding** individual bytes!
    - **Let's take a deeper look.**

Addr	Val
...	...
F405	06
F404	05
F403	04
F402	03
F401	02
F400	01

Let's take a look at larger Arrays

- So, if we did this: ("**int**" in Java is a 32-bit integer)

```
int[] arr = {1, 2, 3};
```

- In memory it'd look like *this*
  - Why are there all these 0s?
  - What endianness is being used here?
    - Which "end" of the 8-digit hex number is first?
- What memory address is **`arr[1]`** at? **`arr[2]`**?
- If an array starts at memory address **A**...
  - **...and each item is *b* bytes long...**
  - ...then item *i* is at address **A + (*i* × *b*)**
    - On the last slide, *b* happened to be 1

Addr	Val
F40B	00
F40A	00
F409	00
F408	03
F407	00
F406	00
F405	00
F404	02
F403	00
F402	00
F401	00
F400	01

Arrays

- If you wanna **print all the values in an array:**

```
for(int i = 0; i < length; i++)
    print(data[i]);
```

- Let's focus on ^ this bit ^ for now
- **data** is an array of **words,** so how big is each item?
- In this calculation, what is **A** ? ***b*** ? ***i*** ?
- So what's the address calculation?
  - *Address of item i* = `data + (i * 4)`
    - Do you think you could convert that into assembly?
      - Well we haven't done the loop yet...
        - But we'll get to that

Arrays in MIPS

The Practical Application of Two-Thousand Rice

Defining Arrays in MIPS

- First you need to make space for it just like a variable
  - How did we write that variable?

```
myVar: .word 1     # int myVar = 1;
```

- For a small array you can list all the values:

```
myArray: .word 1, 2, 3, 4
```

- But for a big array, that would be annoying ☹
- So you can write:

```
big_array:    .word 0xBEEFC0DE:100
```

- This fills the array with 100 copies of 0xBEEFC0DE
- Notice how similar these look to variables
  - (psst… that's cause there's not really any difference!)

Load Address (la) instruction

- ***Recall:*** *Address of item i* = **`arrayAddress + (i * b)`**
- If the address calculation needs the address of the array...
  - We've gotta get that address into a register right?
    - Can't add something unless it's in registers!
- This is what the **la** instruction does:

```
la t0, myArray # t0 = &myArray[0];
```

- **la** means **load address**
  - ***It doesn't load anything from memory.***
  - Only lw/lh/lhu/lb/lbu load from memory
  - All the other "loads" (li, la) just "put a value in a register"
- What it does: **`t0`** now contains **myArray**'s ***address***

Accessing Arrays: Let's explore!

- We want to **print out** the value in **`myArray[3]`**.
- **What's the address calculation?**
- Now turn that into MIPS
  - Let's come up with the **instructions *first***
    - And *then* decide **which registers to use**
  - How do we put the address of **`myArray`** in a register?
  - Now to translate the math
  - Now we have the address; how do we get the value?
  - How do we print it out?
- If we want to *store* a value into the array…
  - We just use a store instruction instead of a load.

- ***See***: array_ex1.asm

How does the CPU know t0 holds an address?

- ***WHAT DO YOU THINK***
  - ***IT DOESN'T!!***

- **Addresses are just numbers too!!**
- Which means we can **do math** on addresses.
  - Which we did.
  - This is how arrays and strings *work.*
- You can also **have a variable whose value *is an address.***
  - hey
  - 449 students
  - what are these called
  - **pointers**
    - (& is like **la**, * is like **lw/sw**)

Memory Alignment

- We are exploring this a bit in **Lab 2**.
- Let's remove the **mul** instruction
  - "*fetch address not aligned on word boundary*"?
- In MIPS, all memory accesses must be **aligned**
- **Alignment** is just:
  - The address of an *n*-byte value must be a **multiple of *n***
    - so for 4-byte words…
- That's it, that's all, there's nothing more to it.
  - It's not scary.

	Addr	Val
✖	F40B	00
✖	F40A	00
✖	F409	00
○	F408	03
✖	F407	00
✖	F406	00
✖	F405	00
○	F404	02
✖	F403	00
✖	F402	00
✖	F401	00
○	F400	01