Logical Instructions

RV32 So Far…

- Add/sub

```
add rd, rs1, rs2
sub rd, rs1, rs2
```

- Add immediate

```
addi rd, rs1, imm
```

- Load/store

```
lw  rd,  rs1, imm
lb  rd,  rs1, imm
lbu rd,  rs1, imm
sw  rs1, rs2, imm
sb  rs1, rs2, imm
```

- Branching

```
beq  rs1, rs2, Label
bne  rs1, rs2, Label
bge  rs1, rs2, Label
blt  rs1, rs2, Label
bgeu rs1, rs2, Label
bltu rs1, rs2, Label
j Label
```

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
Bit-by-bit AND	&	&	`and`
Bit-by-bit OR	|	|	`or`
Bit-by-bit XOR	^	^	`xor`
Shift left logical	<<	<<	`sll`
Shift right logical	>>	>>	`srl`

RISC-V Logical Instructions

- Always two variants
  - Register: `and x5, x6, x7 # x5 = x6 & x7`
  - Immediate: `andi x5, x6, 3 # x5 = x6 & 3`
- Used for 'masks'
  - `andi` with `0000 00FF`$_{hex}$ isolates the least significant byte
  - `andi` with `FF00 0000`$_{hex}$ isolates the most significant byte

No NOT in RISC-V

- There is no logical NOT in RISC-V
  - Use **`xor`** with $\mathtt{11111111}_{\mathtt{two}}$
  - Remember - simplicity…

Logical Shifting

- Shift Left Logical (**sll**) and immediate (**slli**):

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

  - Store in **x11** the value from **x12** shifted by 2 bits to the left (they fall off end), inserting 0's on right; << in C.
  - Before: $0000\ 0002_{hex}$
    $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two}$
  - After: $0000\ 000\underline{8}_{hex}$
    $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 10\underline{00}_{two}$
  - What arithmetic effect does shift left have?
- Shift Right: **srl** is opposite shift; **>>**

Arithmetic Shifting

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

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110\ 0111_{two} = -25_{ten}$

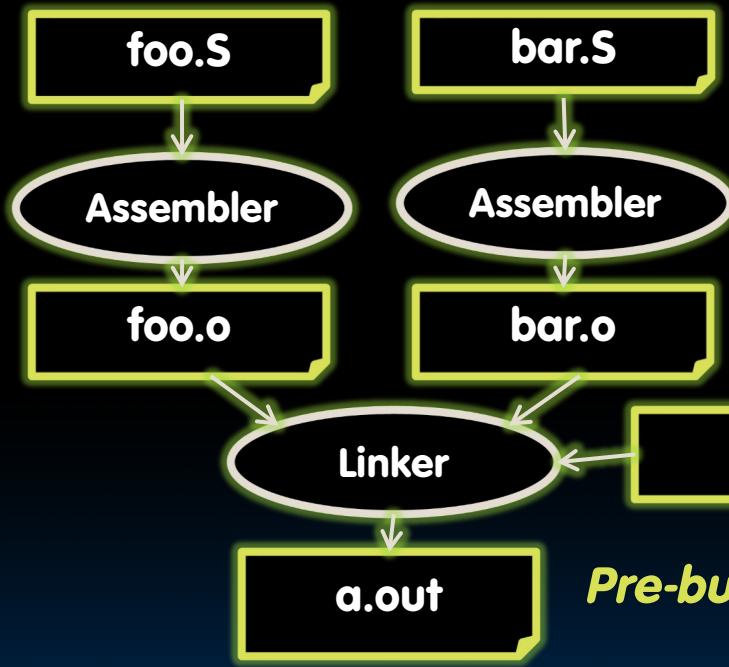
- If execute **`srai x10, x10, 4`**, result is:

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2_{ten}$

- 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

A Bit About Machine Program

Assembler to Machine Code (More Later in Course)



Assembler source files (text)

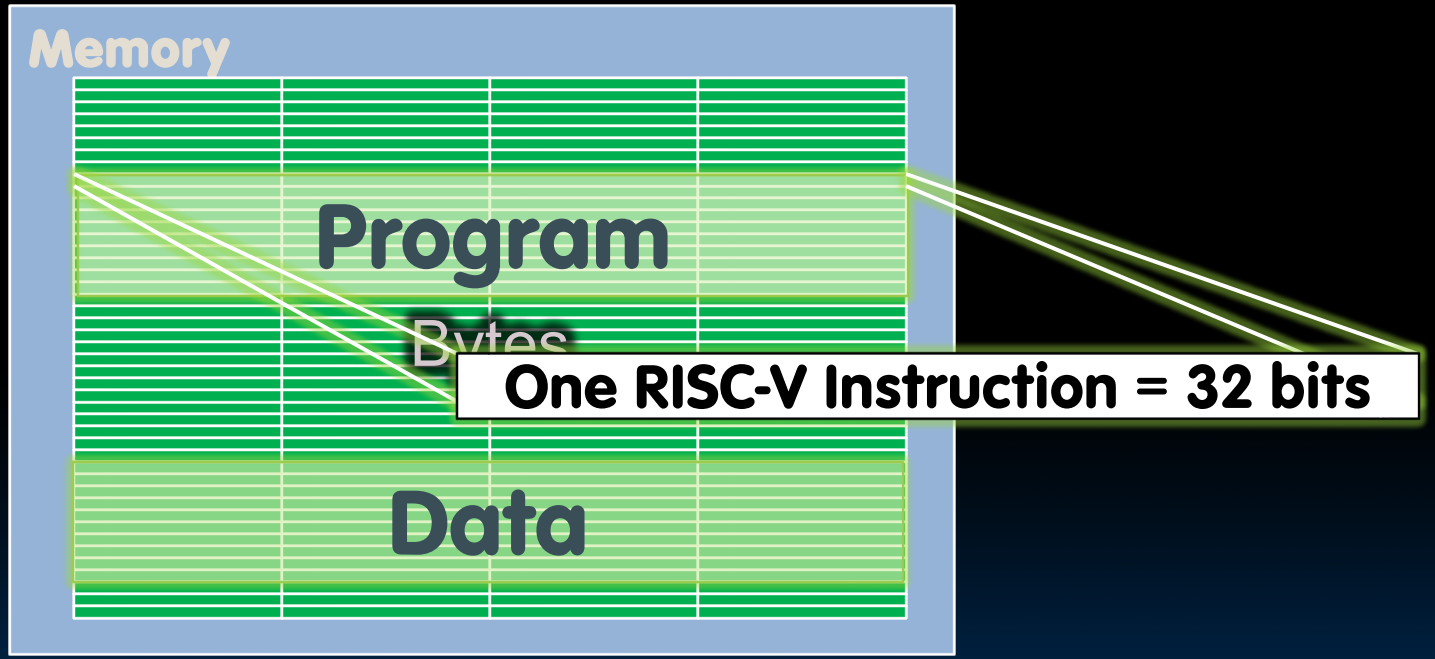
Assembler converts human-readable assembly code to instruction bit patterns

Machine code object files

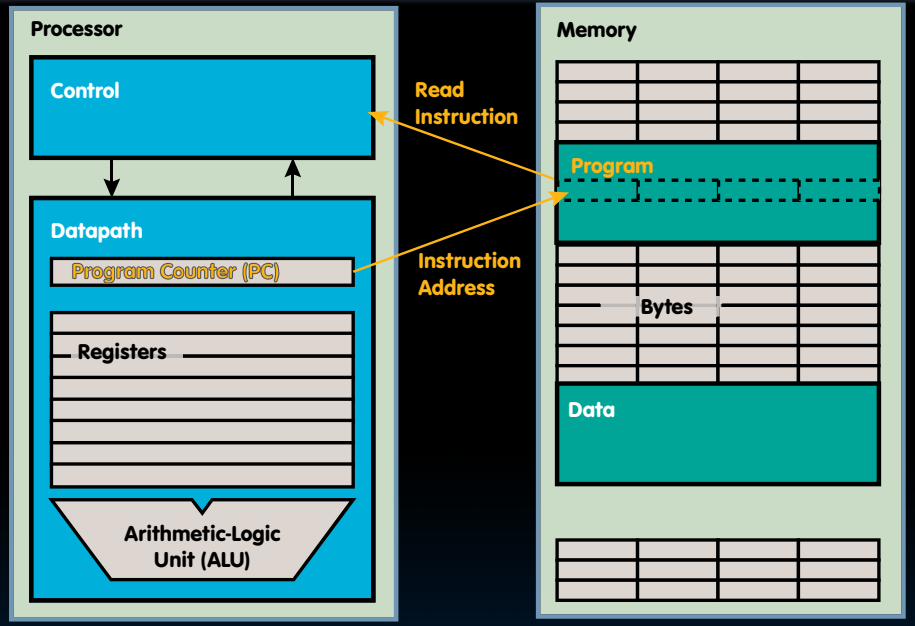
Pre-built object file libraries

Machine code executable file

How Program is Stored



Program Execution



- **PC** (program counter) is a register internal to the processor that holds byte address of next instruction to be executed

- Instruction is fetched from memory, then control unit executes instruction using datapath and memory system, and updates PC
  (default add +4 bytes to PC, to move to next sequential instruction; branches, jumps alter)

Helpful RISC-V Assembler Features

- Symbolic register names
  - E.g., `a0-a7` for argument registers (`x10-x17`) for function calls
  - 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`
  - E.g., `nop` = `addi x0, x0, 0`

RISC-V Function Calls

C Functions

```
main() {
   int i,j,k,m;
   ...
   i = mult(j,k); ...
   m = mult(i,i); ...
}

/* really dumb mult function */
int mult (int mcand, int mlier){
   int product = 0;
   while (mlier > 0)  {
     product = product + mcand;
     mlier = mlier -1; }
   return product;
   }
```

What information must compiler/programmer keep track of?

What instructions can accomplish this?

Six Fundamental Steps in Calling a Function

1. Put arguments 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 return value in a place where calling code can access it and restore any registers you used; release local storage
6. Return control to point of origin, since a function can be called from several points in a program

RISC-V Function Call Conventions

- Registers faster than memory, so use them
- **`a0–a7 (x10–x17)`:** eight *argument* registers to pass parameters and two return values (**`a0–a1`**)
- **`ra`**: one *return address* register to return to the point of origin (**`x1`**)
- Also **`s0–s1`** (**`x8–x9`**) and **`s2–s11`** (**`x18–x27`**): saved registers (more about those later)

Instruction Support for Functions (1/4)

C

```
... sum(a,b);... /* a,b:s0,s1 */
    }
int sum(int x, int y) {
return x+y;
}
```

RISC-V

```
address (shown in decimal)
1000
1004
1008
1012
1016
…
2000
2004
```

In RISC-V, all instructions are 4 bytes, and stored in memory just like data. So, here we show the addresses of where the programs are stored.

Instruction Support for Functions (2/4)

C

```
... sum(a,b);... /* a,b:s0,s1 */
}
        int sum(int x, int y) {
        return x+y;
}
```

RISC-V

```
address (shown in decimal)
    1000 mv a0,s0              # x = a
    1004 mv a1,s1              # y = b
    1008 addi ra,zero,1016     #ra=1016
    1012 j    sum             #jump to sum
    1016 …                    # next inst.
    …
    2000 sum: add a0,a0,a1
    2004 jr   ra #new instr."jump reg"
```

Instruction Support for Functions (3/4)

C

```
... sum(a,b);... /* a,b:s0,s1 */
    }
        int sum(int x, int y) {
        return x+y;
    }
```

RISC-V

- Question: Why use `jr` here? Why not use `j`?
- Answer: `sum` might be called by many places, so we can't return to a fixed place. The calling proc to `sum` must be able to say "return here" somehow.

```
...
2000 sum: add a0,a0,a1
2004 jr    ra #new instr."jump reg"
```

Instruction Support for Functions (4/4)

- Single instruction to jump and save return address: jump and link (`jal`)
- Before:

```
1008 addi ra,zero,1016  # ra=1016
1012 j sum              # goto sum
```

- After:

```
1008 jal sum  # ra=1012,goto sum
```

- Why have a `jal`?
  - Make the common case fast: function calls very common
  - Reduce program size
  - Don't have to know where code is in memory with `jal`!

RISC-V Function Call Instructions

- Invoke function*: jump and link* instruction (`jal`)

  (really should be `laj` *"link and jump"*)

  - "link" means form an *address* or *link* that points to calling site to allow function to return to proper address
  - Jumps to address and simultaneously saves the address of the following instruction in register ra

```
jal FunctionLabel
```

- Return from function: *jump register* instruction (`jr`)
  - Unconditional jump to address specified in register: `jr ra`
  - Assembler shorthand: `ret = jr ra`

Summary of Instruction Support

Actually, only two instructions:

- `jal rd, Label` – jump-and-link
- `jalr rd, rs, imm` – jump-and-link register

`j, jr` and `ret` are pseudoinstructions!

- `j: jal x0, Label`

UC Berkeley
Teaching Professor
Dan Garcia

CS61C

Great Ideas in Computer Architecture (a.k.a. Machine Structures)

UC Berkeley
Professor
Bora Nikolić

RISC-V Assembly Language

RISC-V Function Call Example

Review: Six Basic Steps in Calling a Function

1. Put arguments in a place (registers) where function can access them
2. Transfer control to function (`jal`)
3. Acquire (local) storage resources needed for function
4. Perform desired task of the function
5. Put return value in a place where calling code can access it and restore any registers you used; release local storage
6. Return control to point of origin, since a function can be called from several points in a program (`ret`)

Function Call Example

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

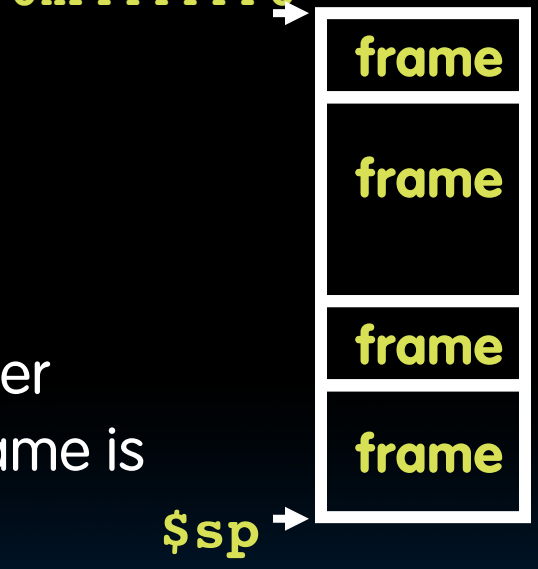
- Parameter variables **g**, **h**, **i**, and **j** in argument registers **a0**, **a1**, **a2**, and **a3**, and **f** in **s0**
- Assume need one temporary register **s1**

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

- Need a place to save old values before calling function, restore them when return, and delete
- Ideal is *stack*: last-in-first-out (LIFO) 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`**)
- Convention is grow stack down from high to low addresses
  - *Push* decrements **`sp`**, *Pop* increments **`sp`**

Stack

- Stack frame includes:
  - Return "instruction" address
  - Parameters (arguments)
  - Space for other local variables
- Stack frames contiguous blocks of memory; stack pointer tells where bottom of stack frame is
- When procedure ends, stack frame is tossed off the stack; frees memory for future stack frames



Reminder: Leaf

```
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**, and **f** in **s0**
- Assume need one temporary register **s1**

RISC-V Code for Leaf()

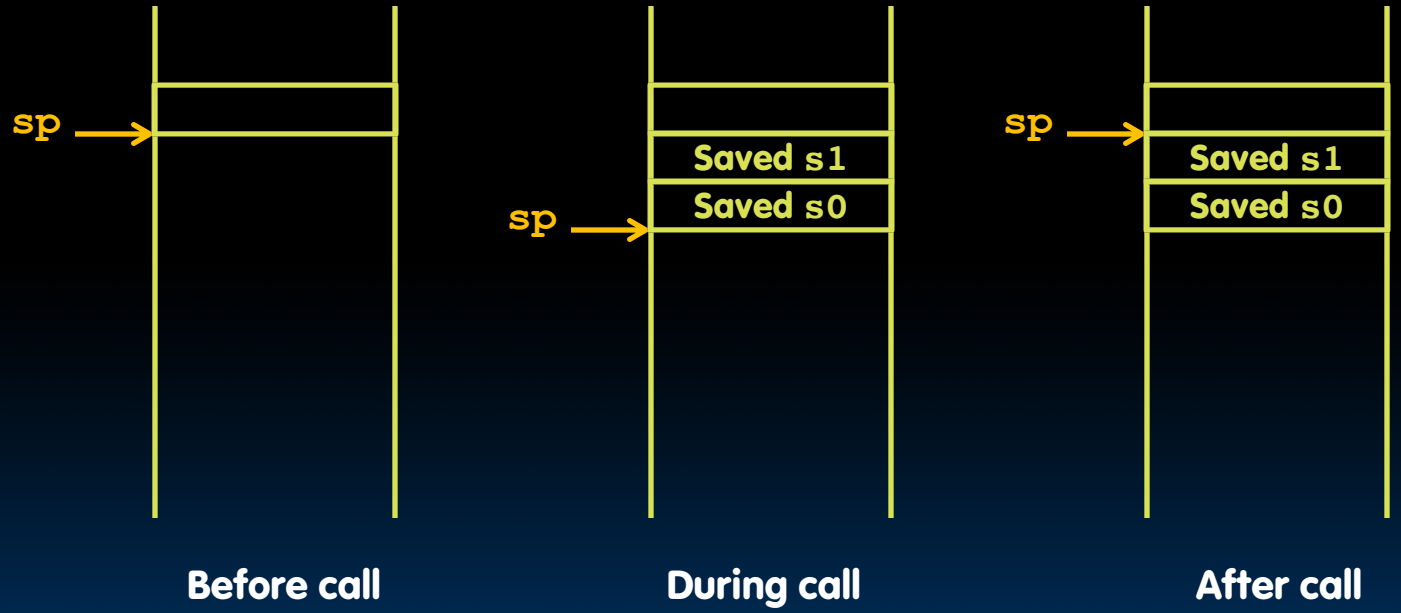
```
Leaf:   addi sp,sp,-8 # adjust stack for 2 items
        sw s1, 4(sp)  # save s1 for use afterwards
        sw s0, 0(sp)  # save s0 for use afterwards

        add s0,a0,a1 # f = 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
```

Stack Before, During, After Function

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



Nested Calls and Register Conventions

What If a Function Calls a Function? Recursive Function Calls?

- Would clobber values in `a0-a7` and `ra`
- What is the solution?

Nested Procedures

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

- Something called **sumSquare**, now **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** – again, use stack

Register Conventions (1/2)

- CalleR: the calling function
- CalleE: the function being called
- When callee returns from executing, the caller needs to know which registers may have changed and which are guaranteed to be unchanged.
- Register Conventions: A set of generally accepted rules as to which registers will be unchanged after a procedure call (`jal`) and which may be changed.

Register Conventions (2/2)

To reduce expensive loads and stores from spilling and restoring registers, RISC-V function-calling convention divides registers into two categories:

1. Preserved across function call
   - 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 *cannot* rely on values being unchanged
   - Argument/return registers `a0-a7`,`ra`,
     "temporary registers" `t0-t6`

RISC-V Symbolic Register Names

Numbers hardware understands

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	-
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	-
x4	tp	Thread pointer	-
x5	t0	Temporary/Alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/Frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/Return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller

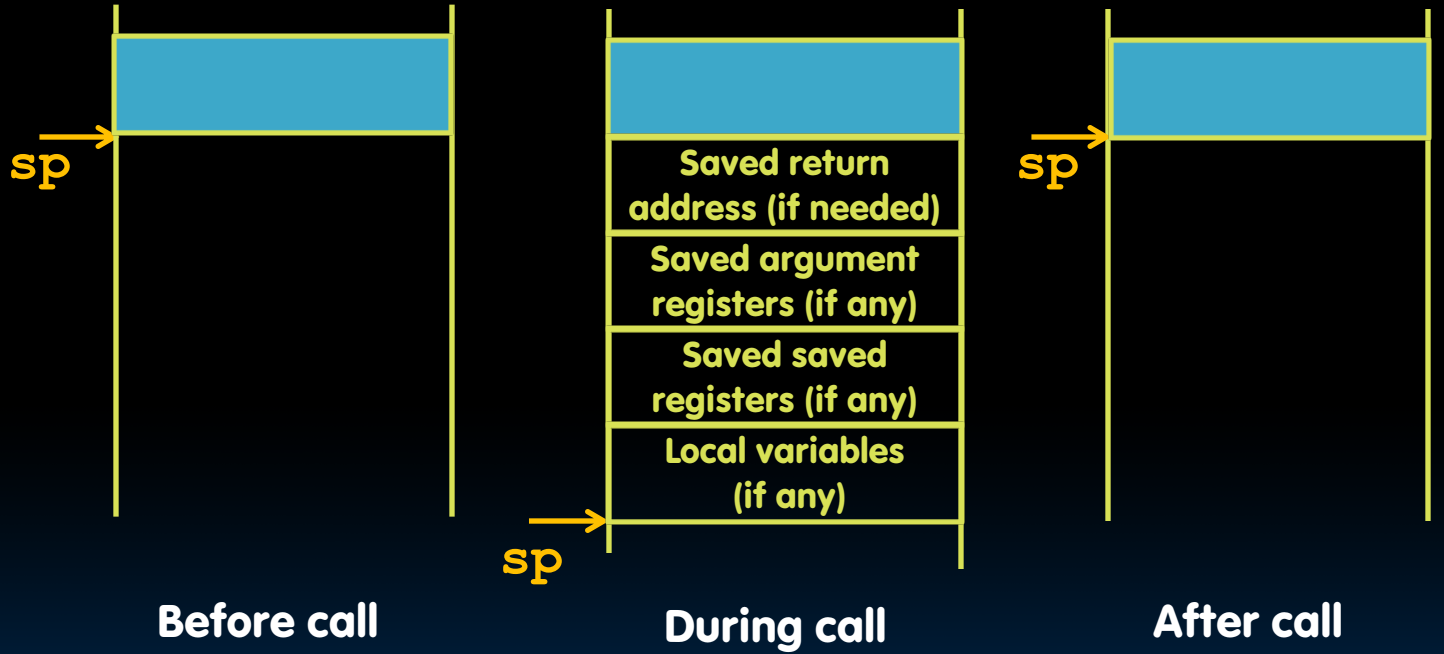
Human-friendly symbolic names in assembly code

Memory Allocation

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 don't fit 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)

- Recall - `sp` 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; }
sumSquare:
"push"  addi sp,sp,-8       # 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
"pop"   addi sp,sp,8        # restore stack
        jr ra
mult:   ...
```

Memory Allocation

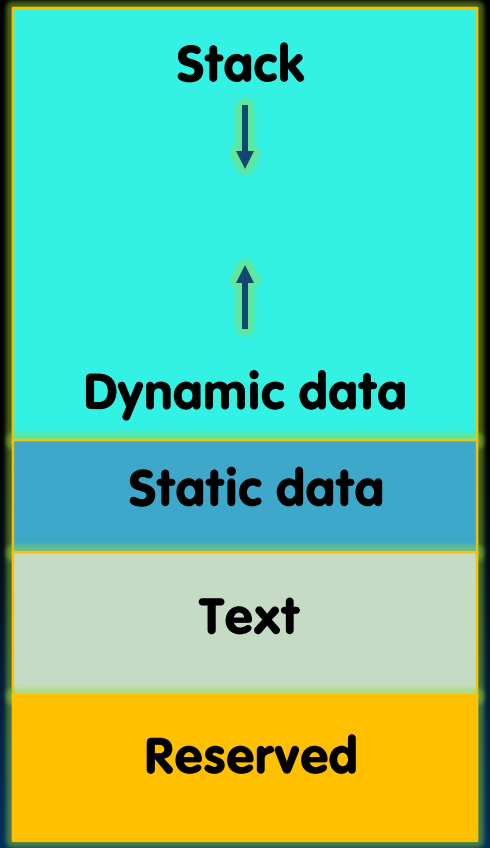
- When a C program is run, there are three important memory areas allocated:
  - Static: Variables declared once per program, cease to exist only after execution completes - e.g., C globals
  - Heap: Variables declared dynamically via `malloc`
  - Stack: Space to be used by procedure during execution; this is where we can save register values

Where is the Stack in Memory?

- RV32 convention (RV64/RV128 have different memory layouts)
- Stack starts in high memory and grows down
  - Hexadecimal: $\texttt{bfff\_fff0}_{\text{hex}}$
  - Stack must be aligned on 16-byte boundary
    (not true in previous examples)
- RV32 programs (*text segment*) in low end
  - $\texttt{0001\_0000}_{\text{hex}}$
- *static data segment* (constants and other static variables) above text for static variables
  - RISC-V convention *global pointer* (`gp`) points to static
  - RV32 `gp` = $\texttt{1000\_0000}_{\text{hex}}$
- *Heap* above static for data structures that grow and shrink ; grows up to high addresses

RV32 Memory Allocation

$\texttt{Sp} = \texttt{bfff fff0}_{\texttt{hex}}$



$\texttt{1000 0000}_{\texttt{hex}}$

$\texttt{pc} = \texttt{0001 0000}_{\texttt{hex}}$

0

“And In Conclusion…”

RV32 So Far...

- Arithmetic/logic

```
add rd, rs1, rs2
sub rd, rs1, rs2
and rd, rs1, rs2
or  rd, rs1, rs2
xor rd, rs1, rs2
sll rd, rs1, rs2
srl rd, rs1, rs2
sra rd, rs1, rs2
```

- Immediate

```
addi rd, rs1, imm
subi rd, rs1, imm
andi rd, rs1, imm
ori  rd, rs1, imm
xori rd, rs1, imm
slli rd, rs1, imm
srli rd, rs1, imm
srai rd, rs1, imm
```

- Load/store

```
lw  rd,  rs1, imm
lb  rd,  rs1, imm
lbu rd,  rs1, imm
sw  rs1, rs2, imm
sb  rs1, rs2, imm
```

- Branching/jumps

```
beq  rs1, rs2, Label

bne  rs1, rs2, Label
bge  rs1, rs2, Label
blt   rs1, rs2, Label
bgeu rs1, rs2, Label
bltu rs1, rs2, Label
jal  rd, Label
jalr rd, rs, imm
```

Great Idea #1: Abstraction (Levels of Representation/Interpretation)

High Level Language Program (e.g., C)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

Compiler

Assembly Language Program (e.g., RISC-V)

```
lw    x3, 0(x10)
lw    x4, 4(x10)
sw    x4, 0(x10)
sw    x3, 4(x10)
```

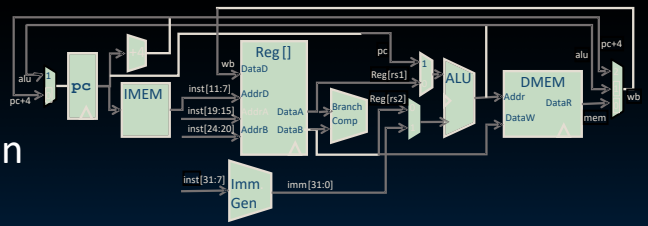
Anything can be represented as a number, i.e., data or instructions

Assembler

Machine Language Program (RISC-V)

```
1000 1101 1110 0010 0000 0000 0000 0000
1000 1110 0001 0000 0000 0000 0000 0100
1010 1110 0001 0010 0000 0000 0000 0000
1010 1101 1110 0010 0000 0000 0000 0100
```

Hardware Architecture Description (e.g., block diagrams)



Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)