CS64 Week 2 Lecture 2

Kyle Dewey

Overview

- MIPS
- `syscall`
- Psuedoinstructions
- Branches
- Memory introduction

MIPS

Why MIPS?

- Relevant in the embedded systems domain
- All processors share the same core concepts as MIPS, just with extra stuff
- ...but most importantly...

It's Simpler

- RISC (reduced instruction set computing)
    - Dozens of instructions as opposed to hundreds
    - Lack of redundant instructions or special cases
- Five stage pipeline versus 24 stages

Code on MIPS

Original

```
x = 5;
y = 7;
z = x + y;
```

MIPS

```
li $t0, 5
li $t1, 7
add $t2, $t0, $t1
```

Code on MIPS

Original

```
x = 5;
y = 7;
z = x + y;
```

MIPS

```
li $t0, 5
li $t1, 7
add $t2, $t0, $t1
```

load **i**mmediate: put the given value into a register

$t0: **t**emporary register **0**

Code on MIPS

Original

```
x = 5;
y = 7;
z = x + y;
```

MIPS

```
li $t0, 5
li $t1, 7
add $t2, $t0, $t1
```

load **i**mmediate: put the given value into a register

$t1: **t**emporary register **1**

Code on MIPS

Original

```
x = 5;
y = 7;
z = x + y;
```

MIPS

```
li $t0, 5
li $t1, 7
add $t2, $t0, $t1
```

add: add the rightmost registers, putting the result in the first register

$t2: **t**emporary register **2**

Available Registers

- 32 registers in all
- For the moment, we will only consider registers `$t0` - `$t9`

Assembly

- The code that you see below is *MIPS assembly*
- Assembly is *almost* what the machine sees. For the most part, it is a direct translation to binary from here (known as *machine code)*

```
li $t0, 5
li $t1, 7
add $t2, $t0, $t1
```

Workflow

Assembly

```
li $t0, 5
li $t1, 7
add $t2, $t0, $t1
```

Assembler
(analogous to a compiler)

Machine Code

```
001101....
```

Machine Code

- This is what the process actually executes and accepts as input
- Each instruction is represented with 32 bits
- Three different instruction formats; for the moment, we'll only look at the R format

---

```
add $t2, $t0, $t1
```

Instruction Register

?

Registers

$t0: ?
$t1: ?
$t2: ?

Program Counter

?

Memory

?

Arithmetic Logic Unit

?

Instruction Register

```
?
```

Registers

```
$t0:  ?
$t1:  ?
$t2:  ?
```

Program Counter

```
0
```

Memory

```
0: li $t0, 5
4: li $t1, 7
8: add $t2, $t0, $t1
```

Arithmetic Logic Unit

```
?
```

Instruction Register

```
li $t0, 5
```

Registers

```
$t0: ?
$t1: ?
$t2: ?
```

Program Counter

```
0
```

Memory

```
0: li $t0, 5
4: li $t1, 7
8: add $t2, $t0, $t1
```

Arithmetic Logic Unit

```
?
```

Instruction Register

```
li $t0, 5
```

Registers

```
$t0: 5
$t1: ?
$t2: ?
```

Program Counter

```
0
```

Memory

```
0: li $t0, 5
4: li $t1, 7
8: add $t2, $t0, $t1
```

Arithmetic Logic Unit

```
?
```

Instruction Register

```
li $t0, 5
```

Registers

```
$t0: 5
$t1: ?
$t2: ?
```

Program Counter

```
4
```

Memory

```
0: li $t0, 5
4: li $t1, 7
8: add $t2, $t0, $t1
```

Arithmetic Logic Unit

```
0 + 4 = 4
```

Instruction Register

```
li $t1, 7
```

Registers

```
$t0: 5
$t1: ?
$t2: ?
```

Program Counter

```
4
```

Memory

```
0: li $t0, 5
4: li $t1, 7
8: add $t2, $t0, $t1
```

Arithmetic Logic Unit

```
?
```

Instruction Register

```
li $t1, 7
```

Registers

```
$t0: 5
$t1: 7
$t2: ?
```

Program Counter

```
4
```

Memory

```
0: li $t0, 5
4: li $t1, 7
8: add $t2, $t0, $t1
```

Arithmetic Logic Unit

```
?
```

Instruction Register

```
li $t1, 7
```

Registers

```
$t0: 5
$t1: 7
$t2: ?
```

Program Counter

```
8
```

Memory

```
0: li $t0, 5
4: li $t1, 7
8: add $t2, $t0, $t1
```

Arithmetic Logic Unit

```
4 + 4 = 8
```

Instruction Register

```
add $t2, $t0, $t1
```

Registers

```
$t0: 5
$t1: 7
$t2: ?
```

Program Counter

```
8
```

Memory

```
0: li $t0, 5
4: li $t1, 7
8: add $t2, $t0, $t1
```

Arithmetic Logic Unit

```
?
```

Instruction Register

```
add $t2, $t0, $t1
```

Registers

```
$t0:  5
$t1:  7
$t2:  ?
```

Program Counter

```
8
```

Memory

```
0: li $t0, 5
4: li $t1, 7
8: add $t2, $t0, $t1
```

Arithmetic Logic Unit

```
5 + 7 = 12
```

Instruction Register

```
add $t2, $t0, $t1
```

Registers

```
$t0: 5
$t1: 7
$t2: 12
```

Program Counter

```
8
```

Memory

```
0: li $t0, 5
4: li $t1, 7
8: add $t2, $t0, $t1
```

Arithmetic Logic Unit

```
5 + 7 = 12
```

Adding More Functionality

- We need a way to display the result
- What does this entail?

Adding More Functionality

- We need a way to display the result
- What does this entail?
  - Input / output. This entails talking to devices, which the operating system handles
  - We need a way to tell the operating system to kick in

Talking to the OS

- We are going to be running on a MIPS emulator, SPIM
- We cannot directly access system libraries (they aren't even in the same machine language)
- How might we print something?

SPIM Routines

- MIPS features a `syscall` instruction, which triggers a *software interrupt*, or *exception*

- Outside of an emulator, these pause the program and tell the OS to check something
- Inside the emulator, it tells the **emulator** to check something

syscall

- So we have the OS/emulator's attention. But how does it know what we want?

syscall

- So we have the OS/emulator's attention. But how does it know what we want?
  - It has access to the registers
  - Put special values in the registers to indicate what you want

(Finally) Printing an Integer

- For SPIM, if register `$v0` contains `1`, then it will print whatever integer is stored in register `$a0`
- Note that `$v0` and `$a0` are distinct from `$t0` - `$t9`

Augmenting with Printing

```
li $t0, 5
li $t1, 7
add $t2, $t0, $t1

li $v0, 1
move $a0, $t2
syscall
```

Exiting

- If you are using SPIM, then you need to say when you are done as well
- How might this be done?

Exiting

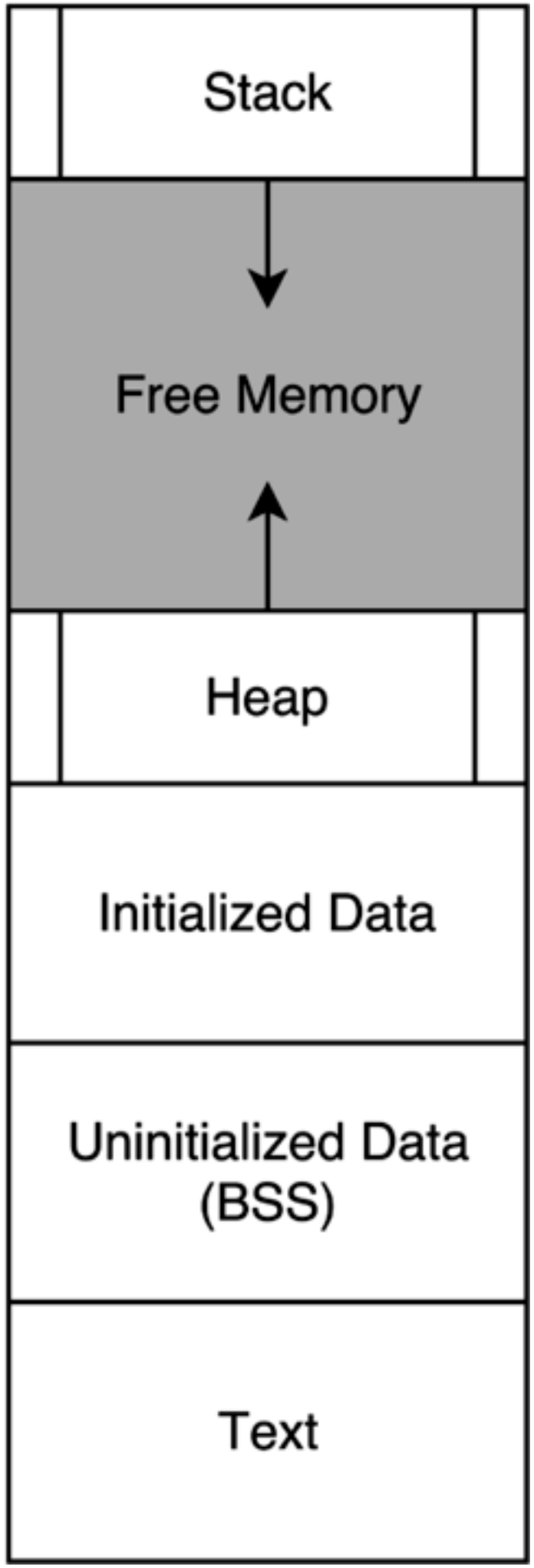
- If you are using SPIM, then you need to say when you are done as well
- How might this be done?
  - `syscall` with a special value in `$v0` (specifically 10 decimal)

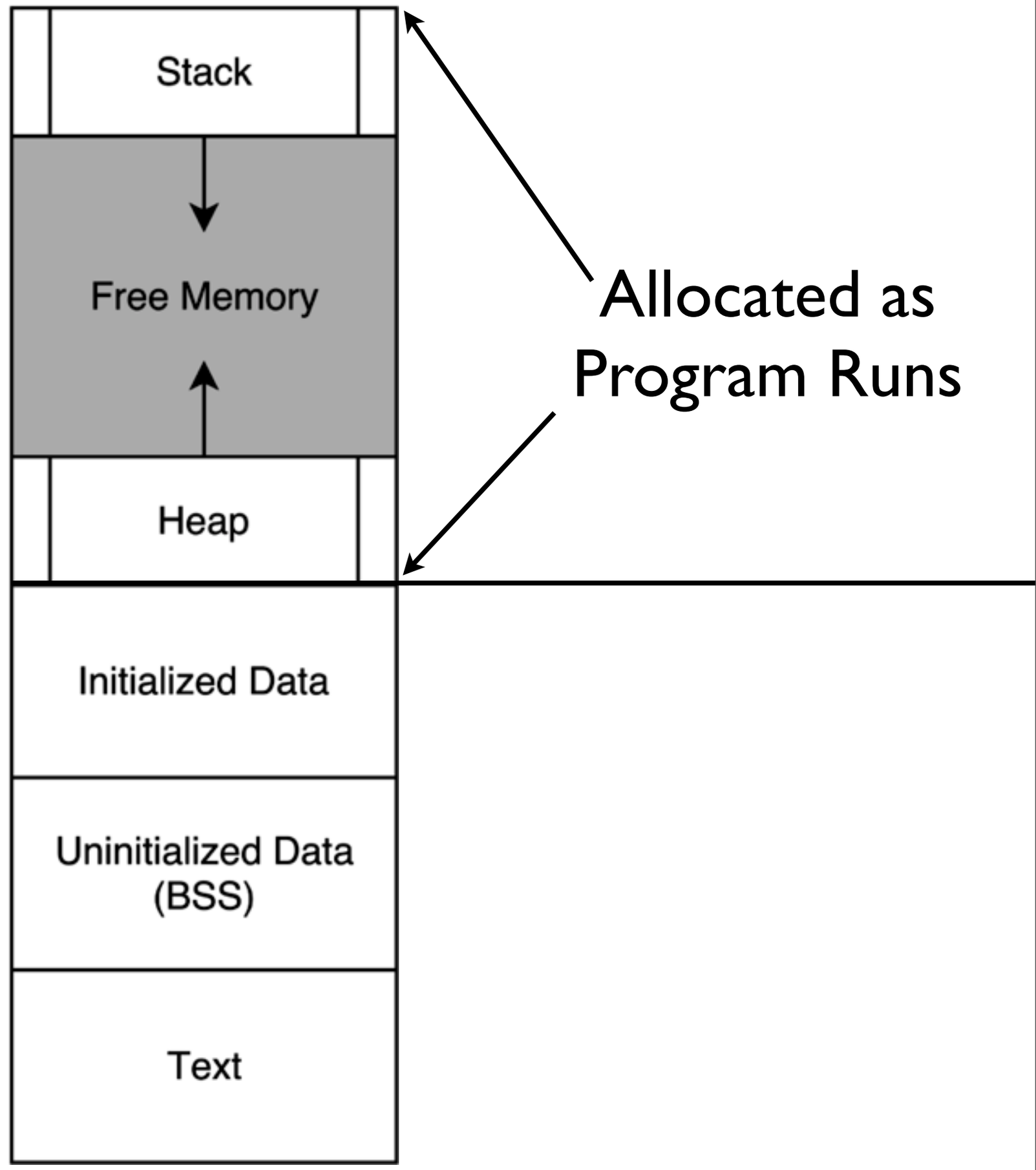
Augmenting with Exiting

```
li $t0, 5
li $t1, 7
add $t2, $t0, $t1

li $v0, 1
move $a0, $t2
syscall

li $v0, 10
syscall
```

Making it a Full Program

- Everything is just a bunch of bits
- We need to tell the assembler which bits should be placed where in memory


Stack
Free Memory
Heap
Initialized Data
Uninitialized Data
(BSS)
Text


Stack
Free Memory
Heap
Initialized Data
Uninitialized Data
(BSS)
Text
Allocated as
Program Runs

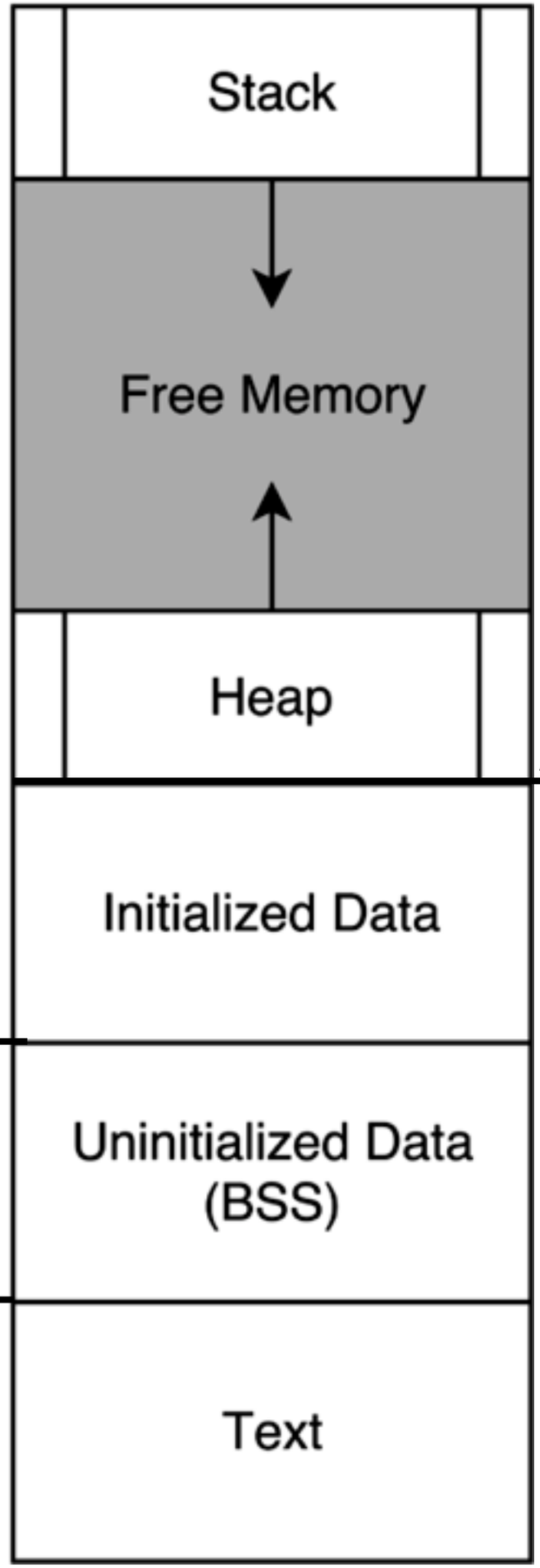
Everything Below is Allocated at Program Load

↓

Constants (e.g., strings)

Mutable Global Variables

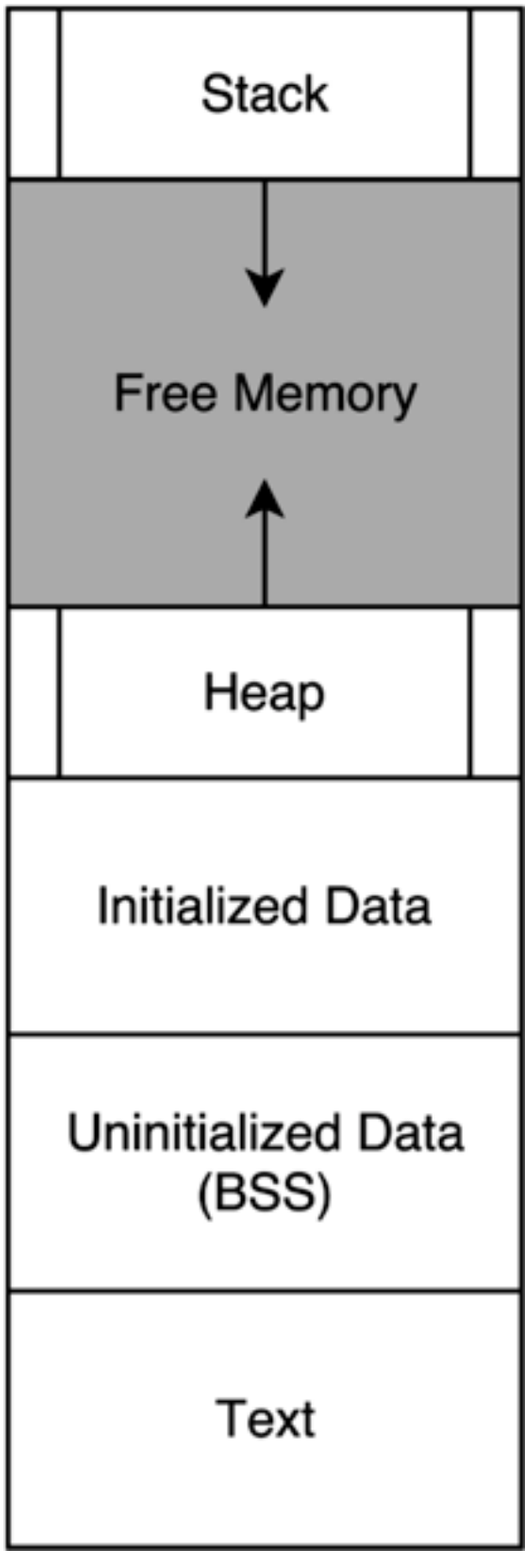
Code



Allocated as Program Runs

Marking Code

- Use a `.text` *directive* to specify code




```
.text

li $t0, 5
li $t1, 7
add $t2, $t0, $t1

li $v0, 1
move $a0, $t2
syscall

li $v0, 10
syscall
```

Running With SPIM

(`add2.asm`)

`move` Instruction

- The move instruction does not actually show up in SPIM
- It is a *pseudoinstruction* which is translated into an actual instruction

Original	Actual
`move $a0, $t2`	`addu $a0, $zero, $t2`

$zero

- Specified like a normal register, but does not behave like a normal register
  - Writes to `$zero` are not saved
  - Reads from `$zero` always return 0

But why?

- Why have move as a pseudoinstruction instead of as an actual instruction?

But why?

- Why have move as a pseudoinstruction instead of as an actual instruction?
  - One less instruction to worry about
  - One design goal of RISC is to cut out redundancy

load intermediate

- The `li` instruction does not actually show up in SPIM
- It is a *pseudoinstruction* which is translated into actual instructions
- Why might `li` work this way?
    - Hint: instructions and registers are both 32 bits long

load intermediate

- The `li` instruction does not actually show up in SPIM
- It is a *pseudoinstruction* which is translated into actual instructions
- Why might `li` work this way?
    - Not enough room in one instruction to fit everything within 32 bits
    - I-type instructions only hold 16 bits

Assembly Coding Strategy

- Best to write it in C-like language, then translate down by hand
- This gets more complex when we get into control structures and memory

```
x = 5;
y = 7;
z = x + y;
```

```
li $t0, 5
li $t1, 7
add $t2, $t0, $t1
```

More Examples

- `swap.asm`
- `negate.asm`
- `mult80.asm`
- `div80.asm`

Control Structure Examples

- `max.asm`
- `sort2.asm`
- `add_0_to_n.asm`

Branches

Conditionals

- Using all the instructions learned so far, how might we code up the following?

```
if (x == 0) {
   printf(“x is zero”);
}
```

Conditionals

- Using all the instructions learned so far, how might we code up the following?

```
if (x == 0) {
  printf("x is zero");
}
```

Answer: We can't (realistically).

Handling Conditionals

- What do we need to implement this?

```
if (x == 0) {
   printf("x is zero");
}
```

Handling Conditionals

- What do we need to implement this?
  - A way to compare numbers
  - A way to *conditionally* execute code

```
if (x == 0) {
  printf("x is zero");
}
```

Relevant Instructions

- Comparing numbers: set-less-than (`slt`)
- Conditional execution: branch-on-equal (`beq`) and branch-on-not-equal (`bne`)
- Do we need anything else?

Relevant Instructions

- Comparing numbers: set-less-than (`slt`)
- Conditional execution: branch-on-equal (`beq`) and branch-on-not-equal (`bne`)
- Do we need anything else?
  - This is sufficient

```
if (x == 0) {
  printf("x is zero");
}
```

```
.data
x_is_zero:
  .asciiz "x is zero"

.text
  bne $t0, $zero, after_print
  li $v0, 4
  la $a0, x_is_zero
  syscall
after_print:
  li $v0, 10
  syscall
```

Loops

- How might we translate the following to assembly?

```
sum = 0;
while (n != 0) {
   sum = sum + n;
   n--;
}
```

Control Structure Examples

- `max.asm`
- `sort2.asm`
- `add_0_to_n.asm`

Memory

Accessing Memory

- Two base instructions: load-word (`lw`) and store-word (`sw`)
- MIPS lacks instructions that do more with memory than access it (e.g., retrieve something from memory and add)
    - Mark of RISC architecture

Global Variables

- Typically, global variables are placed directly in memory, not registers
- Why might this be?

Global Variables

- Typically, global variables are placed directly in memory, not registers
- Why might this be?
    - Not enough registers