System I

RISC-V Assembly

Bo Feng, Lei Wu, Rui Chang

Zhejiang University

Disclaimer

- **Many images and resources used in this lecture are collected from the Internet, and they are used only for the educational purpose. The copyright belong to the original owners, respectively.**

- **Part of slides credit to**
  - **CS 61C, Lisa Yan, Justin Yokota @ UC Berkeley**

Overview

- RISC-V ISA
- RISC-V Assembly Language
  - Variable Definition & Arithmetic Operations
  - Control Flow
  - Function Call
  - From source code to a running program

Overview

- RISC-V ISA
- RISC-V Assembly Language
  - Variable Definition & Arithmetic Operations
  - Control Flow
  - Function Call
  - From source code to a running program

Converting C code to RISC-V

There are only 32 general-purpose registers. What if we have more than 32 variables? Let's translate a program under the following restrictions:

- Only registers x5, x6, and x7 may be modified, and only for intermediate calculations
  - We'll name them `"t0"`, `"t1"`, and `"t2"`, for "temporary register 0-2"
- x2 points to the start of a block of memory that we can use however we want
  - We'll name x2 `"sp",` for "stack pointer"

Converting C code to RISC-V

```c
int a = 5;
char b[] = "string"; // Array will get stored on stack
int c[10];
uint8_t d = b[3];
c[4] = a+d;
c[a] = 20;
```

Converting C code to RISC-V

```
int a = 5;
char b[] = "string";
int c[10];
uint8_t d = b[3];
c[4] = a+d;
c[a] = 20;
```

Step 1: Assign each variable to some offset from sp.

- Exact values don't matter as long as we're consistent

a: 0(sp)

b: 4(sp)

c: 12(sp)

d: 52(sp)

Converting C code to RISC-V

```
int a = 5;
char b[] = "string";
int c[10];
uint8_t d = b[3];
c[4] = a+d;
c[a] = 20;
```

a: 0(sp)

b: 4(sp)

c: 12(sp)

d: 52(sp)

```
li t0 5
sw t0 0(sp)
```

Converting C code to RISC-V

```
int a = 5;
char b[] = "string";
int c[10];
uint8_t d = b[3];
c[4] = a+d;
c[a] = 20;
```

a: 0(sp)

b: 4(sp)

c: 12(sp)

d: 52(sp)

```
li t0 0x73
sb t0 4(sp)
li t0 0x74
sb t0 5(sp)
li t0 0x72
sb t0 6(sp)
li t0 0x69
sb t0 7(sp)
li t0 0x6E
sb t0 8(sp)
li t0 0x67
sb t0 9(sp)
sb x0 10(sp)
```

Converting C code to RISC-V (Better Approach)

```c
int a = 5;
char b[] = "string";
int c[10];
uint8_t d = b[3];
c[4] = a+d;
c[a] = 20;
```

a: 0(sp)

b: 4(sp)

c: 12(sp)

d: 52(sp)

```
li t0 0x69727473
sw t0 4(sp)
li t0 0x0000676E
sw t0 8(sp)
```

Converting C code to RISC-V

```
int a = 5;
char b[] = "string";
int c[10];
uint8_t d = b[3];
c[4] = a+d;
c[a] = 20;
```

a: 0(sp)

b: 4(sp)

c: 12(sp)

d: 52(sp)

Nothing

Converting C code to RISC-V

```
int a = 5;
char b[] = "string";
int c[10];
uint8_t d = b[3];
c[4] = a+d;
c[a] = 20;
```

a: 0(sp)

b: 4(sp)

c: 12(sp)

d: 52(sp)

```
lb t0 7(sp)
sb t0 52(sp)
```

Converting C code to RISC-V

```c
int a = 5;
char b[] = "string";
int c[10];
uint8_t d = b[3];
c[4] = a+d;
c[a] = 20;
```

a: 0(sp)

b: 4(sp)

c: 12(sp)

d: 52(sp)

```
lw t0 0(sp)
lbu t1 52(sp)
add t2 t0 t1
sw t2 28(sp)
```

Converting C code to RISC-V

```
int a = 5;
char b[] = "string";
int c[10];
uint8_t d = b[3];
c[4] = a+d;
c[a] = 20;
```

a: 0(sp)

b: 4(sp)

c: 12(sp)

d: 52(sp)

```
li t0 20
lw t1 0(sp)
sw t0 t1*4+12(sp)
slli t1 t1 2 #t1*=4
addi t1 t1 12
add t1 t1 sp
sw t0 0(t1)
```

Converting C code to RISC-V

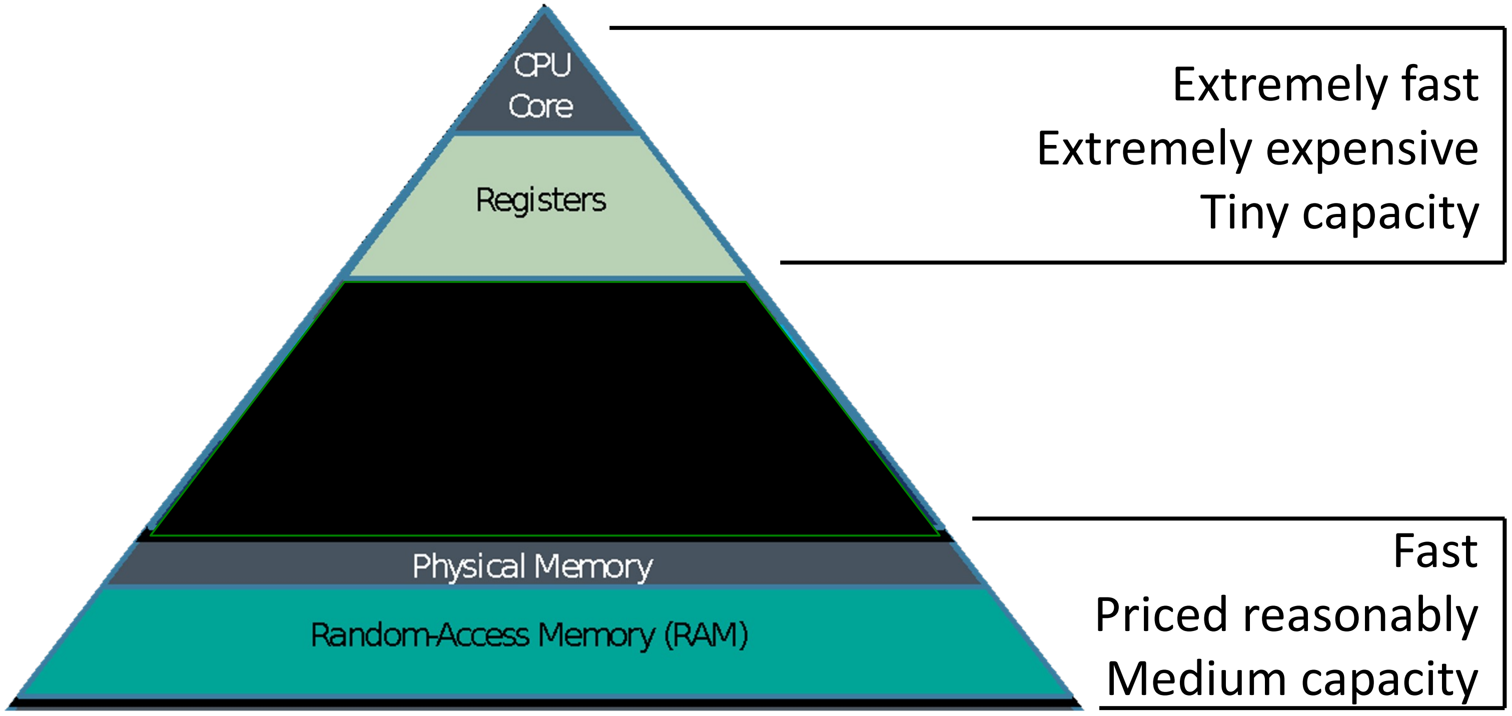
```c
int a = 5;
char b[] = "string";
int c[10];
uint8_t d = b[3];
c[4] = a+d;
c[a] = 20;
```

```
li t0 5
sw t0 0(sp)
li t0 0x69727473
sw t0 4(sp)
li t0 0x0000676E
sw t0 8(sp)
lb t0 7(sp)
sb t0 52(sp)
lw t0 0(sp)
lbu t1 52(sp)
add t2 t0 t1
sw t2 28(sp)
li t0 20
lw t1 0(sp)
slli t1 t1 2 #t1*=4
addi t1 t1 12
add t1 t1 sp
sw t0 0(t1)
```

Why we need so many registers

- As the previous example showed, it's possible to write RISC-V with only a sp and three temporary registers
- Why do we have 32 registers?

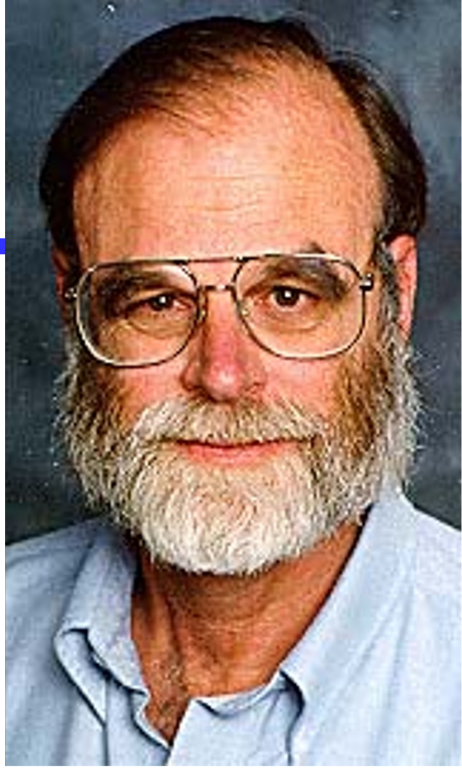
RISC-V Guiding Philosophy



Speed of Registers vs Memory

- Given that
  - Registers: 32 words (128 Bytes)
  - Memory (DRAM): Billions of bytes
    (2 GB to 96 GB on laptop)
- and physics dictates…
  - Smaller is faster
- How much faster are registers than DRAM??
  - About 50-500 times faster!
    (in terms of latency of one access - tens of ns)
    - But subsequent words come every few ns

Jim Gray's Storage Latency Analogy: How Far Away is the Data?



Jim Gray
Turing Award
B.S. Cal 1966
Ph.D. Cal 1969

Sacramento 1.5 hr

100 Memory

1 Registers

My Head 1 min

And in Conclusion…

- Memory is byte-addressable, but **`lw`** and **`sw`** access one word at a time.
- A pointer (used by **`lw`** and **`sw`**) is just a memory address, we can add to it or subtract from it (using offset).
- Memory can be used for variables we can't store in registers, but 100x slower than using registers directly
  - Use loads and stores as infrequently as possible!
- New Instructions:

`lw, sw, lb, sb, lbu`

Overview

- RISC-V ISA
- RISC-V Assembly Language
  - Variable Definition & Arithmetic Operations
  - Control Flow
    - C Control Flow and goto
    - Reducing C with goto
    - RISC-V Control Flow
  - Function Call
  - From source code to a running program

Overview

- RISC-V ISA
- RISC-V Assembly Language
  - Variable Definition & Arithmetic Operations
  - Control Flow
    - C Control Flow and goto
    - Reducing C with goto
    - RISC-V Control Flow
  - Function Call
  - From source code to a running program

RISC-V Guiding Philosophy

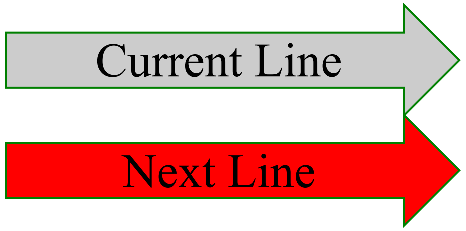
- Goal of assembly: Create a set of instructions such that:
  - Each instruction represents a *single* computation or "step"
    - Ex. `add` adds two registers together, `addi` adds a register and an immediate
  - Every C program can be broken down into instructions
    - Ex.

```
a = b+c+d;   ->     a = b+c;   ->    add x5 x6 x7
                    a = a+d;         add x5 x5 x8
```

  - Each instruction works *in isolation* without depending on context
    - A program's behavior should depend only on memory, registers, and the current line being run
  - RISC: There should be as few unique instructions as possible

Control Flow in C

- In C, we run code one line at a time.
- Most of the time, when we run a line of code, the next line that we will run is the line immediately afterwards
- A few lines make it so that the next line isn't the line immediately afterwards, but somewhere else (we "jump" to another line of code).




```c
int x = 5;
int y = 10;
int z = x+y;
```

Control Flow in C

Lines in C that affect program flow:

- If statements

Current Line

Next Line if cond

Next Line if !cond

```
if(cond) {
    line;
}

line;
```

Control Flow in C

Lines in C that affect program flow:

- If statements
  - If-else statements

Current Line

Next Line if cond

Next Line if !cond

```
if(cond) {
    line;
}
else {
    line;
}
line;
```

Control Flow in C

Lines in C that affect program flow:

- If statements
- While Loops

Next Line

Current Line

```
while(cond) {
    line;
    line;
}
```

Control Flow in C

Lines in C that affect program flow:

- If statements
- While Loops
    - Do-While Loops

Next Line if cond

Current Line

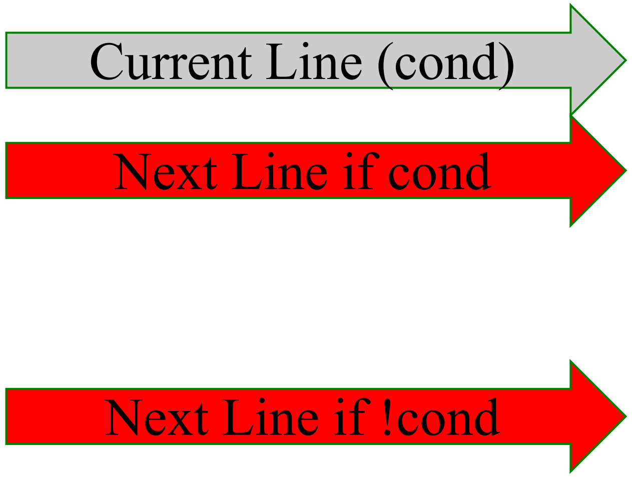
Next Line if !cond

```
do {
    line;
    line;
} while(cond);
line;
```

Control Flow in C

Lines in C that affect program flow:

- If statements
- While Loops
- For Loops




```
for(line;cond;line) {
    line;
    line;
}
line;
```

Control Flow in C

Lines in C that affect program flow:

- If statements
- While Loops
- For Loops
- Break/Continue

Current Line

Next Line

```
while(true) {
    line;
    break;
}
line;
```

Control Flow in C

Lines in C that affect program flow:

- If statements
- While Loops
- For Loops
- Break/Continue
- Function Calls

Next Line

Current Line

```
int foo(n) {
    int a = 5;
    return a+n;
}
...
line;
foo(5);
line;
```

Control Flow in C

Lines in C that affect program flow:

- If statements
- While Loops
- For Loops
- Break/Continue
- Function Calls
  - Both the function call, and the return!
  - Return line depends on which line called foo.
  - Elaborated in the next subsection

Current Line

Next Line?

Next Line?

```
int foo(n) {
    int a = 5;
    return a+n;
}
...
line;
foo(5);
line;
foo(5);
line;
```

New C operator: `goto` and Labels

A **label** is an identifier to a particular line of code

- Doesn't count as a line of code itself; merely "points out" a particular line
- Each label must have a unique name (like variable names)

The `goto` statement changes the next line to be run to the labelled line

- The label can be either before or after the `goto` statement.

Next Line

Current Line

```
Target: line;
        line;
        line;
        goto Target;
        line;
```

goto Example: Handling Mallocs

```
int* a = malloc(sizeof(int)*1000);
int* b = malloc(sizeof(int)*1000000);
int* c = malloc(sizeof(int)*1000000000);
FILE* d = fopen(filename);
```

goto Example: Handling Mallocs

```
int* a = malloc(sizeof(int)*1000);
int* b = malloc(sizeof(int)*1000000);
int* c = malloc(sizeof(int)*1000000000);
FILE* d = fopen(filename);
```

Bad code: malloc can fail (returning NULL), and we should catch that before it causes a segfault

goto Example: Handling Mallocs

```
int* a = malloc(sizeof(int)*1000);
if(a == NULL) allocation_failed();
int* b = malloc(sizeof(int)*1000000);
if(b == NULL) allocation_failed();
int* c = malloc(sizeof(int)*1000000000);
if(c == NULL) allocation_failed();
FILE* d = fopen(filename);
if(d == NULL) allocation_failed();
```

goto Example: Handling Mallocs

```
int* a = malloc(sizeof(int)*1000);
if(a == NULL) allocation_failed();
int* b = malloc(sizeof(int)*1000000);
if(b == NULL) allocation_failed();
int* c = malloc(sizeof(int)*1000000000);
if(c == NULL) allocation_failed();
FILE* d = fopen(filename);
if(d == NULL) allocation_failed();
```

Bad code: leaks memory since a gets allocated but never freed.

goto Example: Handling Mallocs

```
int* a = malloc(sizeof(int)*1000);
if(a == NULL) allocation_failed();
int* b = malloc(sizeof(int)*1000000);
if(b == NULL) {
    free(a);
    allocation_failed();
}
int* c = malloc(sizeof(int)*1000000000);
if(c == NULL) {
    free(b);
    free(a);
    allocation_failed();
}
FILE* d = fopen(filename);
if(d == NULL) {
    free(c);
    free(b);
    free(a);
    allocation_failed();
}
```

goto Example: Handling Mallocs

```
int* a = malloc(sizeof(int)*1000);
if(a == NULL) goto ErrorA;
int* b = malloc(sizeof(int)*1000000);
if(b == NULL) goto ErrorB;
int* c = malloc(sizeof(int)*1000000000);
if(c == NULL) goto ErrorC;
FILE* d = fopen(filename);
if(d == NULL) {
           free(c);
ErrorC:    free(b);
ErrorB:    free(a);
ErrorA:    allocation_failed();
}
```

NEVER USE goto!!!!

- `goto` has a tendency to create completely illegible code
- Generally considered bad practice, except in very specific situations
  - Error handling
  - Jumping out of nested loops
- Even with the above, there are other approaches that don't use `goto`
- Nevertheless, `goto` is useful in that we can create any other control flow statements with just `goto` and *conditional* `goto` statements

Overview

- RISC-V ISA
- RISC-V Assembly Language
  - Variable Definition & Arithmetic Operations
  - Control Flow
    - C Control Flow and goto
    - Reducing C with goto
    - RISC-V Control Flow
  - Function Call
  - From source code to a running program

Reducing C with `goto`: Break

```
while(true) {
    line;
    break;
}
line;
```

Reducing C with `goto`: Break

```
while(true) {
    line;
    goto AfterWhile;
}
AfterWhile: line;
```

Reducing C with `goto`: If

```
if(cond) {
    line;
    line;
} else {
    line;
    line;
}
line;
```

Reducing C with `goto`: If

```
    if(cond) goto IfCase;
    goto ElseCase;
IfCase:
    line;
    line;
    goto AfterIf;
ElseCase:
    line;
    line;
AfterIf: line;
```

Reducing C with `goto`: If without an Else

```
if(!cond) goto AfterIf;
line;
line;
AfterIf: line;
```

Reducing C with `goto`: Do-While

```
do {
    line;
    line;
} while(cond)
line;
```

Reducing C with `goto`: Do-While

```
Loop: line;
line;
if(cond) goto Loop;
line;
```

Reducing C with `goto`: While

```
while(cond) {
    line;
    line;
}
line;
```

Reducing C with `goto`: While

```
Loop: if(!cond) goto AfterLoop;
line;
line;
goto Loop;
AfterLoop: line;
```

Reducing C with `goto`: For

```
for(startline;cond;incline) {
    line;
    line;
}
line;
```

Reducing C with `goto`: For

```
startline;
while(cond) {
    line;
    line;
    incline;
}
line;
```

Reducing C with `goto`: For

```
startline;
Loop: if(!cond) goto AfterLoop
line;
line;
incline;
goto Loop
AfterLoop: line;
```

Overview

- RISC-V ISA
- RISC-V Assembly Language
  - Variable Definition & Arithmetic Operations
  - Control Flow
    - C Control Flow and goto
    - Reducing C with goto
    - RISC-V Control Flow
  - Function Call
  - From source code to a running program

RISC-V Control Flow Operations

- Like in C, RISC-V allows you to write labels to signify particular lines of code
- RISC-V has instructions for both conditional and unconditional jumps:
  - `j Label`
    - Jumps to the specified label
    - Technically a pseudoinstruction; more on this later
  - Branch instructions:
    - General format: `bxx rs1 rs2 Label`
    - Jumps to the specified Label *if the condition is met*
    - If the condition is not met, just moves to the next line

RISC-V Control Flow Operations

List of branch instructions:

- `beq rs1 rs2 Label:` Branch if EQual
- `bne rs1 rs2 Label:` Branch if Not Equal
- `blt rs1 rs2 Label:` Branch if Less Than (signed) (rs1 < rs2)
- `bge rs1 rs2 Label:` Branch if Greater or Equal (signed)
- `bltu rs1 rs2 Label:` Branch if Less Than (unsigned)
- `bgeu rs1 rs2 Label:` Branch if Greater or Equal (unsigned)
- Note that `bgt`, `bgtu`, `ble`, and `bleu` are pseudoinstructions (can make them by reversing inputs of existing instructions)

RISC-V Control Flow Operations: Example

```
int a = 0;
for(int i = 0; i < 10; i++) {
    if(i == 7) {
        break;
    }
    a = a + i;
}
a = a + 50;
```

RISC-V Control Flow Operations: Example

```
int a = 0;
for(int i = 0; i < 10; i++) {
    if(i == 7) goto End;
    a = a + i;
}
End: a = a + 50;
```

RISC-V Control Flow Operations: Example

```
int a = 0;
int i = 0;
Loop: if(i >= 10) goto End;
    if(i == 7) goto End;
    a = a + i;
    i = i + 1;
    goto Loop;
End: a = a + 50;
```

RISC-V Control Flow Operations: Example

```
int a = 0;
int i = 0;
Loop:
    int j = 10;
    if(i >= j) goto End;
    j = 7;
    if(i == j) goto End;
    a = a + i;
    i = i + 1;
    goto Loop;
End: a = a + 50;
```

RISC-V Control Flow Operations: Example

```
       li x10 0          #int a = 0;
       li x5 0           #int i = 0;
Loop:
       li x6 10          #int j = 10;
       bge x5 x6 End     #if(i >= j) goto End;
       li x6 7           #j = 7;
       beq x5 x6 End     #if(i == j) goto End;
       add x10 x10 x5    #a = a + i;
       addi x5 x5 1      #i = i + 1;
       j Loop            #goto Loop;
End:   addi x10 x10 50   #a = a + 50;
```

Overview

- RISC-V ISA
- RISC-V Assembly Language
  - Variable Definition & Arithmetic Operations
  - Control Flow
  - Function Call
    - C Functions
    - RISC-V Memory Model
    - RISC-V Functions
    - Calling Convention
  - From source code to a running program

Overview

- RISC-V ISA
- RISC-V Assembly Language
  - Variable Definition & Arithmetic Operations
  - Control Flow
  - Function Call
    - C Functions
    - RISC-V Memory Model
    - RISC-V Functions
    - Calling Convention
  - From source code to a running program

C Functions

```
int foo(int i) {
    if(i == 0) return 0;
    int a = i + foo(i-1);
    return a;
}
int j = foo(3);
int k = foo(100);
int m = j+k;
```

Two jumps for each function: Jump to the function for the function call, and jump back to the next line of code after the function returns

C Functions

```c
int foo(int i) {
    if(i == 0) return 0;
    int a = i + foo(i-1);
    return a;
}
int j = foo(3);
int k = foo(100);
int m = j+k;
```

Calling a function:

- Set function arguments
- Goto the start of the function

During a function call:

- Keep local scope separate from global scope
- Perform the desired task of the function

Returning from a function:

- Place the return value in a variable that can be accessed
- Goto the line immediately after the function call

Problem with Maintaining Scope

- In RISC-V, local scope doesn't exist; all registers are "kept" throughout the program
- If a function changes register x10, then the global value of x10 will also change
- Can we solve this by just making sure each function uses a different set of registers?
  - No; recursive function calls won't be able to use different registers
- We'll need a way to store variables somewhere that no called function can change

Problem with returning from a function

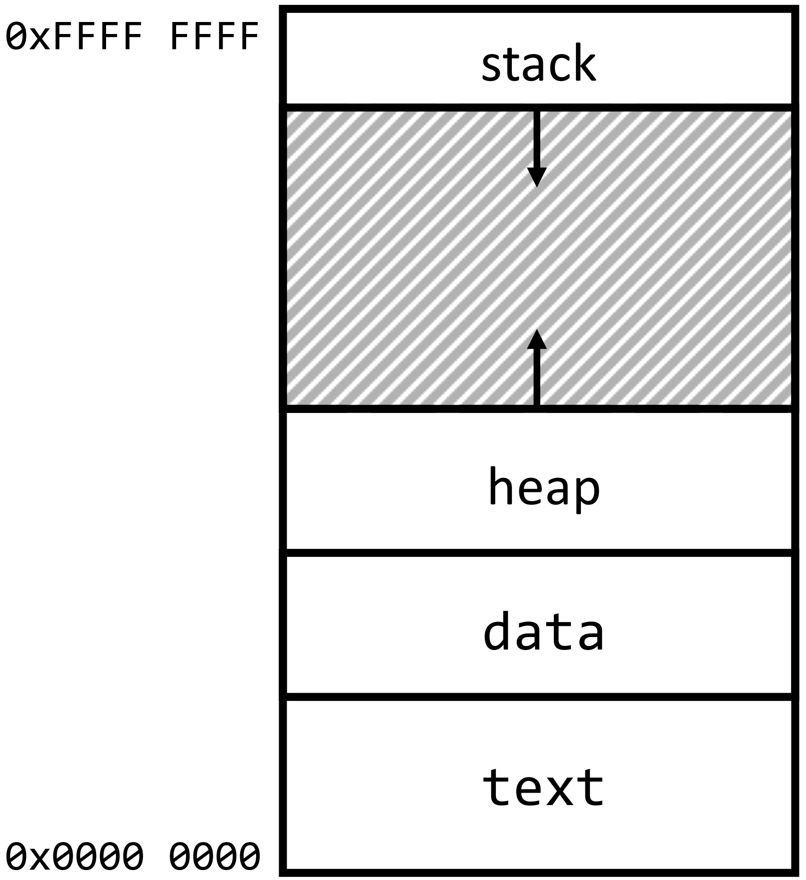
- In C, all `goto`s need to go to a specific label (that can't change)
- However, when returning from a function, we need to jump to different places depending on who called the function (the *return address*).
- This can be solved if we treat the return address as an input to the function
- C doesn't actually let you store a label in a variable/argument, so we won't be able to reduce functions in C using just gotos
- We'll need a way to send in the return address to a function, and jump to that return address when we finish with the function.

Overview

- RISC-V ISA
- RISC-V Assembly Language
  - Variable Definition & Arithmetic Operations
  - Control Flow
  - Function Call
    - C Functions
    - RISC-V Memory Model
    - RISC-V Functions
    - Calling Convention
  - From source code to a running program

Review: C Memory Model

- In C, memory was divided into four segments:
  - Code/Text
  - Static/Data
  - Heap
  - Stack
- RISC-V uses the same memory layout. Today, we'll take a closer look at the text and stack segments



Text

- RISC-V code is also a form of data. This data gets stored in the text section of memory.
- In RISC-V, every (real) instruction is stored as a 32-bit number.
  - Thus, the "next" instruction is always stored 4 bytes after the current instruction.
- A special 33rd register called the **Program Counter** (or PC) keeps track of which line of code is currently being run.

Current Line → 0x0000 0008

Address	Data
`0x0000 0000`	`addi x5 x0 5`
`0x0000 0004`	`xor x5 x6 x6`
`0x0000 0008`	`jal x1 Label`
`0x0000 000C`	`sw x5 8(x2)`
`0x0000 0010`	`beq x0 x0 XX`
`0x0000 0014`	`bne x0 x0 XX`

Register	Value
`PC`	`0x0000 0008`

RISC-V Jump Instructions

- The address of an instruction can be used (along with the PC) to perform the jumps we need for functions.
- `jal rd Label`: Jump And Link
  - Jumps to the given label, but also sets rd to PC+4 (the line after the current line)
  - Ex. If we run the current line, `x1` will be set to `0x0000 000C`, and PC will move to Label

Current Line →

Address	Data
`0x0000 0000`	`addi x5 x0 5`
`0x0000 0004`	`xor x5 x6 x6`
`0x0000 0008`	`jal x1 Label`
`0x0000 000C`	`sw x5 8(x2)`
`0x0000 0010`	`beq x0 x0 XX`
`0x0000 0014`	`bne x0 x0 XX`

Register	Value
`PC`	`0x0000 0008`

RISC-V Jump Instructions

- `jal rd Label`: Jump And Link
  - Jumps to the given label, but also sets rd to PC+4 (the return address)
  - Often used for function calls
- `j Label`: Jump
  - (From last lecture) Jumps to the given label.
    Pseudoinstruction for `jal x0 Label`
  - Used for unconditional jumps (ex. loops)

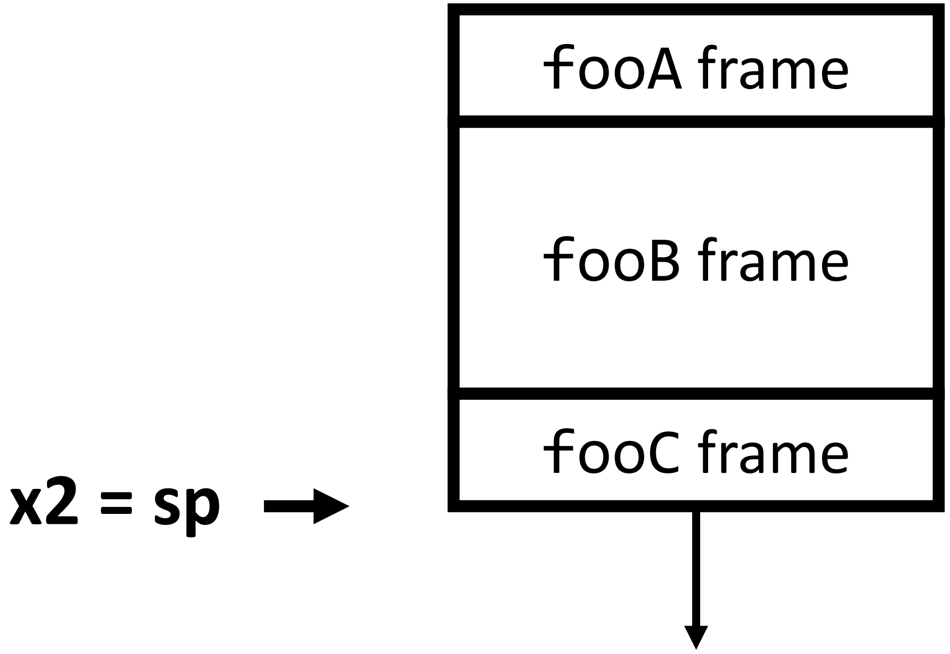
RISC-V Jump Instructions

- `jalr rd rs1 imm`: Jump and Link Register
  - Jumps to the instruction at address rs1+imm, and sets rd to PC+4
  - Less common than other jumps, but used for higher-order functions and *some* function calls (more in the future)
- `jr rs1`: Jump to Register
  - Jumps to the instruction at address rs1
  - Also a pseudoinstruction for `jalr x0 rs1 0`
  - Often used to return from a function

Stack

- In C: Each function call automatically creates a stack frame, with nested calls growing the stack downward.
- In RISC-V: One of our registers (by convention x2, nicknamed sp, or "stack pointer") is set to the bottom of the stack. A function can choose to create a stack frame, by manipulating sp.

```
fooA() { fooB(); }
fooB() { fooC(); }
fooC() { … }
```



RISC-V: Rules for Manipulating the Stack

- Anything above the sp at the start of a function belongs to another function. **You may not modify anything above the sp without permission.**
- Everything below the sp is safe to modify.
  - But anyone else can modify it, so **you can't leave data there** and expect it to stay the same
- By decrementing the sp, we can allocate as much space as we need for our function, that we can use however we want.
- After finishing a function call, **the sp must be set to its value from before the function call**

Address	Data
`0xFFFF FF0C`	
`0xFFFF FF08`	
`0xFFFF FF04`	
`0xFFFF FF00`	
`0xFFFF FEFC`	
`0xFFFF FEF8`	

Register	Value
`sp`	`0xFFFF FF04`

Manual Stack Manipulation: Example

```
fooB:
addi sp sp -8
...
jal x1 fooC
...
addi sp sp 8
jr ra
```

Current Line

Space that we aren't allowed to change

By convention, stack addresses written from greatest to smallest

Address	Data
0xFFFF FF0C	0x12345678
0xFFFF FF08	0x9ABCDEF0
0xFFFF FF04	0x00000000
0xFFFF FF00	0xABADCAFE
0xFFFF FEFC	0x4F639DAB
0xFFFF FEF8	0x14857642

Register	Value
sp	0xFFFF FF04

Manual Stack Manipulation: Example

```
fooB:
addi sp sp -8
...
jal x1 fooC
...
addi sp sp 8
jr ra
```

Current Line

Space that we can change however we want

Address	Data
0xFFFF FF0C	0x12345678
0xFFFF FF08	0x9ABCDEF0
0xFFFF FF04	0x00000000
0xFFFF FF00	0xABADCAFE
0xFFFF FEFC	0x4F639DAB
0xFFFF FEF8	0x14857642

Register	Value
sp	0xFFFF FEFC

Manual Stack Manipulation: Example

```
fooB:
addi sp sp -8
...
jal x1 fooC
...
addi sp sp 8
jr ra
```

Current Line

Space that fooC isn't allowed to change (guaranteed to stay the same)

Address	Data
0xFFFF FF0C	0x12345678
0xFFFF FF08	0x9ABCDEF0
0xFFFF FF04	0x00000000
0xFFFF FF00	0xDEADBEEF
0xFFFF FEFC	0xC561CCCC
0xFFFF FEF8	0x14857642

Register	Value
sp	0xFFFF FEFC

Manual Stack Manipulation: Example

```
fooB:
addi sp sp -8
...
jal x1 fooC
...
addi sp sp 8
jr ra
```

Current Line

Address	Data
0xFFFF FF0C	0x12345678
0xFFFF FF08	0x9ABCDEF0
0xFFFF FF04	0x00000000
0xFFFF FF00	0xDEADBEEF
0xFFFF FEFC	0xC561CCCC
0xFFFF FEF8	0xF00CF00C

After fooC, sp shouldn't be different

Register	Value
sp	0xFFFF FEFC

Manual Stack Manipulation: Example

```
fooB:
addi sp sp -8
...
jal x1 fooC
...
addi sp sp 8
jr ra
```

Current Line

Address	Data
0xFFFF FF0C	0x12345678
0xFFFF FF08	0x9ABCDEF0
0xFFFF FF04	0x00000000
0xFFFF FF00	0xDEADBEEF
0xFFFF FEFC	0xC561CCCC
0xFFFF FEF8	0xF00CF00C

sp needs to be restored to its original value

Register	Value
sp	0xFFFF FEFC

Manual Stack Manipulation: Example

```
fooB:
addi sp sp -8
...
jal x1 fooC
...
addi sp sp 8
jr ra
```

Current Line

Data will eventually be overwritten by another stack frame

Address	Data
0xFFFF FF0C	0x12345678
0xFFFF FF08	0x9ABCDEF0
0xFFFF FF04	0x00000000
0xFFFF FF00	0xDEADBEEF
0xFFFF FEFC	0xC561CCCC
0xFFFF FEF8	0xF00CF00C

Register	Value
sp	0xFFFF FF04

Overview

- RISC-V ISA
- RISC-V Assembly Language
  - Variable Definition & Arithmetic Operations
  - Control Flow
  - Function Call
    - C Functions
    - RISC-V Memory Model
    - RISC-V Functions
    - Calling Convention
  - From source code to a running program

Converting a C function into RISC-V

```
int foo(int i) {
    if(i == 0) return 0;
    int a = i + foo(i-1);
    return a;
}
int j = foo(3);
int k = foo(100);
int m = j+k;
```

Step 1: Define how foo plans to use registers

Inputs:

- i: x10
    - We'll call this register "a0" for "argument"
- Return Address: x1
    - We'll call this register "ra"

Output: x10

- Yes, we'll reuse a0 for the return value

Converting a C function into RISC-V

```c
int foo(int i) {
    if(i == 0) return 0;
    int a = i + foo(i-1);
    return a;
}
int j = foo(3);
int k = foo(100);
int m = j+k;
```

Step 1: Define how foo plans to use registers

Stack Pointer: x2

- Nicknamed "sp"

Register that will NOT be changed by foo: x8, x9

- We can still use these registers, as long as they get restored by the end of the function
- We'll call these registers "s0" and "s1" for "saved"

Converting a C function into RISC-V

```
int foo(int i) {
    if(i == 0) return 0;
    int a = i + foo(i-1);
    return a;
}
int j = foo(3);
int k = foo(100);
int m = j+k;
```

Step 1: Define how foo plans to use registers
Registers that *may* be changed by our function call:
x5

- Since foo can change this, anything that calls foo shouldn't save important data in this register
- We'll call this register "t0" for "temporary"

Converting a C function into RISC-V

```
int foo(int i) {
    if(i == 0) return 0;
    int a = i + foo(i-1);
    return a;
}
int j = foo(3);
int k = foo(100);
int m = j+k;
```

Step 1: Define how foo plans to use registers

Register	Role in foo
`x10 = a0`	i, return value
`x1  = ra`	return address
`x2  = sp`	stack pointer
`x8  = s0`	Saved Register
`x9  = s1`	Saved Register
`x5  = t0`	Temporary

Converting a C function into RISC-V

```
int foo(int i) {
    ...
}
int j = foo(3);
int k = foo(100);
int m = j+k;
```

Register	Role in foo
`x10 = a0`	i, return value
`x1  = ra`	return address
`x2  = sp`	stack pointer
`x8  = s0`	Saved Register
`x9  = s1`	Saved Register
`x5  = t0`	Temporary

Converting a C function into RISC-V

```
# int foo(int i) {
#     ...
# }
li a0 3     # int j = foo(3);
jal ra foo # call foo
mv s0 a0   # mv rd rs1 sets rd = rs1
# int k = foo(100);
# int m = j+k;
```

Register	Role in foo
`x10 = a0`	i, return value
`x1  = ra`	return address
`x2  = sp`	stack pointer
`x8  = s0`	Saved Register
`x9  = s1`	Saved Register
`x5  = t0`	Temporary

Converting a C function into RISC-V

```
# int foo(int i) {
#     ...
# }
li a0 3     # int j = foo(3);
jal ra foo # call foo
mv s0 a0   # mv rd rs1 sets rd = rs1
li a0 100  # int k = foo(100);
jal ra foo # call foo
mv s1 a0   # Saves return value in s1
# int m = j+k;
```

Register	Role in foo
`x10 = a0`	i, return value
`x1  = ra`	return address
`x2  = sp`	stack pointer
`x8  = s0`	Saved Register
`x9  = s1`	Saved Register
`x5  = t0`	Temporary

Converting a C function into RISC-V

```
# int foo(int i) {
#     ...
# }
li a0 3       # int j = foo(3);
jal ra foo    # call foo
mv s0 a0      # mv rd rs1 sets rd = rs1
li a0 100     # int k = foo(100);
jal ra foo    # call foo
mv s1 a0      # Saves return value in s1
add a0 s0 s1  # int m = j+k;
```

Register	Role in foo
`x10 = a0`	i, return value
`x1  = ra`	return address
`x2  = sp`	stack pointer
`x8  = s0`	Saved Register
`x9  = s1`	Saved Register
`x5  = t0`	Temporary

Converting a C function into RISC-V

```
int foo(int i) {
    if(i == 0) return 0;
    int a = i + foo(i-1);
    return a;
}
...
```

Register	Role in foo
`x10 = a0`	i, return value
`x1  = ra`	return address
`x2  = sp`	stack pointer
`x8  = s0`	Saved Register
`x9  = s1`	Saved Register
`x5  = t0`	Temporary

Converting a C function into RISC-V

```
int foo(int i) {
    if(i == 0) return 0;
    int j = i - 1;
    j = foo(j);
    int a = i + j;
    return a;
}
...
```

Register	Role in foo
`x10 = a0`	i, return value
`x1  = ra`	return address
`x2  = sp`	stack pointer
`x8  = s0`	Saved Register
`x9  = s1`	Saved Register
`x5  = t0`	Temporary

Converting a C function into RISC-V

```
int foo(int i) {
    if(i == 0) return 0;
    int j = i - 1;
    j = foo(j);
    int a = i + j;
    return a;
}
...
```

Function call will change ra, a0. Need to save both somewhere

Register	Role in foo
`x10 = a0`	i, return value
`x1 = ra`	return address
`x2 = sp`	stack pointer
`x8 = s0`	Saved Register
`x9 = s1`	Saved Register
`x5 = t0`	Temporary

Converting a C function into RISC-V

```
foo: # int foo(int i)
addi sp sp -4 #Prologue
sw ra 0(sp)   #Prologue
#     if(i == 0) return 0;
#     int j = i - 1;
#     j = foo(j);
#     int a = i + j;
Epilogue:
lw ra 0(sp)   #Epilogue
addi sp sp 4  #Epilogue
#     return a;
...
```

Option 1: Save ra on the stack at the start of the function

Then restore ra from the stack (and restore the stack) at the end.

Register	Role in foo
`x10 = a0`	i, return value
`x1  = ra`	return address
`x2  = sp`	stack pointer
`x8  = s0`	Saved Register
`x9  = s1`	Saved Register
`x5  = t0`	Temporary

Converting a C function into RISC-V

```
foo: # int foo(int i)
addi sp sp -4 #Prologue
sw ra 0(sp)   #Prologue
mv s0 a0      #Move i
#     if(i == 0) return 0;
#     int j = i - 1;
#     j = foo(j);
#     int a = i + j;
Epilogue:
lw ra 0(sp)   #Epilogue
addi sp sp 4  #Epilogue
#     return a;
...
```

Option 2: Save a0 in a saved register so it won't get changed by foo's call

Register	Role in foo
`x10 = a0`	i, return value
`x1  = ra`	return address
`x2  = sp`	stack pointer
`x8  = s0`	Saved Register
`x9  = s1`	Saved Register
`x5  = t0`	Temporary

Converting a C function into RISC-V

```
foo: # int foo(int i)
addi sp sp -8 #Prologue
sw ra 0(sp)   #Prologue
sw s0 4(sp)   #Prologue
mv s0 a0      #Move i
#     if(i == 0) return 0;
#     int j = i - 1;
#     j = foo(j);
#     int a = i + j;
Epilogue:
lw ra 0(sp)   #Epilogue
lw s0 4(sp)   #Epilogue
addi sp sp 8  #Epilogue
#     return a;
...
```

If we modify s0, we need to restore it. Save its old value on the stack, and restore it later.

Register	Role in foo
`x10 = a0`	i, return value
`x1  = ra`	return address
`x2  = sp`	stack pointer
`x8  = s0`	Saved Register
`x9  = s1`	Saved Register
`x5  = t0`	Temporary

Converting a C function into RISC-V

```
foo: # int foo(int i)
addi sp sp -8 #Prologue
sw ra 0(sp)   #Prologue
sw s0 4(sp)   #Prologue
mv s0 a0      #Move i
#     if(i == 0) return 0;
addi t0 s0 -1 #int j = i - 1;
mv a0 t0
jal ra foo    #j = foo(j);
mv t0 a0
add a0 s0 t0  #int a = i + j;
Epilogue:
lw ra 0(sp)   #Epilogue
lw s0 4(sp)   #Epilogue
addi sp sp 8  #Epilogue
jr ra         #return a;
...
```

Use t0 for j, and a0 for a. Due to how foo works, we need to move data to/from a0 for function input/output.

Register	Role in foo
`x10 = a0`	i, return value
`x1  = ra`	return address
`x2  = sp`	stack pointer
`x8  = s0`	Saved Register
`x9  = s1`	Saved Register
`x5  = t0`	Temporary

Converting a C function into RISC-V

```
foo: # int foo(int i)
addi sp sp -8 #Prologue
sw ra 0(sp)   #Prologue
sw s0 4(sp)   #Prologue
mv s0 a0      #Move i
#     if(i == 0) return 0;
addi a0 s0 -1 #int j = i - 1;
jal ra foo    #j = foo(j);
add a0 s0 a0  #int a = i + j;
Epilogue:
lw ra 0(sp)   #Epilogue
lw s0 4(sp)   #Epilogue
addi sp sp 8  #Epilogue
jr ra         #return a;
...
```

Alternative:
Use a0 for j,
and for a.
Saves moving
from t0 to a0
and back in this
particular code.

Register	Role in foo
`x10 = a0`	i, return value
`x1  = ra`	return address
`x2  = sp`	stack pointer
`x8  = s0`	Saved Register
`x9  = s1`	Saved Register
`x5  = t0`	Temporary

What code should go here?

```
foo: # int foo(int i)
addi sp sp -8 #Prologue
sw ra 0(sp)   #Prologue
sw s0 4(sp)   #Prologue
mv s0 a0      #Move i
<CODE>        #if(i == 0) return 0;
Next:
addi a0 s0 -1 #int j = i - 1;
jal ra foo    #j = foo(j);
add a0 s0 a0  #int a = i + j;
Epilogue:
lw ra 0(sp)   #Epilogue
lw s0 4(sp)   #Epilogue
addi sp sp 8  #Epilogue
jr ra         #return a;
...
```

Option A:
`beq s0 x0 Next`
`li a0 0`
`j Epilogue`
	Option B:
	`beq s0 x0 Next`
	`li a0 0`
	`jr ra`

Option C:
`bne s0 x0 Next`
`li a0 0`
`j Epilogue`
	Option D:
	`bne s0 x0 Next`
	`li a0 0`
	`jr ra`

What code should go here?

What code should go here?

CS 61C

```
foo: # int foo(int i)
addi sp sp -8 #Prologue
sw ra 0(sp)   #Prologue
sw s0 4(sp)   #Prologue
mv s0 a0      #Move i
<CODE>        #if(i == 0) return 0;
Next:
addi a0 s0 -1 #int j = i - 1;
jal ra foo    #j = foo(j);
add a0 s0 a0  #int a = i + j;
Epilogue:
lw ra 0(sp)   #Epilogue
lw s0 4(sp)   #Epilogue
addi sp sp 8  #Epilogue
jr ra         #return a;
...
```

Option A:
`beq s0 x0 Next`
`li a0 0`
`j Epilogue`
	Option B:
	`beq s0 x0 Next`
	`li a0 0`
	`jr ra`

Option C:
`bne s0 x0 Next`
`li a0 0`
`j Epilogue`
	Option D:
	`bne s0 x0 Next`
	`li a0 0`
	`jr ra`

Option A	0
Option B	0
Option C	0
Option D	

What code should go here?

```
foo: # int foo(int i)
addi sp sp -8 # Prologue
sw ra 0(sp)   # Prologue
sw s0 4(sp)   # Prologue
mv s0 a0      # Move i
bne s0 x0 Next# if i != 0, skip this
li a0 0       # int a = 0;
j Epilogue    # Go to Epilogue (to restore stack)
Next:
addi a0 s0 -1 # int j = i - 1;
jal ra foo    # j = foo(j);
add a0 s0 a0  # int a = i + j;
Epilogue:
lw ra 0(sp)   # Epilogue
lw s0 4(sp)   # Epilogue
addi sp sp 8  # Epilogue
jr ra         # return a;
...
```

Option A:
`beq s0 x0 Next`
`li a0 0`
`j Epilogue`
	Option B:
	`beq s0 x0 Next`
	`li a0 0`
	`jr ra`

Option C:
`bne s0 x0 Next`
`li a0 0`
`j Epilogue`
	Option D:
	`bne s0 x0 Next`
	`li a0 0`
	`jr ra`

What code should go here?

```
j main
foo:           # int foo(int i)
addi sp sp -8 # Prologue
sw ra 0(sp)   # Prologue
sw s0 4(sp)   # Prologue
mv s0 a0      # Move i
bne s0 x0 Next# if i != 0, skip this
li a0 0       # int a = 0;
j Epilogue    # Go to Epilogue (to restore stack)
Next:
addi a0 s0 -1 # int j = i - 1;
jal ra foo    # j = foo(j);
add a0 s0 a0  # int a = i + j;
Epilogue:
lw ra 0(sp)   # Epilogue
lw s0 4(sp)   # Epilogue
addi sp sp 8  # Epilogue
jr ra         # return a;
main:
li a0 3       # int j = foo(3);
jal ra foo    # call foo
mv s0 a0      # mv rd rs1 sets rd = rs1
li a0 100     # int k = foo(100);
jal ra foo    # call foo
mv s1 a0      # Saves return value in s1
add a0 s0 s1  # int m = j+k;
```

Overview

- RISC-V ISA
- RISC-V Assembly Language
  - Variable Definition & Arithmetic Operations
  - Control Flow
  - Function Call
    - C Functions
    - RISC-V Memory Model
    - RISC-V Functions
    - Calling Convention
  - From source code to a running program

Calling Convention

- When we wrote foo, we chose "roles" for each register based on how we wanted to use them
  - In order for someone else to use foo, they would have to know everything in the table on the right
- We could choose to make one of these tables for every function we need to make
- Better solution: Standardize a set of conventions that everyone agrees to follow.

Register	Role in foo
`x10 = a0`	i, return value
`x1  = ra`	return address
`x2  = sp`	stack pointer
`x8  = s0`	Saved Register
`x9  = s1`	Saved Register
`x5  = t0`	Temporary

Calling Convention

Each register is given a name according to what its role is (no need to memorize the exact mapping):

- `zero`: The x0 register, which always stores 0
- `ra`: x1, which is used to store return addresses
  - Two new pseudoinstructions that explicitly use this:
    - `jal Label -> jal ra Label`
    - `ret       -> jr ra`

#	Name	Description	#	Name	Desc
x0	**zero**	Constant 0	**x16**	**a6**	*Args*
x1	**ra**	*Return Address*	**x17**	**a7**	
x2	**sp**	Stack Pointer	**x18**	**s2**	Saved Registers
x3	**gp**	Global Pointer	**x19**	**s3**	
x4	**tp**	Thread Pointer	**x20**	**s4**	
x5	**t0**	*Temporary Registers*	**x21**	**s5**	
x6	**t1**		**x22**	**s6**	
x7	**t2**		**x23**	**s7**	
x8	**s0**	Saved Registers	**x24**	**s8**	
x9	**s1**		**x25**	**s9**	
x10	**a0**	*Function Arguments or Return Values*	**x26**	**s10**	
x11	**a1**		**x27**	**s11**	
x12	**a2**	*Function Arguments*	**x28**	**t3**	*Temporaries*
x13	**a3**		**x29**	**t4**	
x14	**a4**		**x30**	**t5**	
x15	**a5**		**x31**	**t6**	
Caller saved registers					
Callee saved registers (except **x0**, **gp**, **tp**)					

Calling Convention

Callee Saved registers: Registers that must be restored by the end of a function call (i.e. if you want to use it, the called function needs to save the old value)

- `sp`: The x2 register, which is the stack pointer
- `s0-s11`: Saved registers

#	Name	Description	#	Name	Desc
`x0`	`zero`	Constant 0	`x16`	`a6`	*Args*
`x1`	`ra`	*Return Address*	`x17`	`a7`	
`x2`	`sp`	Stack Pointer	`x18`	`s2`	Saved Registers
`x3`	`gp`	Global Pointer	`x19`	`s3`	
`x4`	`tp`	Thread Pointer	`x20`	`s4`	
`x5`	`t0`	*Temporary Registers*	`x21`	`s5`	
`x6`	`t1`		`x22`	`s6`	
`x7`	`t2`		`x23`	`s7`	
`x8`	`s0`	Saved Registers	`x24`	`s8`	
`x9`	`s1`		`x25`	`s9`	
`x10`	`a0`	*Function Arguments or Return Values*	`x26`	`s10`	
`x11`	`a1`		`x27`	`s11`	
`x12`	`a2`	*Function Arguments*	`x28`	`t3`	*Temporaries*
`x13`	`a3`		`x29`	`t4`	
`x14`	`a4`		`x30`	`t5`	
`x15`	`a5`		`x31`	`t6`	

Caller saved registers

Callee saved registers (except `x0`, `gp`, `tp`)

Calling Convention

Caller Saved registers: Registers that do not need to be restored by a called function (i.e. if you want to save a variable in this register, it needs to be saved somewhere before you call another function)

- `ra`
- `a0-a7`: Registers used for function arguments
    - `a0, a1` also used for function outputs
    - If a function needs more than 8 arguments, can use the stack to store more arguments
- `t0-t6`: Temporary Registers

#	Name	Description	#	Name	Desc
`x0`	`zero`	Constant 0	`x16`	`a6`	*Args*
`x1`	`ra`	*Return Address*	`x17`	`a7`	
`x2`	`sp`	Stack Pointer	`x18`	`s2`	Saved Registers
`x3`	`gp`	Global Pointer	`x19`	`s3`	
`x4`	`tp`	Thread Pointer	`x20`	`s4`	
`x5`	`t0`	*Temporary Registers*	`x21`	`s5`	
`x6`	`t1`		`x22`	`s6`	
`x7`	`t2`		`x23`	`s7`	
`x8`	`s0`	Saved Registers	`x24`	`s8`	
`x9`	`s1`		`x25`	`s9`	
`x10`	`a0`	*Function Arguments or Return Values*	`x26`	`s10`	
`x11`	`a1`		`x27`	`s11`	
`x12`	`a2`	*Function Arguments*	`x28`	`t3`	*Temporaries*
`x13`	`a3`		`x29`	`t4`	
`x14`	`a4`		`x30`	`t5`	
`x15`	`a5`		`x31`	`t6`	

Caller saved registers

Callee saved registers (except `x0`, `gp`, `tp`)

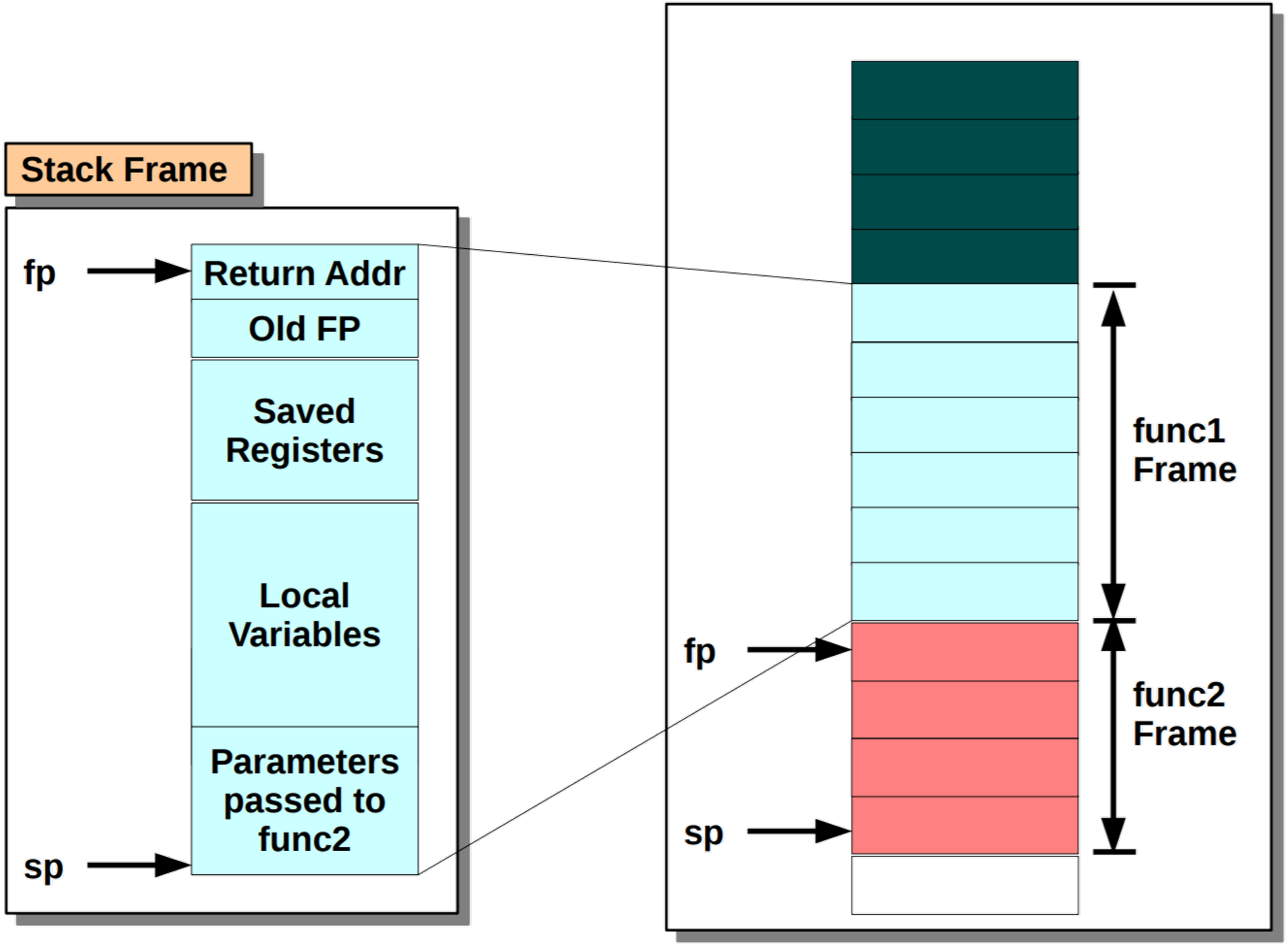
Calling Convention

Other registers: Registers that are out of scope for this class (don't use them!)

- `gp`: The x3 register, used to store a reference to the heap. Also called the "global pointer"
- `tp`: The x4 register, used to store separate stacks for threads (multithreading will be covered in mid-March)

#	Name	Description	#	Name	Desc
`x0`	`zero`	Constant 0	`x16`	`a6`	*Args*
`x1`	`ra`	*Return Address*	`x17`	`a7`	
`x2`	`sp`	Stack Pointer	`x18`	`s2`	Saved Registers
`x3`	`gp`	Global Pointer	`x19`	`s3`	
`x4`	`tp`	Thread Pointer	`x20`	`s4`	
`x5`	`t0`	*Temporary Registers*	`x21`	`s5`	
`x6`	`t1`		`x22`	`s6`	
`x7`	`t2`		`x23`	`s7`	
`x8`	`s0`	Saved Registers	`x24`	`s8`	
`x9`	`s1`		`x25`	`s9`	
`x10`	`a0`	*Function Arguments or Return Values*	`x26`	`s10`	
`x11`	`a1`		`x27`	`s11`	
`x12`	`a2`	*Function Arguments*	`x28`	`t3`	*Temporaries*
`x13`	`a3`		`x29`	`t4`	
`x14`	`a4`		`x30`	`t5`	
`x15`	`a5`		`x31`	`t6`	
Caller saved registers					
Callee saved registers (except `x0`, `gp`, `tp`)					

Stack Frame (revisited)



How To Smash the Stack?

```
vulnerable.c
------------------------------------------------
void main(int argc, char *argv[]) {
  char buffer[512];

  if (argc > 1)
    strcpy(buffer,argv[1]);
}
------------------------------------------------
```

RISC-V Summary

- Over the past four lectures, we've covered almost everything about programming in RISC-V.
- Arithmetic operations allow you to do math with registers
  - Immediate versions for register-constant operations
- Loads/Stores for accessing memory
- Branches for conditionally changing the current line of code
- Jumps for function calls and unconditional jumps
- Only a few remaining instructions left!