CS 61C:
More RISC-V Instructions and *How to Implement Functions*

Administrivia

- Project 1 due tonight!
- Other Assignments Due this Week:
  - Homework 2: 9/22 (Wed)
  - Homework 3: 9/24 (Friday)
  - Lab 3, RISC-V Assembly Language, due 9/24
    - Reminder: lab checkoffs will end *promptly* at 4PM on Fridays!
- Upcoming Assignments:
  - Project 2 being released:
  - Start with lab 4 however:
    It is specifically designed to lead into the project

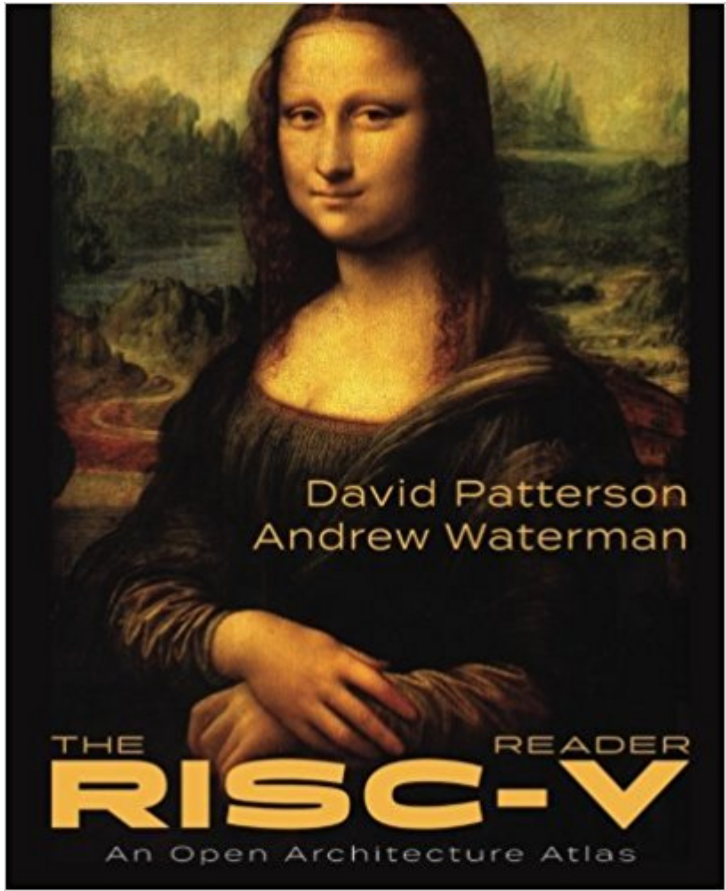
Pedagogical Notes...

- Yes we know this class is an annoying amount of work...
    - We cover many topics, most new to you. Conceptually, not difficult. Keep up!
    - And we have cut stuff from the end of the semester! And one less project than in the past.
- Project 1: *Learn C*
    - We covered just about everything but unions in it...
    - Leads into CS162
- Project 2: *Internalize Assembly*
    - You are going to have to really get the calling convention right for it to work...
    - Leads into CS161 (stack smashing), 162 (context switching), 164 (compilers), etc.

RISC-V book!

- “The RISC-V Reader”, David Patterson, Andrew Waterman

- Available from Amazon
- Print edition `$19.99`
- **Recommended, not required**
- Alternatively, just refer to the ISA documentation directly:
  - https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf



Outline

- RISC-V ISA and C-to-RISC-V Review + new instructions
- Register Conventions
- Function Calls
- And in Conclusion …

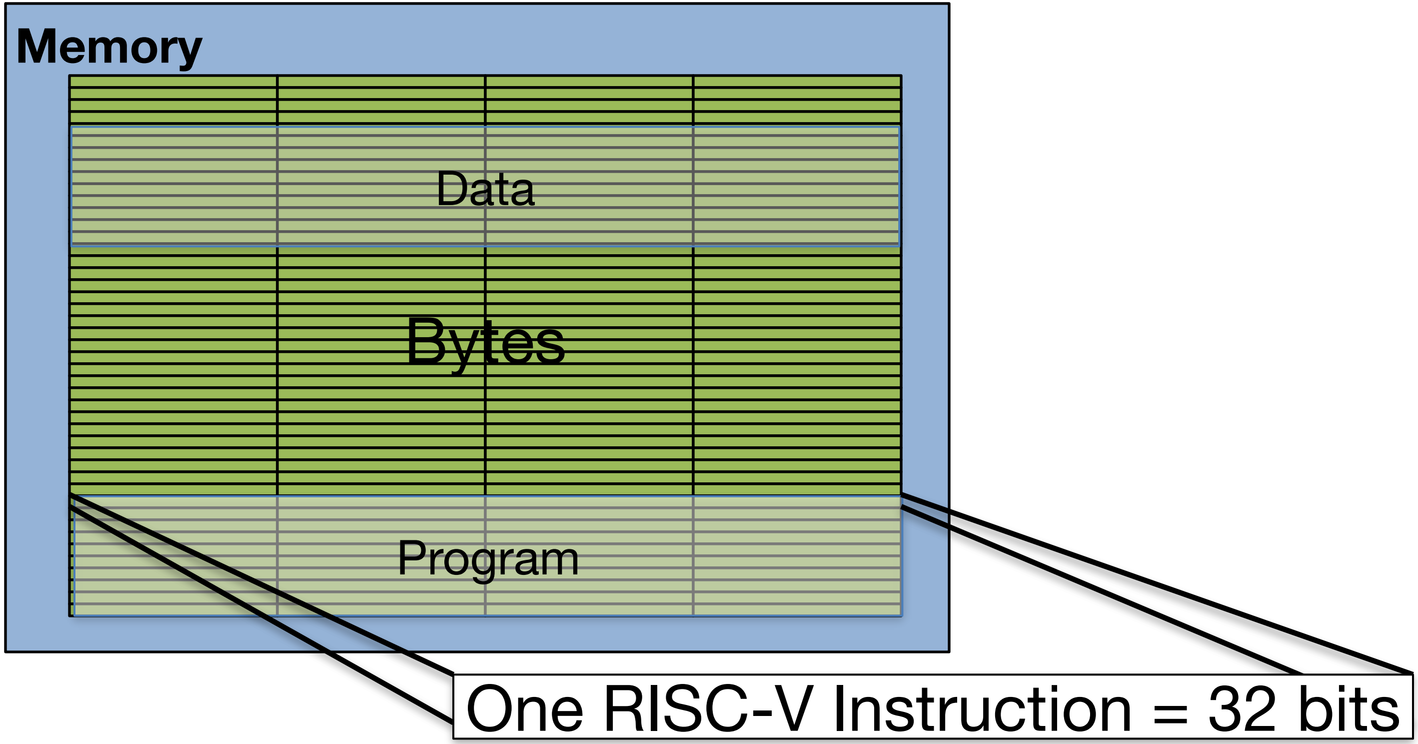
Outline

- RISC-V ISA and C-to-RISC-V Review + new instructions
- Program Execution Overview
- Function Calls
- And in Conclusion …

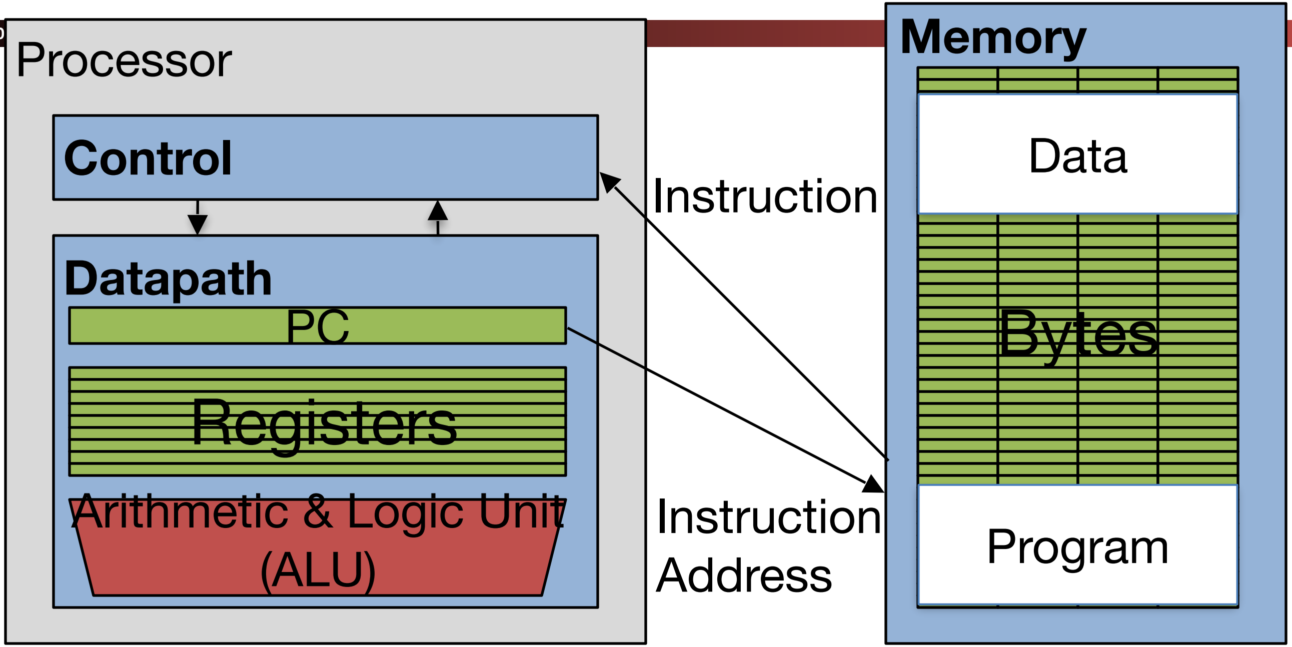
Review From Last Lecture …

- Computer's native operations called instructions. The instruction set defines all the valid instructions.
- RISC-V is example RISC instruction set - used in CS61C
  - Lecture/problems use 32-bit RV32 ISA, book uses 64-bit RV64 ISA
- Rigid format: one operation, two source operands, one destination. So far …
  - **`add,sub`**
  - **`lw,sw,lb,sb`** to move data to/from registers from/to memory
- Simple mappings from arithmetic expressions, array access, in C to RISC-V instructions

How Program is Stored

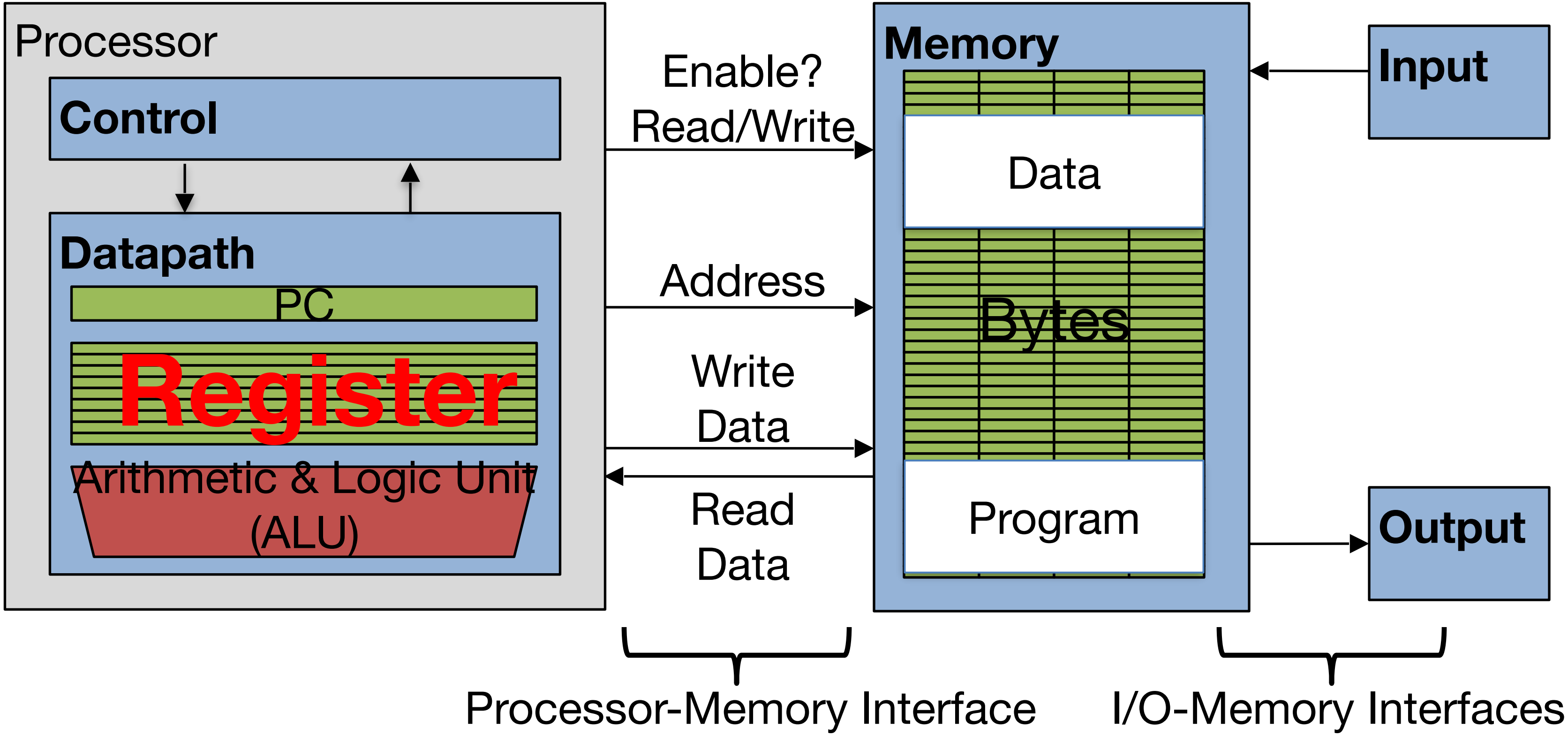


Program Execution

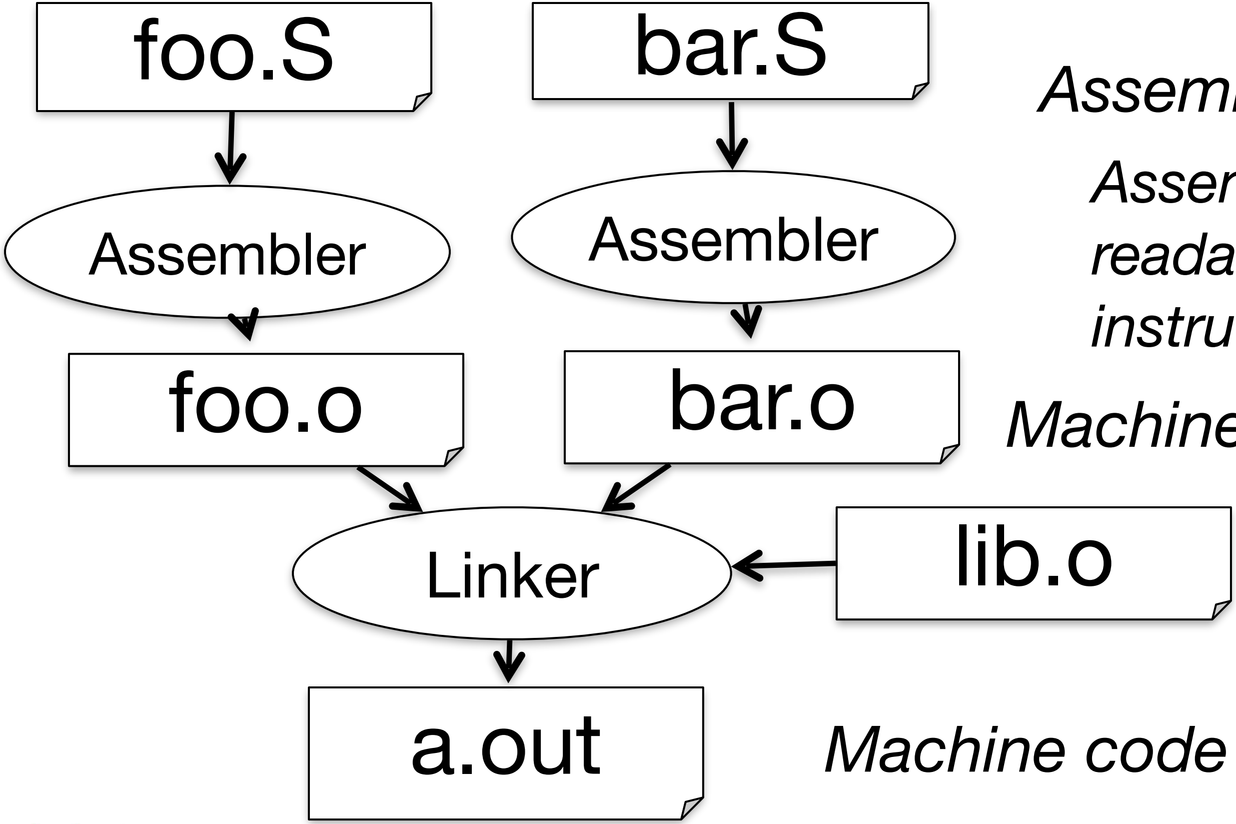


- **PC** (program counter) is special internal register inside processor holding <u>byte</u> address of next instruction to be executed
- Instruction is fetched from memory, then control unit executes instruction using datapath and memory system, and updates program counter (default is <u>add +4 bytes to PC</u>, to move to next sequential instruction)

Recap: Registers live inside the Processor



Assembler to Machine Code (more later in course)



Assembler source files (text)

Assembler converts human-readable assembly code to instruction bit patterns

Machine code object files

Pre-built object file libraries

Machine code executable file

RISC-V *Logical* Instructions

Useful to operate on fields of bits within a word

e.g., characters within a word (8 bits)

Operations to pack /unpack bits into words

Called *logical* operations

Logical operations	C operators	Java operators	RISC-V instructions
Bitwise AND	&	&	**`and`**
Bitwise OR	|	|	**`or`**
Bitwise XOR	^	^	**`xor`**
Shift left logical	<<	<<	**`sll`**
Shift right	>>	>>	**`srl/sra`**

Logical Shifting

- Shift Left Logical:

  `slli x11,x12,2`   # x11 = x12<<2

  - Store in x11 the value from x12 shifted 2 bits to the left (they fall off end), inserting 0's on right; << in C

  Before: $0000\ 0002_{16}$ = $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_2$

  After: $0000\ 000\underline{8}_{16}$ = $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 10\underline{00}_2$

What arithmetic effect does shift left have?

- Shift Right Logical: `srli` is opposite shift; >>
  - Zero bits inserted at left of word, right bits shifted off end

Arithmetic Shifting

- *Shift right arithmetic* (**`srai`**) moves *n* bits to the right (inserting sign bit into empty bits)
- For example, if register x10 contained

  $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110\ 0111_2 = -25_{10}$

- If execute sra x10, x10, 4, result is:

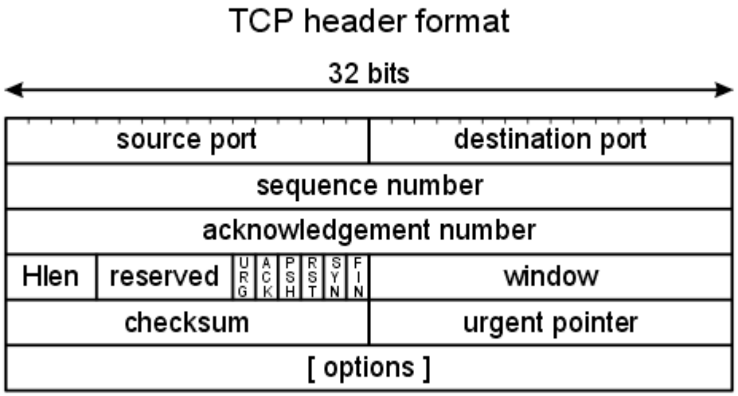
  $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_2 = -2_{10}$

- Unfortunately, this is NOT same as dividing by 2^n
  - Fails for odd negative numbers
  - C arithmetic semantics is that division should round towards 0

Why Shifts and Logical Operations? “Bit Twiddling…”

- Often have to pack/unpack fields
- Eg, in C:
  - int *packet
    packet[0] = sport << 16 | dport
- Becomes (packet in x1, sport in x2, dport in x3)

```
slli x4, x2, 16
or x4, x4, x3
sw x4, 0(x1)
```



Decision Making / Control Flow Instructions

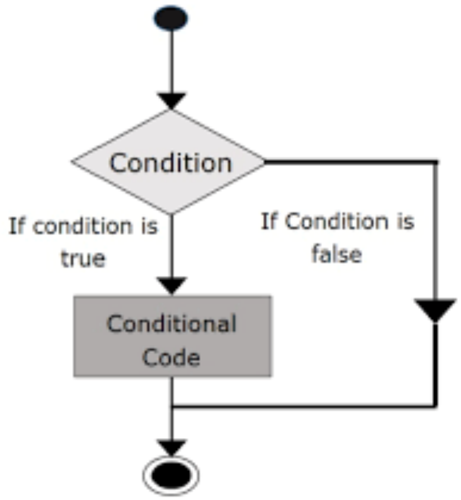
- Need special instructions for *if-else-statements* and *looping* in standard programming languages
- Normal operation on CPU is to execute instructions in sequence
- Based on computation, execute in different order

- RISC-V: *if*-statement instruction is

  `beq register1,register2,L1`

  means: go to instruction labeled L1
  if (value in register1) == (value in register2)
  ....otherwise, go to next instruction

- **`beq`** stands for *branch if equal*
- Other instruction: **`bne`** for *branch if not equal*



Branch – change of control flow

- **Types of Branches:**

- **Conditional Branch** – change control flow depending on outcome of comparison
  - branch *if* equal (**`beq`**) or branch *if not* equal (**`bne`**)
  - Also branch if less than (**`blt`**) and branch if greater than or equal (**`bge`**)

- **Unconditional Branch** – always branch
  - a RISC-V instructions for this call *jumps*

Labels In Assembly Language...

- We commonly see "labels" in the code
  - `foo: add x2 x1 x0`
- The assembler converts these into positions in the code
  - At what address in the code is that label ...
- Labels give control flow instructions, such as jumps and branches, a place to go …
  - e.g. `bne x0 x2 foo`
- The assembler in outputting the code does the necessary calculation so the jump or branch will go to the right place

Example *if* Statement

- Assuming assignments below, compile *if* block

  f → **`x10`** g → **`x11`** h → **`x12`**

  i → **`x13`** j → **`x14`**

```
if (i == j)
   f = g + h;
```

```
    bne x13,x14,done
    add x10,x11,x12
done:
```

Example *if-else* Statement

- Assuming assignments below, compile

  f → **x10** g → **x11** h → **x12** i → **x13** j → **x14**

```
if (i == j)                 bne x13,x14,else
  f = g + h;                add x10,x11,x12
else                        j done #jump
  f = g - h;         else:  sub x10,x11,x12
                     done:
```

Magnitude Compares in RISC-V

- Until now, we've only tested equalities (== and != in C); General programs need to test <, >, >=, <= as well.

  "Branch on Less Than"

  Syntax: `blt reg1,reg2, label`

  Meaning: if (reg1 < reg2) // Registers are signed
  goto label;

- "Branch on Less Than Unsigned"

  Syntax: `bltu reg1,reg2, label`

  Meaning: if (reg1 < reg2) // treat registers as unsigned integers
  goto label;

  "Branch on Greater Than or Equal" (and it's unsigned version) also exist: bge, bgeu

But RISC philosophy…

- RISC-V doesn’t have "branch if greater than” or “branch if less than or equal”
- Instead you can reverse the arguments:

$$A > B \quad \equiv \quad B < A$$
$$A \leq B \quad \equiv \quad B \geq A$$

- The assembler defines pseudo-instructions for your convenience:
  - **`bgt x2 x3 foo`** *becomes*
  - **`blt x3 x2 foo`**

C Loop Mapped to RISC-V Assembly


```
int A[20];
int sum = 0;
for (int i=0; i<20; i++)
    sum += A[i];
```

Loop has 7 instructions

```
# Assume x8 holds pointer to A
# Assign x10=sum, x11=i
add x10, x0, x0 # sum=0
add x11, x0, x0 # i=0
addi x12,x0,20  # x12=20
Loop:
bge x11, x12, exit:
sll x13, x11, 2    # i * 4
add x13, x13, x8   # A + i
lw x13, 0(x13)     # *(A + i)
add x10, x10, x13  # increment sum
addi x11, x11, 1   # i++
j Loop             # Iterate
exit:
```

x10 = 0;
x11 = 0

x11 ≥ x12 ⇒ exit
(i, 20)

x13 = x11 << 2
x11 × 2^2nd
i

x13 = A + x13
& A[i], pointer, i×4

x13 = *(A[i])

x10 = x10 + x13
sum

C Loop Mapped to RISC-V Assembly


```
int A[20];
int sum = 0;
for (int i=0; i<20; i++)
    sum += A[i];
```

Loop now 6 instructions

Slightly optimized

```
# Assume x8 holds base address of A
# Assign x10=sum, x11=i*4
add x10, x0, x0 # sum=0
add x11, x0, x0 # i=0
addi x12,x0,80  # x12=20*4
Loop:
bge x11, x12, exit:
add x13, x11, x8  # A + i
lw x13, 0(x13)    # *(A + i)
add x10, x10, x13 # increment sum
addi x11, x11, 4  # i++
j Loop            # Iterate
exit:
```

More optimizations:


```
int A[20];
int sum = 0;
for (int i=0; i<20; i++)
    sum += A[i];
```

- Inner loop is now 4 instructions rather than 6
  - Directly increment ptr into A array
  - And only 1 branch/jump rather than two
    - Because first time through is always true so can move check to the end!
    - The compiler will often do this automatically for optimization

```
# Assume x8 holds base address of A
# Assign x10=sum
# Assume x11 holds ptr to next A
add  x10, x0, x0   # sum=0
add  x11, x0, x8   # Copy of A
addi x12, x8, 80   # x12=80 + A
loop:
lw   x13, 0(x11)
add  x10, x10, x13
addi x11, x11, 4
blt  x11, x12, loop
```

Conditional Branches Summary

- All are of the form {comparison} {reg1} {reg2} {offset}
  - If the condition is met...
    Add the offset (sign extended + left shifted by 1) to the program counter
  - We write the offset as a label in assembly...
    which the assembler than converts to the number
- Used for ifs, loops, etc...
- **`beq`** Branch Equal
  `bne` Branch Not Equal
  `blt` Branch Less Than (also **`bltu`**)
  `bge` Branch Greater Then or Equal (also **`bgeu`**)
- No "branch-less-than-or-equals" and no "branch-greater-than" ..
  - Instead convert to others by swapped arguments

More on unconditional branches…

- Only two actual instructions
  - jump and link
    `jal rd offset` Jump to PC + Immediate (stores current PC+1 into rd)
  - `jalr rd rs (offset)` Jump to rs + immediate (stores current PC+1 into rd)
    Jump and link register
- Jump And Link
  1. Add the offset to the current address in the program counter (PC), i.e., go to that location
     - The offset is stored as a 20 bit immediate
     - Before adding to the PC it is sign extended and left-shifted ***one*** ***(not two)***
  2. At the same time, store into **`rd`** the value of PC+4 (the ***next*** instruction after the jump)
     - So we know where it came from
  - **`j offset == jal x0 offset`** (yes, jump is a pseudo-instruction in RISC-V)
- Two uses:
  - Unconditional jumps in loops and the like
  - Calling other functions

Jump and Link Register

- Similar to "Jump and Link" except in specification of target
  - Instead of PC + immediate it is **`rs`** + immediate
- Again, if you don't want to record where you jump to…
  - **`jr rs`** == **`jalr x0 rs`**
- Two main uses
  - Returning from functions (which were called using Jump and Link)
  - Calling pointers to function
  - We will see how soon!

Outline

- RISC-V ISA and C-to-RISC-V Review
- Register Conventions
- Function Calls
- And in Conclusion …

Helpful RISC-V Assembler Features

- Symbolic register names
  - E.g., **`a0-a7`** for *argument* registers (**`x10-x17`**)
  - E.g., **`zero`** for **`x0`**
- Pseudo-instructions
  - Shorthand syntax for common assembly idioms
  - E.g., "**`mv rd, rs`**" = "**`addi rd, rs, 0`**"
  - E.g., "**`li rd, 13`**" = "**`addi rd, x0, 13`**"

Let immediate

The "ABI" Conventions & Mnemonic Registers

- The "Application Binary Interface" defines our 'calling convention'
  - How to call other functions
- A critical portion is "what do registers mean by convention"
  - We have 32 registers, but how are they used
- Who is responsible for saving registers?
  - ABI defines a contract: When you call another function, that function promises ***not*** to overwrite certain registers
- We also have more convenient names based on this
  - So going forward, no more x3, x6... type nomenclature

The RISC-V Registers and Convention

Register	ABI Name	Description	Saved By Callee?
`x0`	`zero`	Always Zero	N/A
`x1`	`ra`	Return Address	No
`x2`	`sp`	Stack Pointer	Yes
`x3`	`gp`	Global Pointer	N/A
`x4`	`tp`	Thread Pointer	N/A
`x5-7`	`t0-2`	Temporary	No
`x8`	`s0/fp`	Saved Register/Frame Pointer	Yes
`x9`	`s1`	Saved Register	Yes
`x10-x17`	`a0-7`	Function Arguments/Return Values	No
`x18-27`	`s2-11`	Saved Registers	Yes
`x28-31`	`t3-6`	Temporaries	No

Outline

- RISC-V ISA and C-to-RISC-V Review
- Register Conventions
- Function Calls
- And in Conclusion …

Six Fundamental Steps in Calling a Function

1. Put parameters in a place where function can access them
2. Transfer control to function
3. Acquire (local) storage resources needed for function
4. Perform desired task of the function
5. Put result value in a place where calling code can access it and maybe restore any registers you used
6. Return control to point of origin.

   (Note: a function can be called from several points in a program, including from itself.)

The Calling Convention: A Contract Between Functions…

- The "Calling Convention" in the ABI is the format/usage of registers in a way between the function ***caller*** and function ***callee***, if all functions implement it, everything works out
  - It is effectively a contract between functions
- By convention, registers are classified as one of …
  - ***caller-saved*** 函数调用者已储存
    - The function invoked (the callee) can do whatever it wants to them!
    - Means that the caller can not count on their contents not being destroyed
  - ***callee-saved*** 被调用者储存,
    - The function invoked must restore them before returning (if used)

RISC-V Function Call Conventions

- Registers faster than memory, so use them
- argument
  `a0–a7` (**`x10-x17`**): eight argument registers to pass parameters and two return values (**`a0-a1`**) (caller saved)
  - Any more arguments should be passed on the stack
  - Technically we could return in **`a2-a7`** as well, but we're mostly dealing with C and not python or golang...
- return address
  `ra`: one return address register for return to the point of origin (x1) (caller saved)
- stack point
  `sp`: pointer to the bottom of the stack (callee saved)

More Conventions

- `s0-s11` Saved registers: Preserved across function calls (callee saved)
- `fp` Frame Pointer: Pointer to the top of the call frame
  - Also is `s0`, the first saved register, callee saved
  - Frame pointer can often be omitted by the compiler, but we will sometimes use it because it makes things clearer how functions are translated.
    - It is however critically important in Intel x86 which does a lot more stack manipulations...
      So remember frame pointers when you get to CS161
- `t0-t6` Temporaries: Caller saved

Example - (a “leaf” function - it calls nothing)


```
int Leaf(int g, int h, int i, int j)
{
   int f;
   f = (g + h) – (i + j);
   return f;
}
```

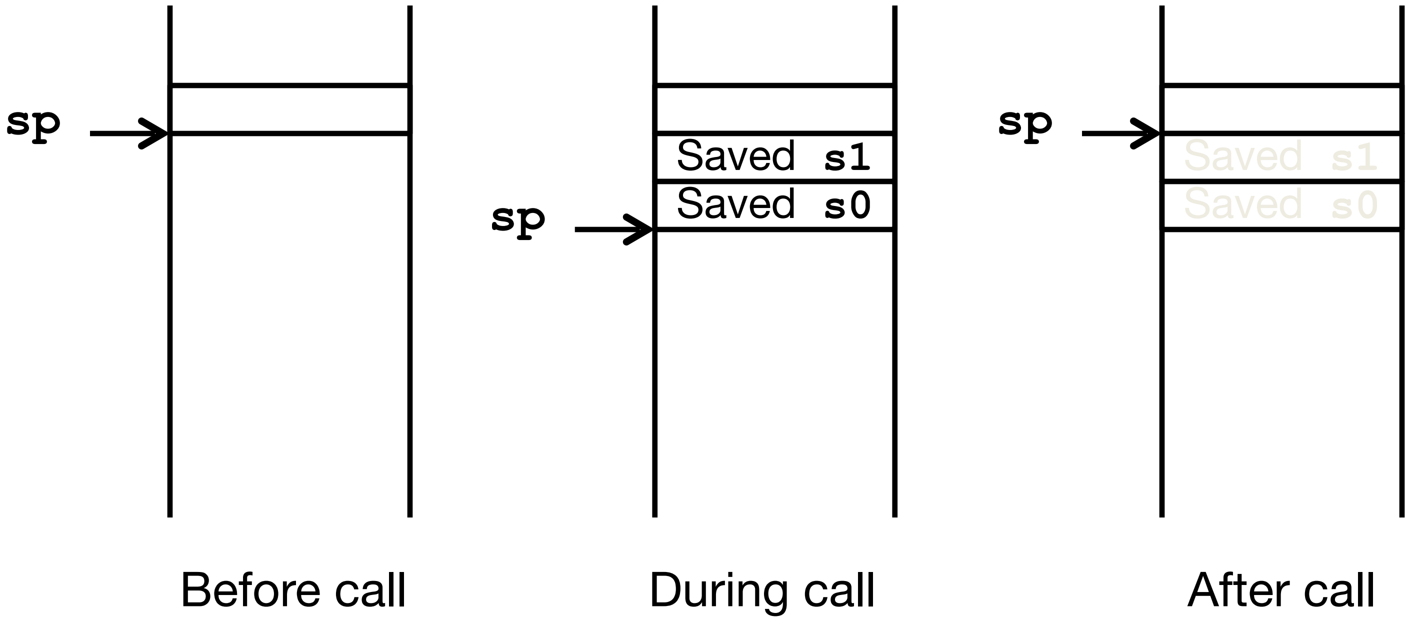
- Parameter variables **g**, **h**, **i**, and **j** in argument registers **a0**, **a1**, **a2**, and **a3**.
- Assume we compute **f** by using **s0** and **s1**
- In real life would probably actually use the **t0**, **t1**, **t2**
- **s0** and **s1** are callee saved, so it's the responsibility of **“leaf”** to save and restore

Where Are Old Register Values Saved to Restore Them After Function Call?

- Need a place to save old values before body of function, restore them when return
- Ideal is *stack*: last-in-first-out queue (e.g., stack of plates)
  - Push: placing data onto stack
  - Pop: removing data from stack
- Stack in memory, so need register to point to it
- **`sp`** is the *stack pointer* in RISC-V (**`x2`**)
- **`sp`** always points to the last used place on the stack
- Convention is grow stack down from high to low addresses
  - Think of it like a stack of plates in the dining commons... If you had a reverse gravity field applied
  - *Push* decrements **`sp`**, *Pop* increments **`sp`**

Stack Before, During, After Function

- Need to save old values of **`s0`** and **`s1`**



RISC-V Code for Leaf()


```
Leaf: addi sp,sp,-8 # adjust stack for 2 items
      sw s1, 4(sp)  # save s1 for use afterwards
      sw s0, 0(sp)  # save s0 for use afterwards

      add s0,a0,a1 # s0 = g + h
      add s1,a2,a3 # s1 = i + j
      sub a0,s0,s1 # return value (g + h) - (i + j)

      lw s0, 0(sp) # restore register s0 for caller
      lw s1, 4(sp) # restore register s1 for caller
      addi sp,sp,8 # adjust stack to delete 2 items
      jr ra        # jump back to calling routine
```

Observations …

- This is a "leaf function": it calls no other function
  1. We didn't need to save **`ra`** (because leaf didn't call any other function and therefore **`ra`** never changed
  2. Instead of **`s0`** and **`s1`** can just use temporary (caller-saved registers) only
- So we could have just as easily used **`t0`** and **`t1`** instead…

```
leaf:
   add t0,a0,a1 # t0 = g + h
   add t1,a2,a3 # t1 = i + j
   sub a0,t0,t1 # return value (g + h) - (i + j)
   ret # ret is shorthand for jalr x0 ra
```

Jump and link register   Jump to ra + immediate

What If a Function Calls a Function?

- Note: this could mean a function calling itself - *recursion*.
- Would clobber (overwrite) the values in `a0-a7` and `ra`
- What is the solution?

Nested Procedures (1/2)


```
int sumSquare(int x, int y) {
  return mult(x,x)+ y;
}
```

- Function called **`sumSquare`** is calling **`mult`**
- So there's a value in **`ra`** that **`sumSquare`** wants to jump back to, but this will be overwritten by the call to **`mult`**

Need to save **`sumSquare`** return address before call to **`mult`**

Nested Procedures (2/2)

- In general, may need to save some registers in addition to **`ra`**.
- Again, use the *stack* for this.
- When a C program is run, there are three important memory areas allocated:
  - Static: Variables declared once per program, cease to exist only after execution completes - e.g., C globals
  - Heap: Variables declared dynamically via **`malloc`**
  - Stack: Space to be used by procedure during execution; this is where we can save register values AND local variables

Optimized Function Convention

To reduce expensive loads and stores from saving (also called "spilling") and restoring registers,
RISC-V function-calling convention divides registers into two categories:

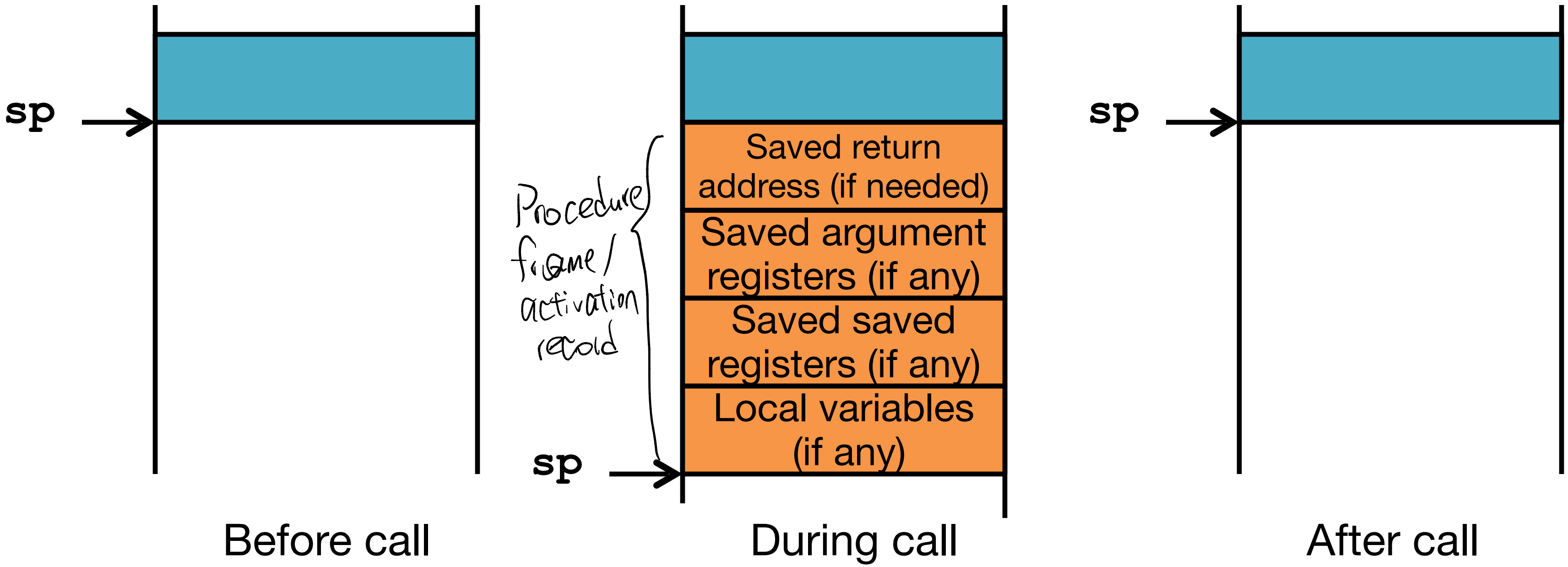
1. Preserved across function call (***callee*** saved)
   - Caller can rely on values being unchanged
   - **`sp`, `gp`, `tp`,** "saved registers" **`s0 - s11`** (**`s0`** is also **`fp`**)
2. Not preserved across function call (***caller*** saved)
   - Caller *cannot* rely on values being unchanged, so if they want to keep them have to save them
   - Argument/return registers **`a0-a7`,`ra`**, "temporary registers" **`t0-t6`**

- Plus two global registers (**`gp`**, **`tp`**) that can be read but shouldn't be changed within a function:
  - Act as pointers to shared and thread-specific global space for global variables

Allocating Space on Stack

- C has two storage classes: automatic and static
  - *Automatic* variables are local to function and discarded when function exits
  - *Static* variables exist across exits from and entries to procedures
- Use stack for automatic (local) variables that aren't in registers
- *Procedure frame* or *activation record*: segment of stack with saved registers and local variables

Stack Before, During, After Function



Using the Stack (1/2)

- So we have a register **sp** which always points to the last used space in the stack
- To use stack, we decrement this pointer by the amount of space we need and then fill it with info
- So, how do we compile this?

```
int sumSquare(int x, int y) {
    return mult(x,x)+ y;
}
```

Using the Stack (2/2)

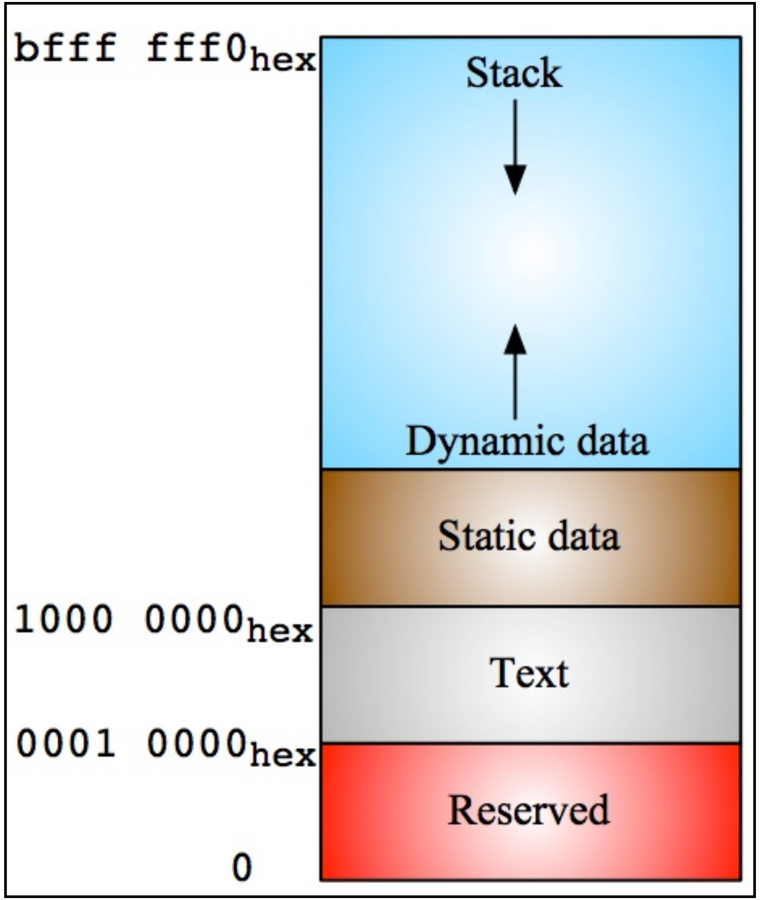

```
int sumSquare(int x, int y) {
   return mult(x,x)+ y; }
```

"push" / "pop"

```
sumSquare:
        addi sp,sp,-8   # reserve space on stack
        sw ra, 4(sp)    # save ret addr
        sw a1, 0(sp)    # save y
        mv a1,a0        # mult(x,x)
        jal mult        # call mult
        lw a1, 0(sp)    # restore y
        add a0,a0,a1    # mult()+y
        lw ra, 4(sp)    # get ret addr
        addi sp,sp,8    # restore stack
        jr ra
mult: ...
```

RV32 Memory Allocation

- RV32 convention (RV64 and RV128 have different memory layouts)
- Stack starts in high memory and grows down
  - Hexadecimal (base 16) : **`bfff_fff0`**$_{hex}$
- RV32 programs (*text segment*) in low end
  - **`0001_0000`**$_{hex}$
- *static data segment* (constants and other static variables) above text for static variables
  - RISC-V convention *global pointer* (**`gp`**) points to static
  - RV32 **`gp`** = **`1000_0000`**$_{hex}$
- *Heap* above static for data structures that grow and shrink ; grows up to high addresses



A Richer Translation Example…

- `struct node {unsigned char c, struct node *next};`
  - c will be at 0, next will be at 4 because of alignment
  - sizeof(struct node) == 8
- `struct node * foo(char c){`

```
    struct node *n;
    if (c < 0) return 0;
    n = malloc(sizeof(struct node));
    n->next = foo(c - 1);
    n->c = c;
    return n;
}
```

So What Will We Need?

- We'll need to save **`ra`**
  - Because we are calling other functions
- We'll need a local variable for c
  - Because we are calling other functions, lets put this in **`s0`**
- We'll need a local variable for n
  - Lets put this in **`s1`**
- So lets form the "preamble" and "postamble"
  - What we always do on entering and leaving the function
  - So we need to save **`ra`**, and the old versions of **`s0`** and **`s1`**

Preamble and Postamble


```
foo:
  addi sp,sp,-12   # Get stack space for 3 registers
  sw s0,0(sp)      # Save s0 (it is callee saved)
  sw s1,4(sp)      # Save s1 (it is callee saved)
  sw ra,8(sp)      # Save ra (it will get overwritten)

{body goes here}   # whole function stuff…

foo_exit:          # Assume return value already in a0
  lw s0,0(sp)      # Restore Registers
  lw s1,4(sp)
  lw ra,8(sp)
  add sp,sp,12     # Restore stack pointer
  ret              # aka.. jalr x0 ra
```

And now the body…


```
  blt a0,x0,foo_true    # if c < 0, jump to foo_true
foo_false:              # this label ends up being ignored but
                        # it is useful documentation
  mv s0,a0              # save c in s0
  li a0,8               # sizeof(struct node) (pseudoinst)
  jal malloc            # call malloc
  mv s1,a0              # save n in s1
  addi a0,s0,-1         # c-1 in a0
  jal foo               # call foo recursively
  sw a0,4(s1)           # write the return value into n->next
  sb s0,0(s1)           # write c into n->c (just a byte)
  mv a0,s1              # return n in a0
  j foo_exit
foo_true:
  add a0,x0,x0          # return 0 in a0
```

We skipped some possible optimizations …

- On the leaf node (c < 0) we didn't need to save `ra` (or even `s0` & `s1` since we don't need to use them)
- We could get away with only one saved register..
  - Save c into `s0`
  - call `malloc`
  - save c into n[0]
  - calc c-1
  - save n in s0
  - recursive call
- For us, our version is good enough.

Outline

- RISC-V ISA and C-to-RISC-V Review
- Program Execution Overview
- Function Call
- Function Call Example
- And in Conclusion …

And in Conclusion …

- Functions called with `jal`, return with `jr ra`.
- The stack is your friend: Use it to save anything you need. Just leave it the way you found it!
- Instructions we know so far…

  Arithmetic: `add, addi, sub`

  Memory: `lw, sw, lb, lbu, sb`

  Decision: `beq, bne, blt, bge`

  Unconditional Branches (Jumps): `j, jal, jr`
- Registers we know so far
  - All of them!
  - `a0-a7` for function arguments, `a0-a1` for return values
  - `sp`, stack pointer, `ra` return address
  - `s0-s11` saved registers
  - `t0-t6` temporaries
  - `zero`