CS 211: Intro to Computer Architecture

10.2: RISC-V Function Calls and Memory

Minesh Patel

Spring 2025 – Thursday 3 April

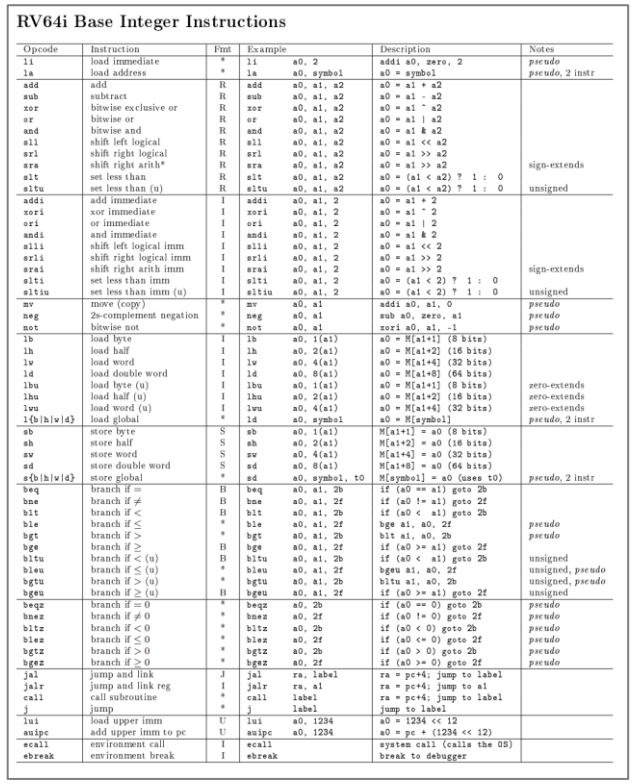
Announcements

- Assignments
  - **Extra Credit:** replaces WA6, due **Next Friday (April 11) @ 11:59 pm**
- Upcoming
  - **WA7:** released last night, due **Next Wednesday (April 9) @ 11:59 pm**
  - **PA4:** TBA today

Useful Resources: Specs and Reference Sheets

https://github.com/riscv/riscv-isa-manual/

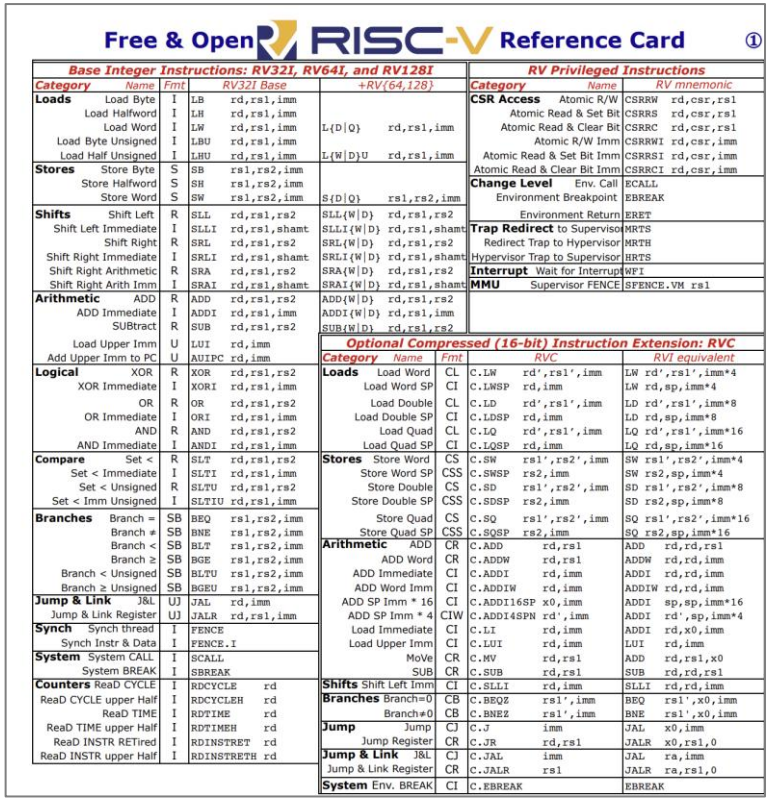
https://github.com/riscv-non-isa/riscv-elf-psabi-doc



RV64i Base Integer Instructions

Opcode	Instruction	Fmt	Example		Description	Notes
li	load immediate	*	li	a0, 2	addi a0, zero, 2	*pseudo*
la	load address	*	la	a0, symbol	a0 = symbol	*pseudo*, 2 instr
add	add	R	add	a0, a1, a2	a0 = a1 + a2	
sub	subtract	R	sub	a0, a1, a2	a0 = a1 - a2	
xor	bitwise exclusive or	R	xor	a0, a1, a2	a0 = a1 ^ a2	
or	bitwise or	R	or	a0, a1, a2	a0 = a1 | a2	
and	bitwise and	R	and	a0, a1, a2	a0 = a1 & a2	
sll	shift left logical	R	sll	a0, a1, a2	a0 = a1 << a2	
srl	shift right logical	R	srl	a0, a1, a2	a0 = a1 >> a2	
sra	shift right arith*	R	sra	a0, a1, a2	a0 = a1 >> a2	sign-extends
slt	set less than	R	slt	a0, a1, a2	a0 = (a1 < a2) ? 1 : 0	
sltu	set less than (u)	R	sltu	a0, a1, a2	a0 = (a1 < a2) ? 1 : 0	unsigned
addi	add immediate	I	addi	a0, a1, 2	a0 = a1 + 2	
xori	xor immediate	I	xori	a0, a1, 2	a0 = a1 ^ 2	
ori	or immediate	I	ori	a0, a1, 2	a0 = a1 | 2	
andi	and immediate	I	andi	a0, a1, 2	a0 = a1 & 2	
slli	shift left logical imm	I	slli	a0, a1, 2	a0 = a1 << 2	
srli	shift right logical imm	I	srli	a0, a1, 2	a0 = a1 >> 2	
srai	shift right arith imm	I	srai	a0, a1, 2	a0 = a1 >> 2	sign-extends
slti	set less than imm	I	slti	a0, a1, 2	a0 = (a1 < 2) ? 1 : 0	
sltiu	set less than imm (u)	I	sltiu	a0, a1, 2	a0 = (a1 < 2) ? 1 : 0	unsigned
mv	move (copy)	*	mv	a0, a1	addi a0, a1, 0	*pseudo*
neg	2s-complement negation	*	neg	a0, a1	sub a0, zero, a1	*pseudo*
not	bitwise not	*	not	a0, a1	xori a0, a1, -1	*pseudo*
lb	load byte	I	lb	a0, 1(a1)	a0 = M[a1+1] (8 bits)	
lh	load half	I	lh	a0, 2(a1)	a0 = M[a1+2] (16 bits)	
lw	load word	I	lw	a0, 4(a1)	a0 = M[a1+4] (32 bits)	
ld	load double word	I	ld	a0, 8(a1)	a0 = M[a1+8] (64 bits)	
lbu	load byte (u)	I	lbu	a0, 1(a1)	a0 = M[a1+1] (8 bits)	zero-extends
lhu	load half (u)	I	lhu	a0, 2(a1)	a0 = M[a1+2] (16 bits)	zero-extends
lwu	load word (u)	I	lwu	a0, 4(a1)	a0 = M[a1+4] (32 bits)	zero-extends
l{b|h|w|d}	load global	*	ld	a0, symbol	a0 = M[symbol]	*pseudo*, 2 instr
sb	store byte	S	sb	a0, 1(a1)	M[a1+1] = a0 (8 bits)	
sh	store half	S	sh	a0, 2(a1)	M[a1+2] = a0 (16 bits)	
sw	store word	S	sw	a0, 4(a1)	M[a1+4] = a0 (32 bits)	
sd	store double word	S	sd	a0, 8(a1)	M[a1+8] = a0 (64 bits)	
s{b|h|w|d}	store global	*	sd	a0, symbol, t0	M[symbol] = a0 (uses t0)	*pseudo*, 2 instr
beq	branch if =	B	beq	a0, a1, 2b	if (a0 == a1) goto 2b	
bne	branch if ≠	B	bne	a0, a1, 2f	if (a0 != a1) goto 2f	
blt	branch if <	B	blt	a0, a1, 2b	if (a0 < a1) goto 2b	
ble	branch if ≤	*	ble	a0, a1, 2f	bge a1, a0, 2f	*pseudo*
bgt	branch if >	*	bgt	a0, a1, 2b	blt a1, a0, 2b	*pseudo*
bge	branch if ≥	B	bge	a0, a1, 2f	if (a0 >= a1) goto 2f	
bltu	branch if < (u)	B	bltu	a0, a1, 2b	if (a0 < a1) goto 2b	unsigned
bleu	branch if ≤ (u)	*	bleu	a0, a1, 2f	bgeu a1, a0, 2f	unsigned, *pseudo*
bgtu	branch if > (u)	*	bgtu	a0, a1, 2b	bltu a1, a0, 2b	unsigned, *pseudo*
bgeu	branch if ≥ (u)	B	bgeu	a0, a1, 2f	if (a0 >= a1) goto 2f	unsigned
beqz	branch if = 0	*	beqz	a0, 2b	if (a0 == 0) goto 2b	*pseudo*
bnez	branch if ≠ 0	*	bnez	a0, 2f	if (a0 != 0) goto 2f	*pseudo*
bltz	branch if < 0	*	bltz	a0, 2b	if (a0 < 0) goto 2b	*pseudo*
blez	branch if ≤ 0	*	blez	a0, 2f	if (a0 <= 0) goto 2f	*pseudo*
bgtz	branch if > 0	*	bgtz	a0, 2b	if (a0 > 0) goto 2b	*pseudo*
bgez	branch if ≥ 0	*	bgez	a0, 2f	if (a0 >= 0) goto 2f	*pseudo*
jal	jump and link	J	jal	ra, label	ra = pc+4; jump to label	
jalr	jump and link reg	I	jalr	ra, a1	ra = pc+4; jump to a1	
call	call subroutine	*	call	label	ra = pc+4; jump to label	
j	jump	*	j	label	jump to label	
lui	load upper imm	U	lui	a0, 1234	a0 = 1234 << 12	
auipc	add upper imm to pc	U	auipc	a0, 1234	a0 = pc + (1234 << 12)	
ecall	environment call	I	ecall		system call (calls the OS)	
ebreak	environment break	I	ebreak		break to debugger	

https://www.cs.utahtech.edu/cs/2810/riscv-card.pdf



Free & Open RISC-V Reference Card ①

Base Integer Instructions: RV32I, RV64I, and RV128I

Category	Name	Fmt	RV32I Base	+RV{64,128}
Loads	Load Byte	I	LB rd,rs1,imm	
	Load Halfword	I	LH rd,rs1,imm	
	Load Word	I	LW rd,rs1,imm	L{D|Q} rd,rs1,imm
	Load Byte Unsigned	I	LBU rd,rs1,imm	
	Load Half Unsigned	I	LHU rd,rs1,imm	L{W|D}U rd,rs1,imm
Stores	Store Byte	S	SB rs1,rs2,imm	
	Store Halfword	S	SH rs1,rs2,imm	
	Store Word	S	SW rs1,rs2,imm	S{D|Q} rs1,rs2,imm
Shifts	Shift Left	R	SLL rd,rs1,rs2	SLL{W|D} rd,rs1,rs2
	Shift Left Immediate	I	SLLI rd,rs1,shamt	SLLI{W|D} rd,rs1,shamt
	Shift Right	R	SRL rd,rs1,rs2	SRL{W|D} rd,rs1,rs2
	Shift Right Immediate	I	SRLI rd,rs1,shamt	SRLI{W|D} rd,rs1,shamt
	Shift Right Arithmetic	R	SRA rd,rs1,rs2	SRA{W|D} rd,rs1,rs2
	Shift Right Arith Imm	I	SRAI rd,rs1,shamt	SRAI{W|D} rd,rs1,shamt
Arithmetic	ADD	R	ADD rd,rs1,rs2	ADD{W|D} rd,rs1,rs2
	ADD Immediate	I	ADDI rd,rs1,imm	ADDI{W|D} rd,rs1,imm
	SUBtract	R	SUB rd,rs1,rs2	SUB{W|D} rd,rs1,rs2
	Load Upper Imm	U	LUI rd,imm	
	Add Upper Imm to PC	U	AUIPC rd,imm	
Logical	XOR	R	XOR rd,rs1,rs2	
	XOR Immediate	I	XORI rd,rs1,imm	
	OR	R	OR rd,rs1,rs2	
	OR Immediate	I	ORI rd,rs1,imm	
	AND	R	AND rd,rs1,rs2	
	AND Immediate	I	ANDI rd,rs1,imm	
Compare	Set <	R	SLT rd,rs1,rs2	
	Set < Immediate	I	SLTI rd,rs1,imm	
	Set < Unsigned	R	SLTU rd,rs1,rs2	
	Set < Imm Unsigned	I	SLTIU rd,rs1,imm	
Branches	Branch =	SB	BEQ rs1,rs2,imm	
	Branch ≠	SB	BNE rs1,rs2,imm	
	Branch <	SB	BLT rs1,rs2,imm	
	Branch ≥	SB	BGE rs1,rs2,imm	
	Branch < Unsigned	SB	BLTU rs1,rs2,imm	
	Branch ≥ Unsigned	SB	BGEU rs1,rs2,imm	
Jump & Link	J&L	UJ	JAL rd,imm	
	Jump & Link Register	UJ	JALR rd,rs1,imm	
Synch	Synch thread	I	FENCE	
	Synch Instr & Data	I	FENCE.I	
System	System CALL	I	SCALL	
	System BREAK	I	SBREAK	
Counters	ReaD CYCLE	I	RDCYCLE rd	
	ReaD CYCLE upper Half	I	RDCYCLEH rd	
	ReaD TIME	I	RDTIME rd	
	ReaD TIME upper Half	I	RDTIMEH rd	
	ReaD INSTR RETired	I	RDINSTRET rd	
	ReaD INSTR upper Half	I	RDINSTRETH rd	

RV Privileged Instructions

Category	Name	RV mnemonic
CSR Access	Atomic R/W	CSRRW rd,csr,rs1
	Atomic Read & Set Bit	CSRRS rd,csr,rs1
	Atomic Read & Clear Bit	CSRRC rd,csr,rs1
	Atomic R/W Imm	CSRRWI rd,csr,imm
	Atomic Read & Set Bit Imm	CSRRSI rd,csr,imm
	Atomic Read & Clear Bit Imm	CSRRCI rd,csr,imm
Change Level	Env. Call	ECALL
	Environment Breakpoint	EBREAK
	Environment Return	ERET
Trap Redirect	to Supervisor	MRTS
	Redirect Trap to Hypervisor	MRTH
	Hypervisor Trap to Supervisor	HRTS
Interrupt	Wait for Interrupt	WFI
MMU	Supervisor FENCE	SFENCE.VM rs1

Optional Compressed (16-bit) Instruction Extension: RVC

Category	Name	Fmt	RVC	RVI equivalent
Loads	Load Word	CL	C.LW rd',rs1',imm	LW rd',rs1',imm*4
	Load Word SP	CI	C.LWSP rd,imm	LW rd,sp,imm*4
	Load Double	CL	C.LD rd',rs1',imm	LD rd',rs1',imm*8
	Load Double SP	CI	C.LDSP rd,imm	LD rd,sp,imm*8
	Load Quad	CL	C.LQ rd',rs1',imm	LQ rd',rs1',imm*16
	Load Quad SP	CI	C.LQSP rd,imm	LQ rd,sp,imm*16
Stores	Store Word	CS	C.SW rs1',rs2',imm	SW rs1',rs2',imm*4
	Store Word SP	CSS	C.SWSP rs2,imm	SW rs2,sp,imm*4
	Store Double	CS	C.SD rs1',rs2',imm	SD rs1',rs2',imm*8
	Store Double SP	CSS	C.SDSP rs2,imm	SD rs2,sp,imm*8
	Store Quad	CS	C.SQ rs1',rs2',imm	SQ rs1',rs2',imm*16
	Store Quad SP	CSS	C.SQSP rs2,imm	SQ rs2,sp,imm*16
Arithmetic	ADD	CR	C.ADD rd,rs1	ADD rd,rd,rs1
	ADD Word	CR	C.ADDW rd,rs1	ADDW rd,rd,rs1
	ADD Immediate	CI	C.ADDI rd,imm	ADDI rd,rd,imm
	ADD Word Imm	CI	C.ADDIW rd,imm	ADDIW rd,rd,imm
	ADD SP Imm * 16	CI	C.ADDI16SP x0,imm	ADDI sp,sp,imm*16
	ADD SP Imm * 4	CIW	C.ADDI4SPN rd',imm	ADDI rd',sp,imm*4
	Load Immediate	CI	C.LI rd,imm	ADDI rd,x0,imm
	Load Upper Imm	CI	C.LUI rd,imm	LUI rd,imm
	MoVe	CR	C.MV rd,rs1	ADD rd,rs1,x0
	SUB	CR	C.SUB rd,rs1	SUB rd,rd,rs1
Shifts	Shift Left Imm	CI	C.SLLI rd,imm	SLLI rd,rd,imm
Branches	Branch=0	CB	C.BEQZ rs1',imm	BEQ rs1',x0,imm
	Branch≠0	CB	C.BNEZ rs1',imm	BNE rs1',x0,imm
Jump	Jump	CJ	C.J imm	JAL x0,imm
	Jump Register	CR	C.JR rd,rs1	JALR x0,rs1,0
Jump & Link	J&L	CJ	C.JAL imm	JAL ra,imm
	Jump & Link Register	CR	C.JALR rs1	JALR ra,rs1,0
System	Env. BREAK	CI	C.EBREAK	EBREAK

https://www.cl.cam.ac.uk/teaching/1617/ECAD+Arch/files/docs/RISCVGreenCardv8-20151013.pdf

Agenda

- **Function Calls**
  - Caller-save and Callee-save
  - Hello, World
- Loads and Stores
- Call Stack

Unconditional Branches

- Unconditional branches are called **jumps**

func.c

```c
void func(void)
{
    while(1) {}
}
```

func.S

```
func:
.loop:
    j .loop // pseudo for: jal x0, .loop
            // (we'll come back to this)
```

- Most interesting use case: **function calls**

func.c

```c
void func(void)
{
    func();
}
```

func.S

```
func:
    call func // pseudo for something complex
              // (we'll come back to this)
```

Function Calls

- Four required components:

Now

1. **`goto`** the right place
2. **`return`** safely
3. Pass arguments to the function
4. Get return values

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

goto the right place

- We already saw this: **calculate a PC-offset**

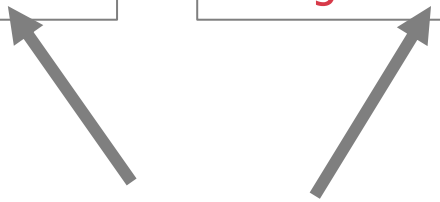
```
b<cond> rs0, rs1, <PC offset>
j <PC offset>
```

main_b.S

```
main:
    beqz x0, main
```

main_j.S

```
main:
    j main // pseudo-instruction
```



Assembler will calculate
the offset based on a **label**

return safely

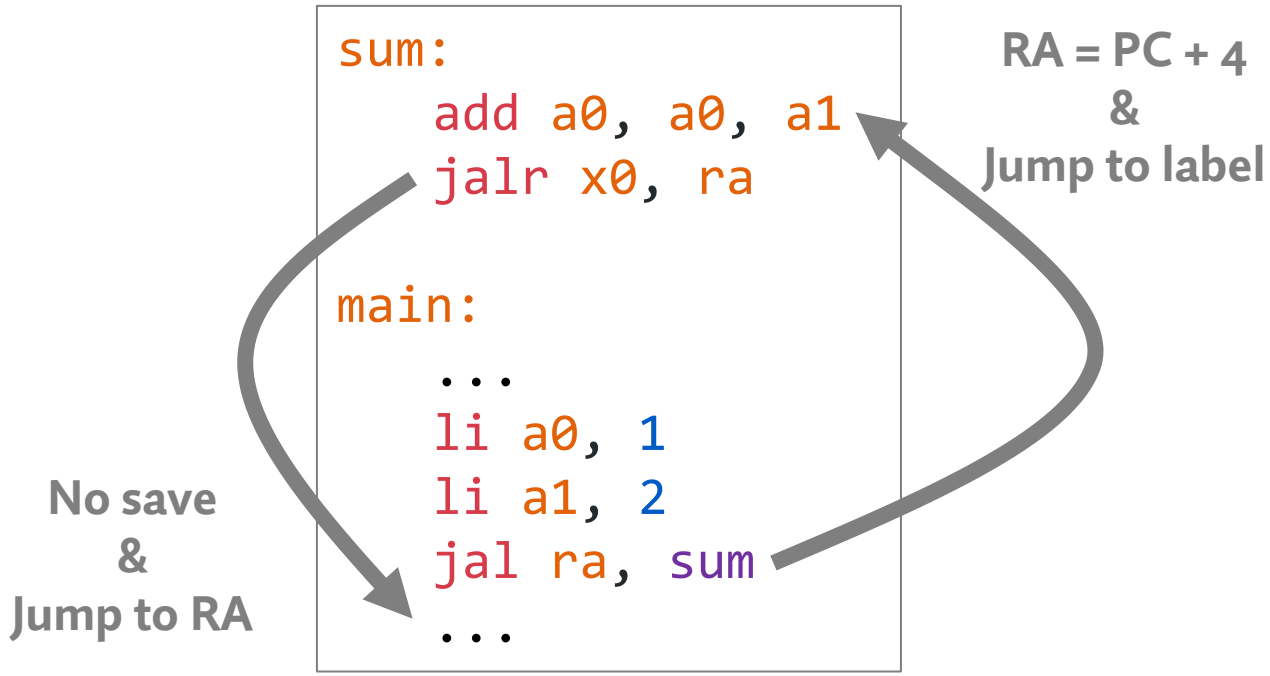
- We need to save the **current PC** before jumping so we can return back

jal	jump and link	J	jal	ra, label	ra = pc+4;	jump to label
jalr	jump and link reg	I	jalr	ra, a1	ra = pc+4;	jump to a1

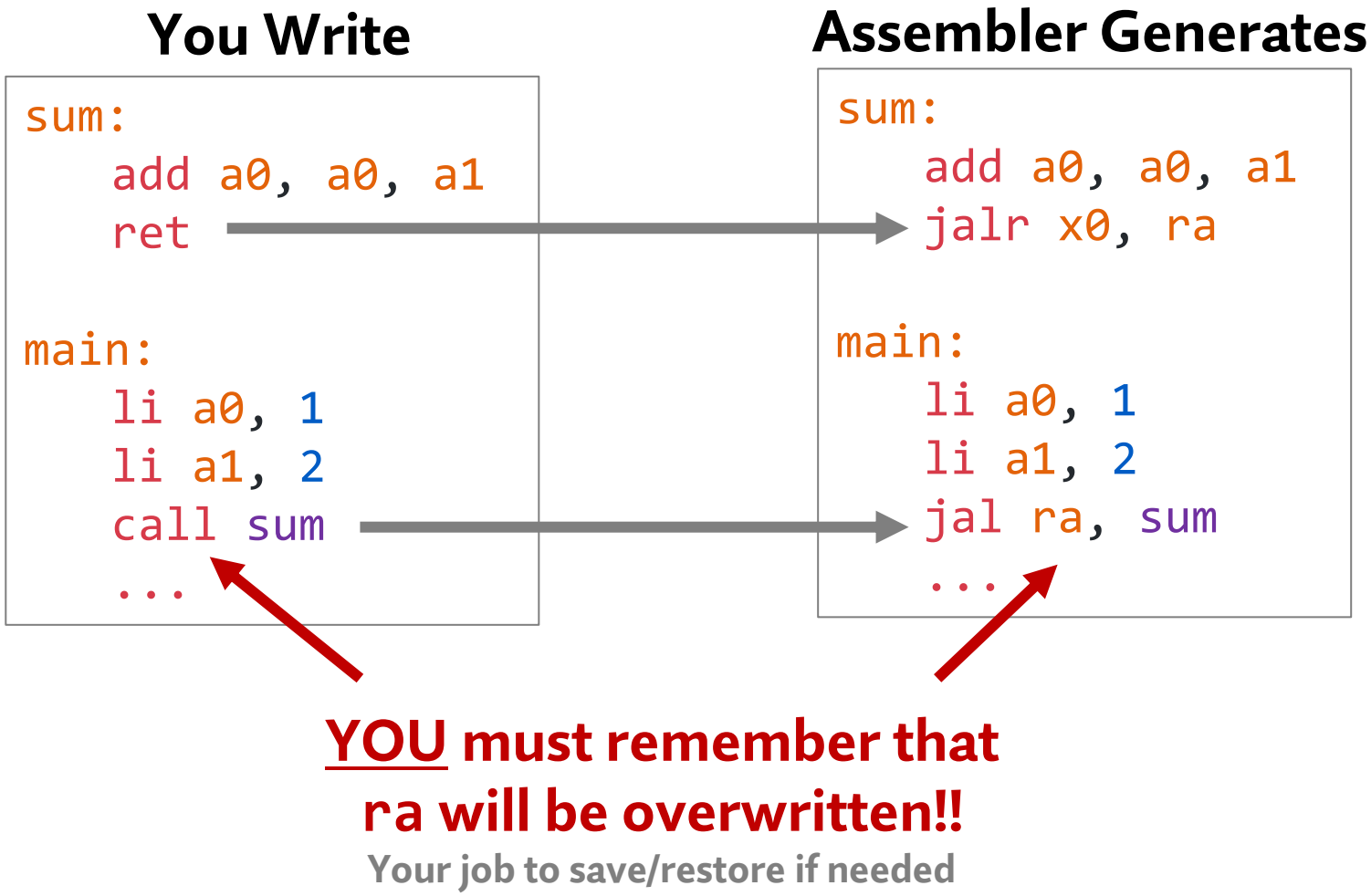
Save the current PC **Goto the jump target**

General-Purpose Registers

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



`call` and `ret` Pseudo-Instructions



Under the Hood: The `call` Pseduo-Instruction

- **Short-range** (nearby PC offset): Jump via an **immediate offset** (i.e., a label)
- **Long-range** (distant PC offset) : Jump via a **pointer in a register**

jal	jump and link	J	jal ra, label	ra = pc+4; jump to label
jalr	jump and link reg	I	jalr ra, a1	ra = pc+4; jump to a1

PC Offset

"near" jump (PC ± 1 MiB)

```
sum:
    add a0, a0, a1
    jalr x0, ra

main:
    li a0, 1
    li a1, 2
    jal ra, sum
    ...
```

"far" jump using auipc (PC ± 2 GiB)

```
sum:
    add a0, a0, a1
    jalr x0, ra

main:
    li a0, 1
    li a1, 2
    auipc a2, sum
    jalr ra, a2
    ...
```

Some gory detail:
https://www.sifive.com/blog/all-aboard-part-3-linker-relaxation-in-riscv-toolchain

`call` and `ret` Pseudo-Instructions

You Write

```
sum:
    add a0, a0, a1
    ret

main:
    li a0, 1
    li a1, 2
    call sum
    ...
```

YOU must remember that ra will be overwritten!!

Your job to save/restore if needed

Near Jump

```
sum:
    add a0, a0, a1
    jalr x0, ra

main:
    li a0, 1
    li a1, 2
    jal ra, sum
    ...
```

Linker will choose

Far Jump

```
sum:
    add a0, a0, a1
    jalr x0, ra

main:
    li a0, 1
    li a1, 2
    auipc a2, sum
    jalr ra, a2
    ...
```

Agenda

- Function Calls
  - **Caller-save and Callee-save**
  - Hello, World
- Loads and Stores
- Call Stack

Saving and Restoring Registers

- **Caller-Save Register:** callee might modify ("clobber") it
- **Callee-Save Register:** callee will NOT modify ("clobber") it

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	—
x5–7	t0–2	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

You Write

```
sum:
    add a0, a0, a1
    ret

main:
    li a0, 1
    li a1, 2
    mv s0, ra
    call sum
    mv ra, s0
    ...
```

Save/ Restore

Note: we should really be saving s0 on the call stack ☹

Putting it All Together: Calling a Function

1. Save any **caller-save registers** you use (almost always includes ra)
2. Use **call/ret**
3. Restore your **saved registers**

General-Purpose Registers

Register	ABI Name	Saver
x0	zero	—
x1	ra	Caller
x2	sp	Callee
x3	gp	—
x4	tp	—
x5–7	t0–2	Caller
x8	s0/fp	Callee
x9	s1	Callee
x10–11	a0–1	Caller
x12–17	a2–7	Caller
x18–27	s2–11	Callee
x28–31	t3–6	Caller

C Code

```
void func(void)
{
    ...
}

int main()
{
    func();
    ...
}
```

ASM Code

```
sum:
    ...
    ret

main:
    mv s0, ra // save ra
    call sum
    mv ra, s0 // restore ra
    ...
```

Practice Exercise: Calculate the Fibonacci Numbers

```c
uint64_t fib(uint64_t n)
{
   return (n == 0 || n == 1) ? n : fib(n - 1) + fib(n - 2);
}
```

fib.c

```c
uint64_t fib(uint64_t n);

int main()
{
   printf("%lu\n", fib(6)); // expect 8
   return EXIT_SUCCESS;
}
```

fib.S

```
.text

.globl fib
fib:
   ...
   ret
```

```
mp2099@ilab1:~/cs211/experiment$ /common/users/shared/cs211_s25_5678/toolchain_glibc3/bin/riscv64-unknown-linux-gnu-gcc -Wl,-rpath=/common/users/shared/cs211_s25_5678/toolchain_glibc3/sysroot/lib -Wl,--dynamic-linker=/common/users/shared/cs211_s25_5678/toolchain_glibc3/sysroot/lib/ld-linux-riscv64-lp64.so.1
-g -march=rv64i -mabi=lp64 fib.c fib.S -o fib
```

Tail Optimization

- Can skip saving the **`ra`** if **`call`** is **the final instruction** in a function
  - The callee's ret will use the caller's **`ra`**

C Code

```c
void func(void)
{
    puts("hello, world");
}

int main()
{
    func();
}
```

ASM Code

```
.hello:
    .asciz "hello, world"

sum:
    la .hello
    tail puts # pseudo-op

main:
    mv s0, ra
    call sum
    mv ra, s0
    ...
```

Somewhere in the C Libraries

```
puts:
    ...
    ret
```

Jump WITHOUT link
(jal x0, puts)

Agenda

- Function Calls
  - Caller-save and Callee-save
  - **Hello, World**
- Loads and Stores
- Call Stack

RV64i So Far

Instructions

Arithmetic

```
add(i) rd, rs1, rs2/imm
sub(i) rd, rs1, rs2/imm
and(i) rd, rs1, rs2/imm
or(i) rd, rs1, rs2/imm
xor(i) rd, rs1, rs2/imm
sll(i) rd, rs1, rs2/imm
srl(i) rd, rs1, rs2/imm
sra(i) rd, rs1, rs2/imm
```

Comparison

```
slt(u)(i) rd, rs1, rs1/imm
```

Loads/Stores

```
l{b/h/w/d} rd, offset(rs)
lu{b/h/w/d} rd, offset(rs)
s{b/h/w/d} rd, offset(rs)
```

Branches

```
beq rs1, rs2, <label>
bne rs1, rs2, <label>
blt(u) rs1, rs2, <label>
bge(u) rs1, rs2, <label>
```

Jumps

```
jal rd, <label>
jalr rd, offset(rs1)
```

Pseudo-Instructions

Constants/Initialization

```
.asciz <C-style string>
mv rd, rs1
li rd, imm
la rd, <label>
```

Function Calls

```
call label
tail label
j label
ret
```

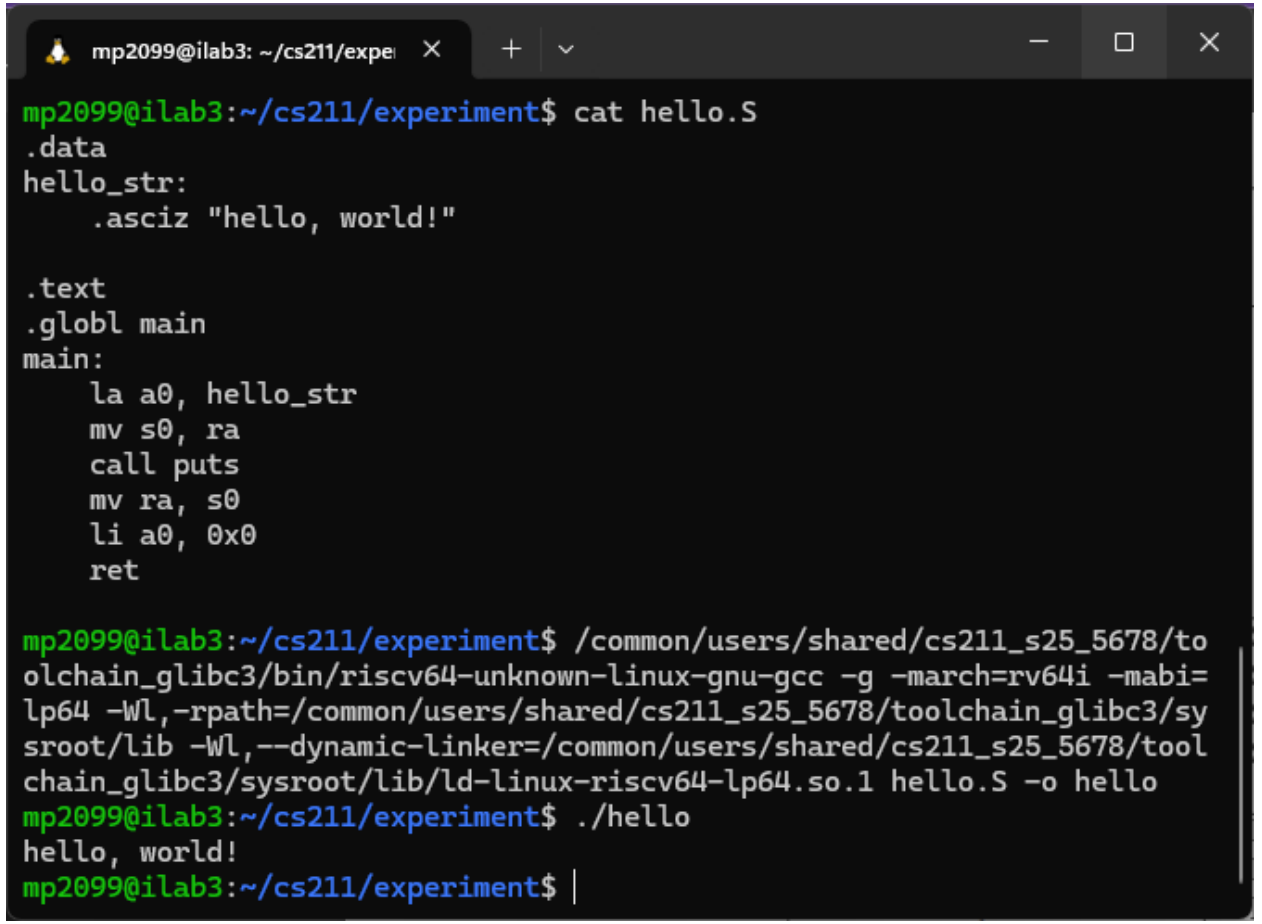
Branches

```
ble(u) rd, <label>
bgt(u) rd, <label>
b{cond}z rd, <label>
```

Finally: Hello World in RISC-V Assembly

```
.data
hello_str:
    .asciz "hello, world!"

.text
.globl main
main:
    la a0, hello_str
    mv s0, ra // save ra
    call puts
    mv ra, s0 // restore ra
    li a0, 0x0 // return 0
    ret
```




```
mp2099@ilab1:~/cs211/experiment$ /common/users/shared/cs211_s25_5678/toolchain_glibc3/bin/riscv64-unknown-linux-gnu-gcc -Wl,-rpath=/common/users/shared/cs211_s25_5678/toolchain_glibc3/sysroot/lib -Wl,--dynamic-linker=/common/users/shared/cs211_s25_5678/toolchain_glibc3/sysroot/lib/ld-linux-riscv64-lp64.so.1
-g -march=rv64i -mabi=lp64 hello.S -o hello
```

Note: we should really be saving s0 on the call stack ☹

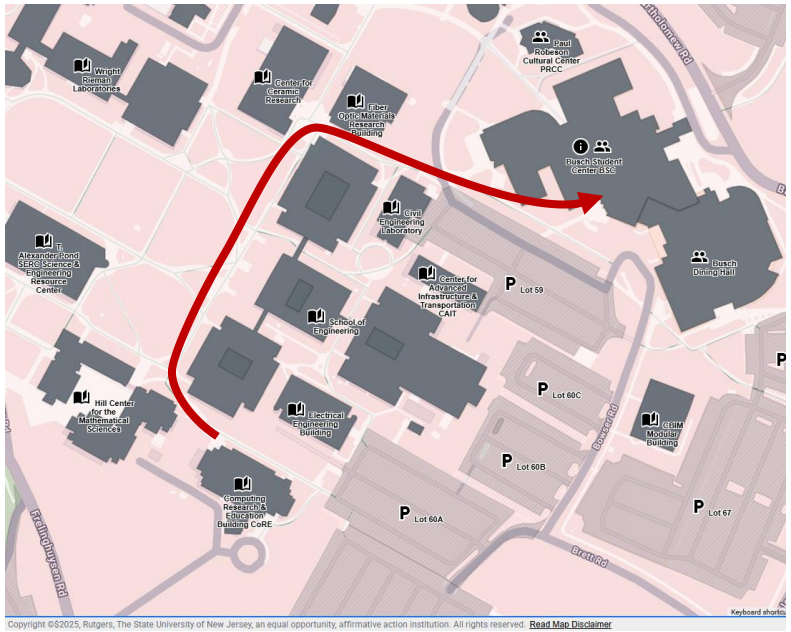
Agenda

- Function Calls
  - Caller-save and Callee-save
  - Hello, World
- **Loads and Stores**
- Call Stack

When 32 Registers Aren't Enough

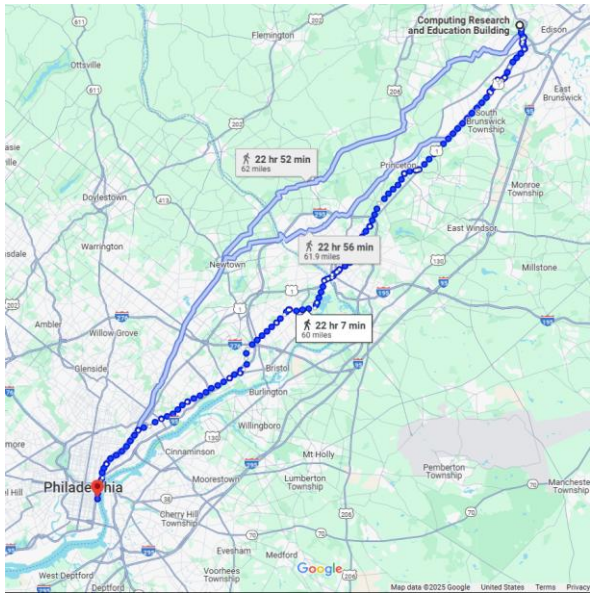
- We spill to memory ☹
- **Recall:** registers are ~2 orders of magnitude **faster** than main memory

Register Access



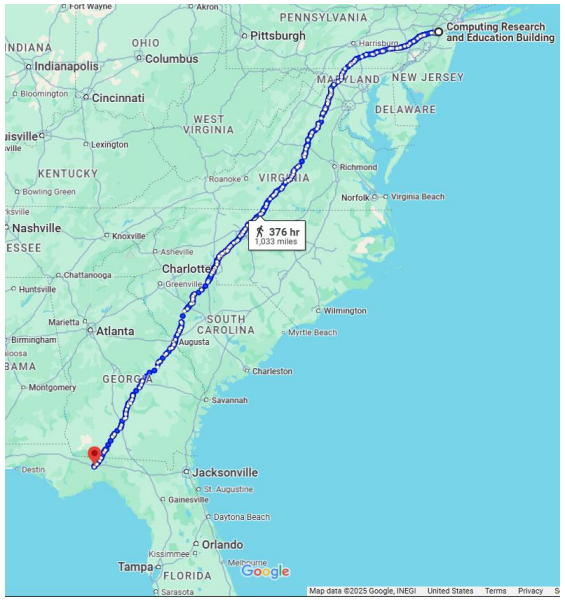
CoRE to BSC (0.3 miles)

Main Memory Access



CoRE to Philly (60 miles)

Disk Access



CoRE to Tallahassee (1000 miles)

Load/Store 8B from Memory (ld/sd)

Load Double-word (64 bits)

```
ld rd, offset(rs)
```

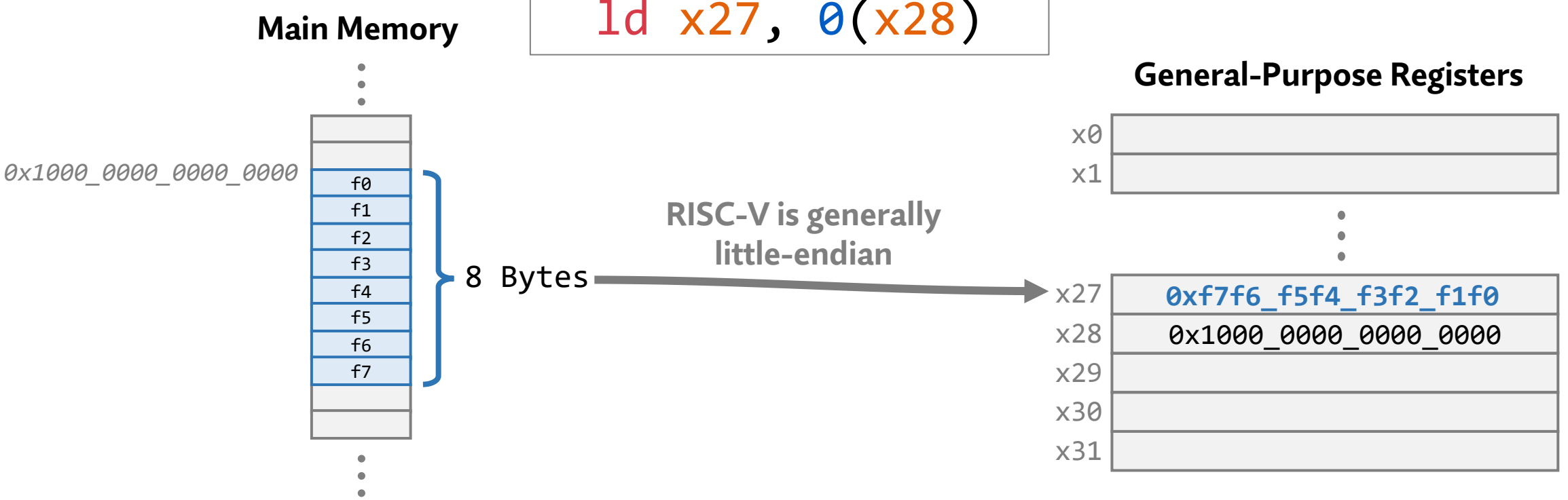
`rd = 8 bytes @ mem[rs + offset]`

Store Double-word (64 bits)

```
sd rd, offset(rs)
```

`mem[rs + offset] = 8 bytes in rd`

```
ld x27, 0(x28)
```



Example: Accessing an Array

Load Double-word (64 bits)

```
ld rd, offset(rs)
```

`rd = 8 bytes @ mem[rs + offset]`

Store Double-word (64 bits)

```
sd rd, offset(rs)
```

`mem[rs + offset] = 8 bytes in rd`

C Code

```
int64_t a0[5] = {1, 2, 3, 4, 5};
a0[0]++;
a0[1]--;
...
```

ASM Code

```
ld a1, 0(a0) // a0[0]
addi a1, a1, 1
sd a1, 0(a0)

ld a1, 8(a0) // a0[1]
addi a1, a1, -1
sd a1, 8(a0)

...
```

Load/Store 1B from Memory (lb/sb)

Load Byte (8 bits)

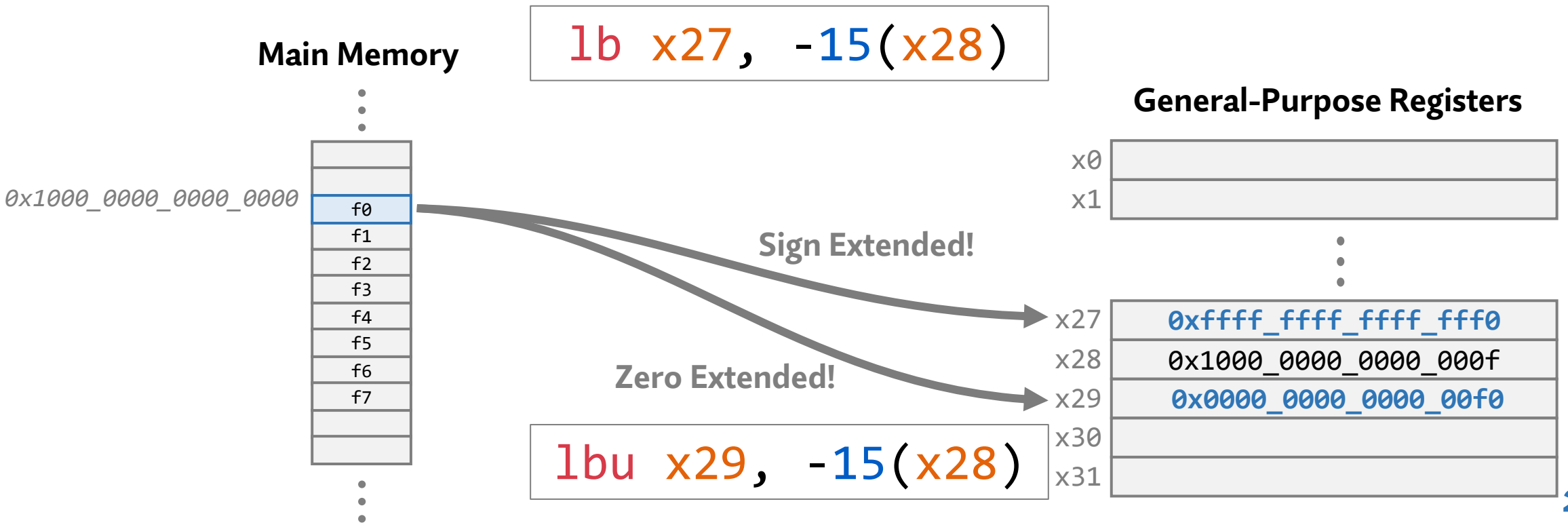
```
ld rb, offset(rs)
```

`rd = 1 byte @ mem[rs + offset]`

Store Byte (8 bits)

```
sb rd, offset(rs)
```

`mem[rs + offset] = 1 byte in rd`



Load/Store 2B from Memory (lh/sh)

Load Half Word (16 bits)

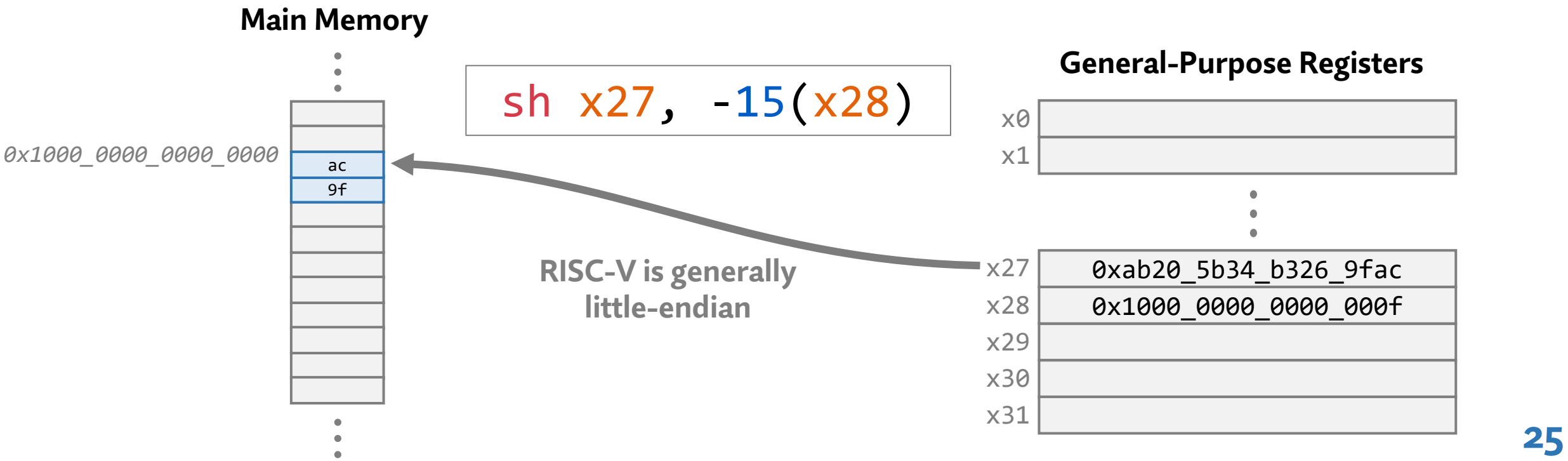
```
lh rb, offset(rs)
```

`rd = 2 bytes @ mem[rs + offset]`

Store Half Word (16 bits)

```
sh rd, offset(rs)
```

`mem[rs + offset] = 2 bytes in rd`



Loads and Stores

Signed Loads

```
lb rd, offset(rs) // int8_t
lh rd, offset(rs) // int16_t
lw rd, offset(rs) // int32_t
ld rd, offset(rs) // int64_t
```

Unsigned Loads

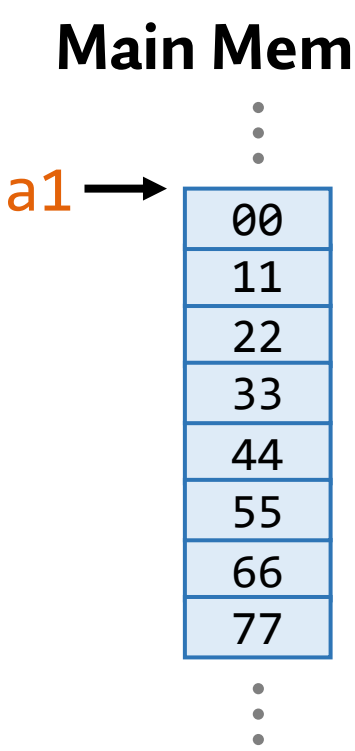
```
lbu rd, offset(rs) // int8_t
lhu rd, offset(rs) // int16_t
lwu rd, offset(rs) // int32_t
ldu rd, offset(rs) // int64_t
```

Stores

```
sb rs1, offset(rs2) // int8_t
sh rs1, offset(rs2) // int16_t
sw rs1, offset(rs2) // int32_t
sd rs1, offset(rs2) // int64_t
```

Aside: Why Little Endian?

- Mostly **long-standing convention**
  - Neat property: **different-width** accesses always get LSBs of **the same number**




```
lb a0, 0(a1) // a0 = 0x0000_0000_0000_0000
```

```
lh a0, 0(a1) // a0 = 0x0000_0000_0000_1100
```

```
lw a0, 0(a1) // a0 = 0x0000_0000_3322_1100
```

```
ld a0, 0(a1) // a0 = 0x7766_5544_3322_1100
```

Agenda

- Function Calls
  - Caller-save and Callee-save
  - Hello, World
- Loads and Stores
- **Call Stack**

Motivating the Call Stack

- Eventually, we **run out of registers**

Enormous (or Many) Arguments

```
struct big_data
{
    uint64_t a[100];
    void *p[100];
    ...
};

struct big_data func(struct big_data b)
{
    ...
}
```

Local Scratch Space

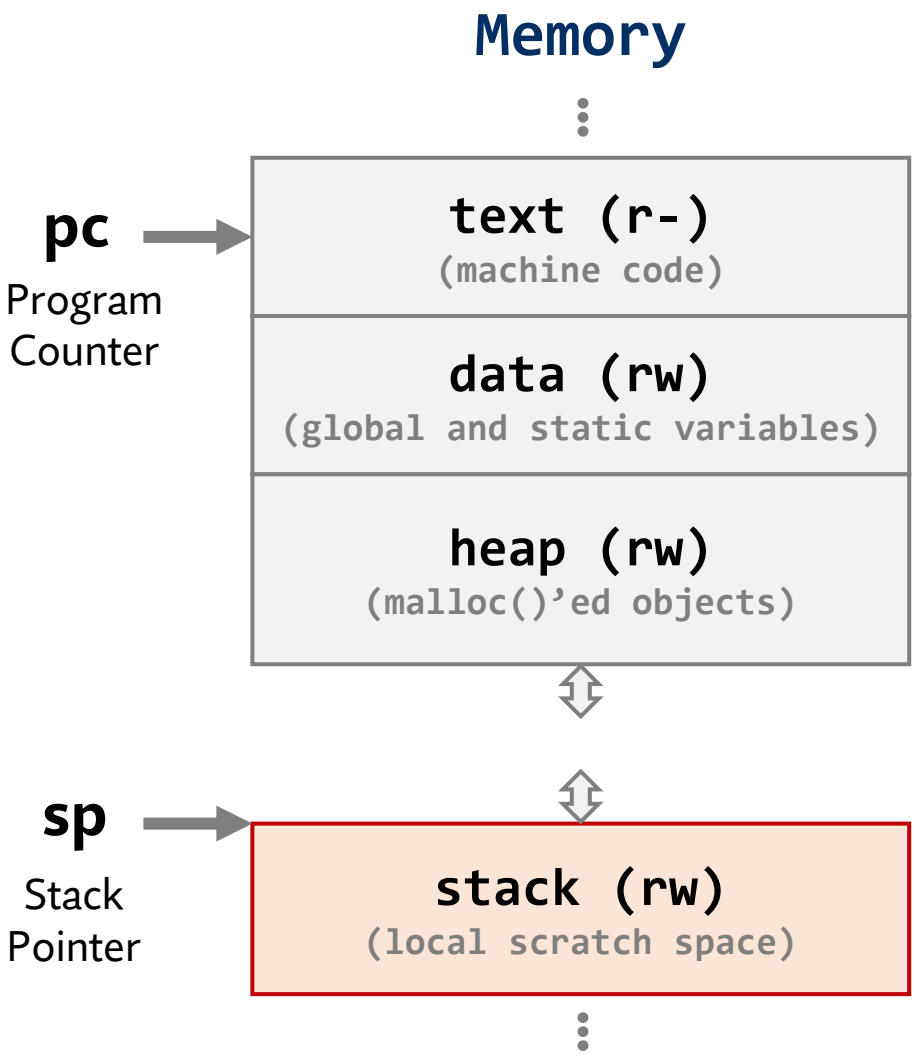
```
void func(void b)
{
    struct big_data b[100];
    ...
}
```

Caller-Save

```
void func(void b)
{
    func2();
}
```

- **Also:** debugging, temporary storage, recursion, and more…

The Call Stack



When we run out of **registers**...

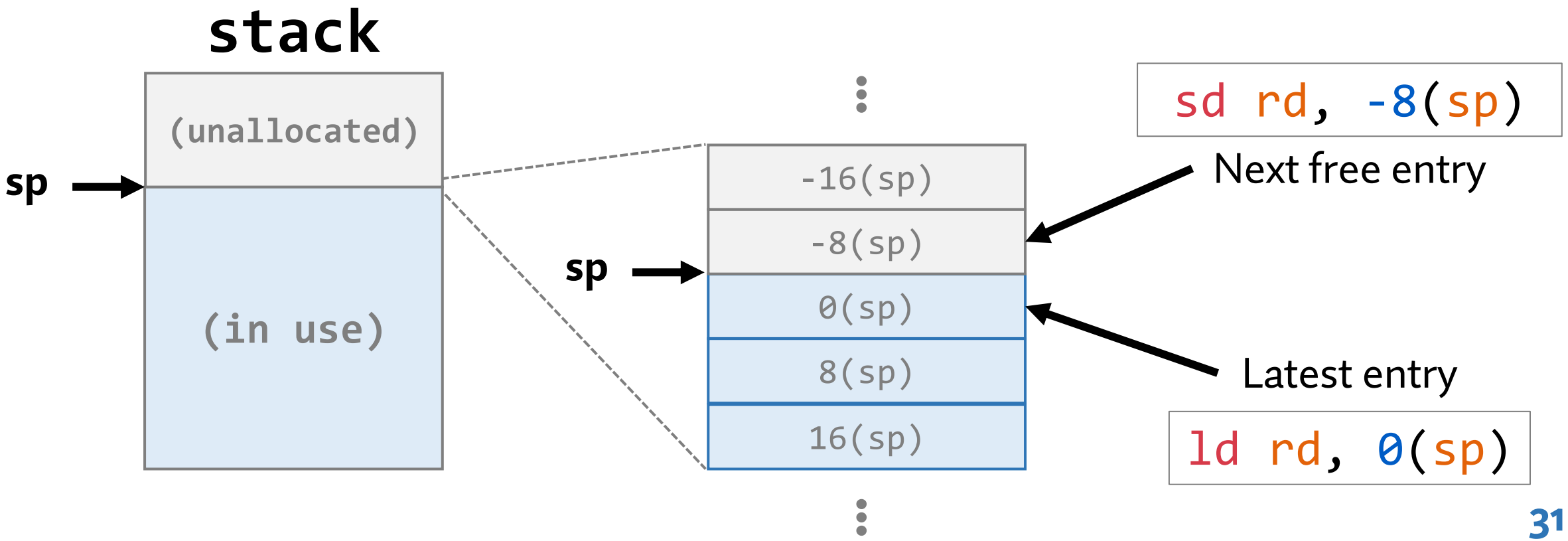
...we resort to **main memory** (`ld`/`sd`)

General-Purpose Registers

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

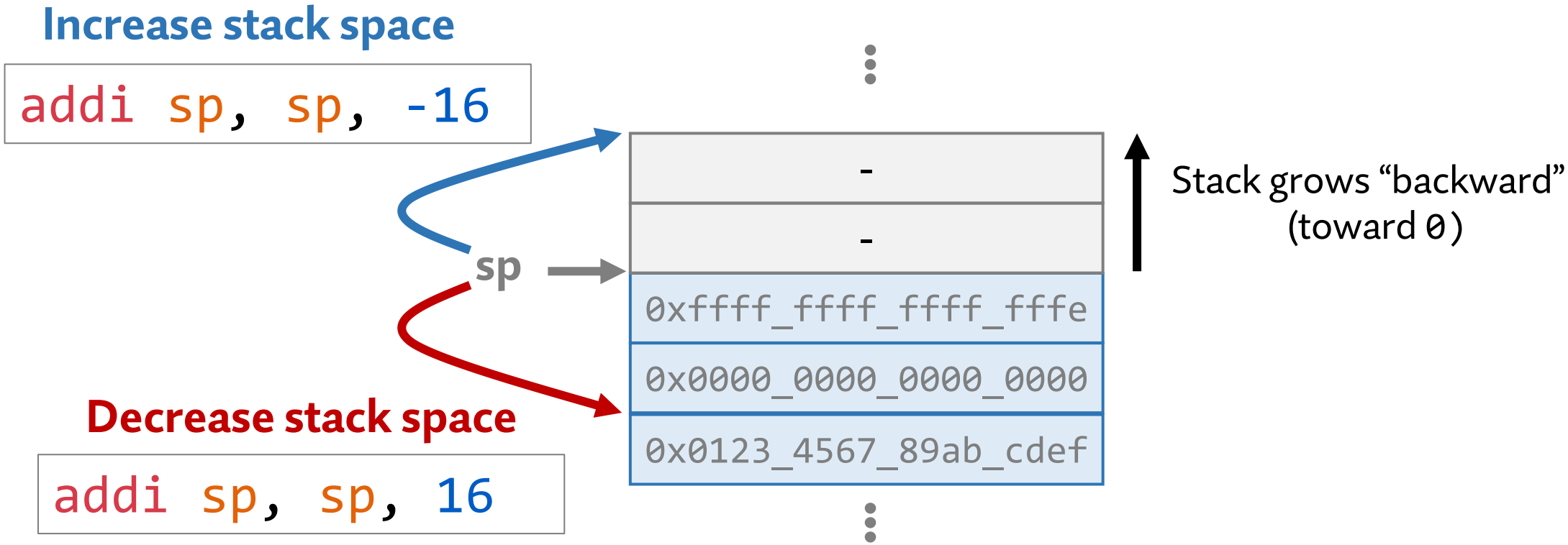
Using the Call Stack

- It's just a **region of memory** with a fancy name
  - Memory is just an array of bytes; sp is **just another pointer**
  - **Only vaguely resembles** an algorithmic data structure



Using the Call Stack

- We **allocate**/**deallocate** stack space by changing the **stack pointer (SP)**
  - **Performance:** Much faster than calling `malloc()`/`free()`



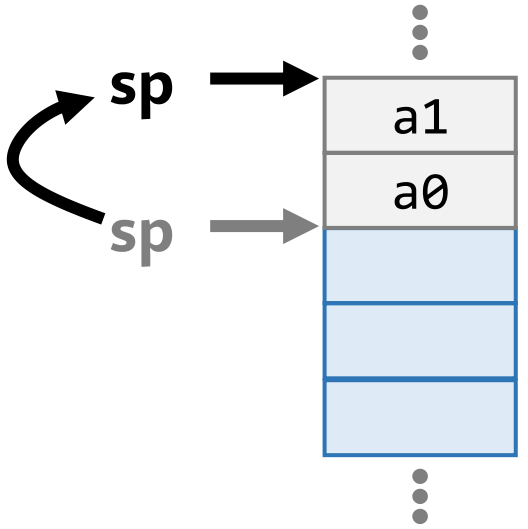
- **Convention:** SP should always be a **multiple of 16 bytes**

Stack "Push" and "Pop"

- **Push (or allocation)**
  - Increase stack space
  - Store data into the new space

Stack "push"

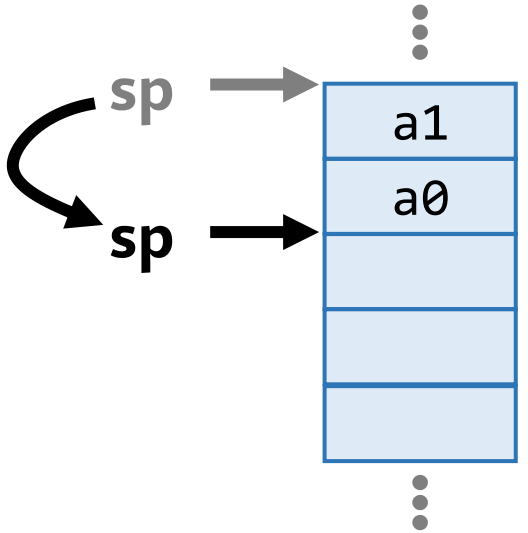
```
addi sp, sp, -16
sd a0, 8(sp)
sd a1, 0(sp)
```



- **Pop (or deallocation)**
  - Load the values we want
  - Decrease stack space
    - Data remains, but we ignore it

Stack "pop"

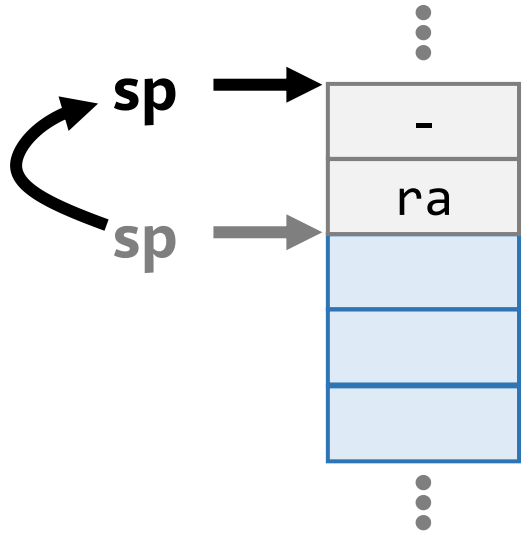
```
ld a1, 0(sp)
ld a0, 8(sp)
addi sp, sp, -16
```



Example: Properly Saving the Return Address

C Code

```
int main()
{
    func();
    ...
}
```



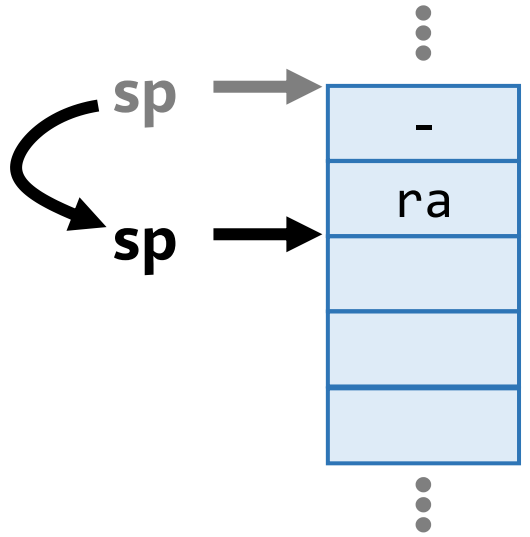
ASM Code

Stack "push"

```
addi sp, sp, -16
sd ra, 8(sp)
call func
```

Stack "pop"

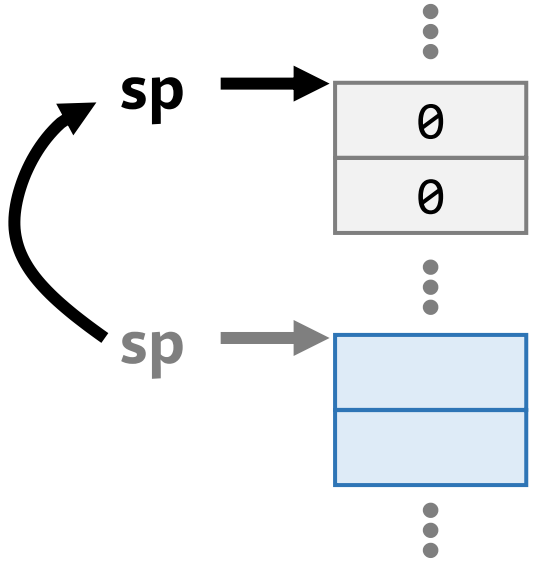
```
ld ra, 8(sp)
addi sp, sp, -16
```



Example: Allocating a Local Array

C Code

```c
int main()
{
    uint64_t b[20] = {0};
    ...
}
```



ASM Code

Allocate

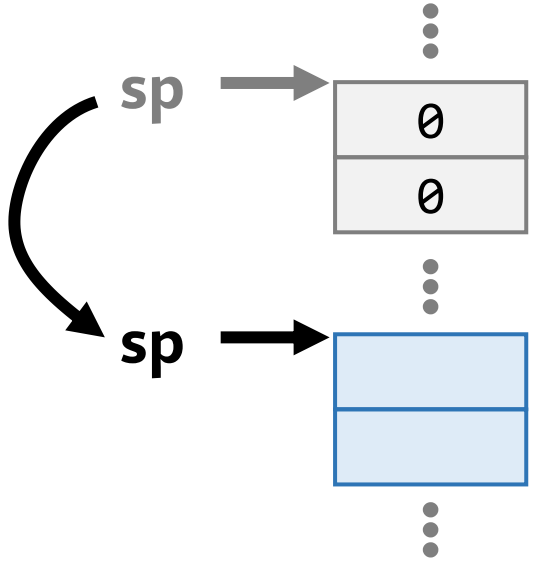
```
addi sp, sp, -160
```

Initialize + Use

```
sd x0, 0(sp)
sd x0, 8(sp)
...
```

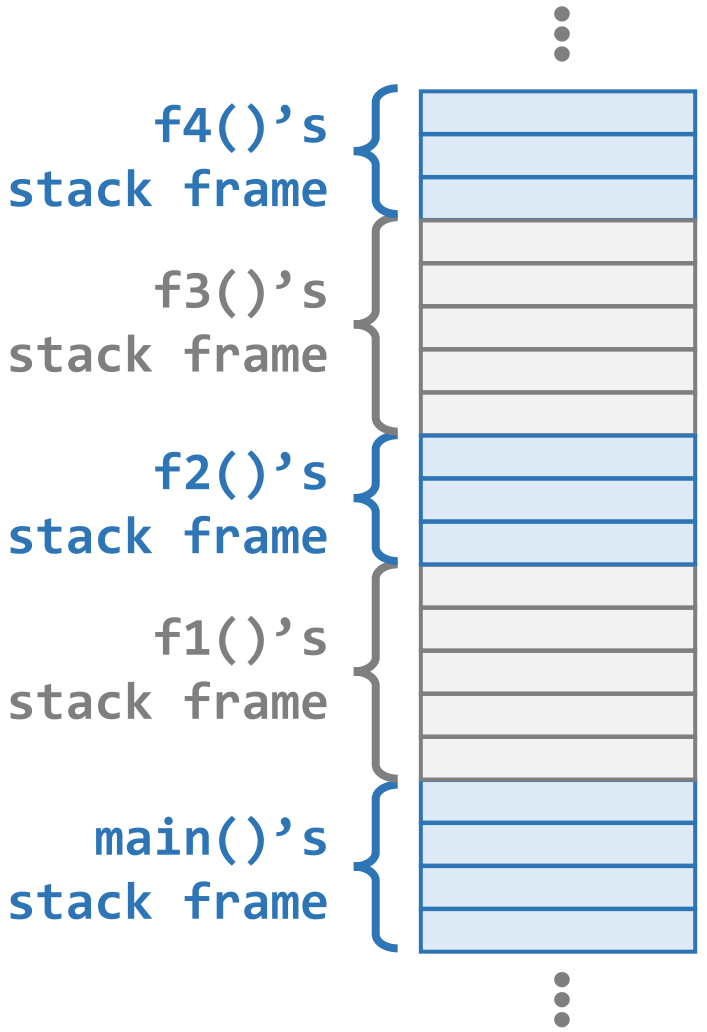
Deallocate

```
addi sp, sp, 160
```



Stack Frames

- **Stack frame:** a function's local data on the stack
  - Arguments
  - Saved registers
  - Local variables
  - Intermediate scratch space
- We **grow the stack** when calling a function
- We **reduce the stack** when returning



C Code

```
void f4(void) { f5(); }
void f3(void) { f4(); }
void f2(void) { f3(); }
void f1(void) { f2(); }
int main() { f1(); return 0; }
```

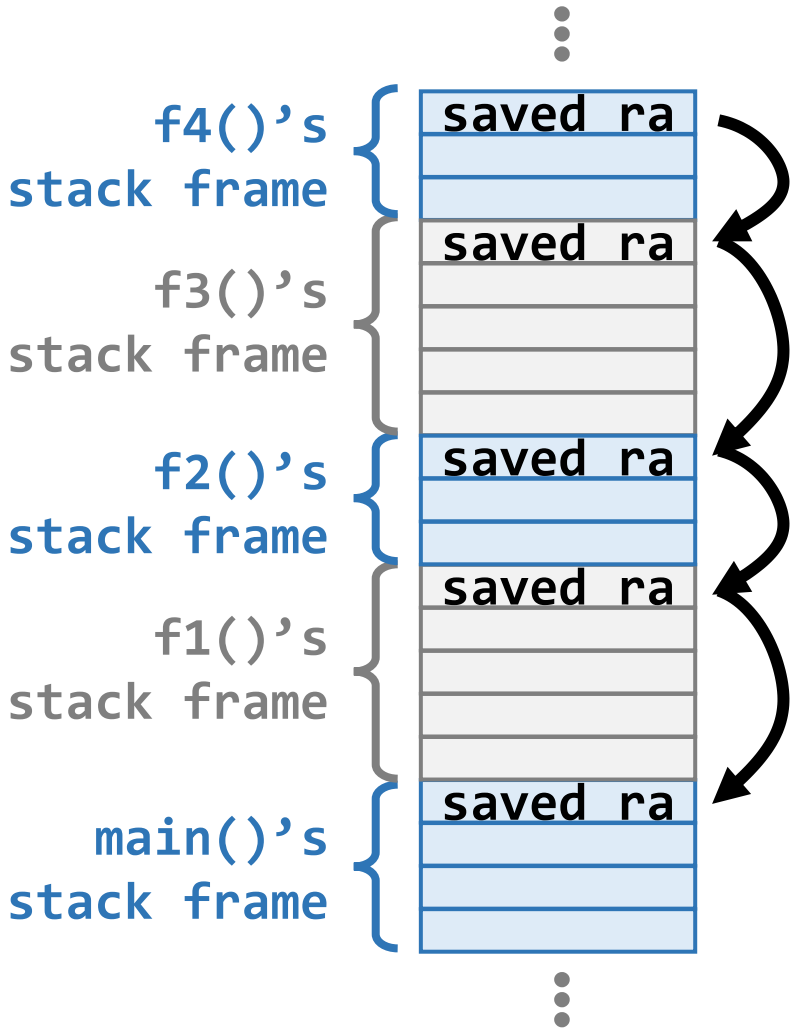
Backtrace: Unwinding the Stack

- Debuggers (e.g., GDB) use stack frames to figure out **which function called which**

- **Saved ra's** form a **linked list**!
  - Only works if you actually save the RA on the stack
  - Guess what "gcc –g" does ☺

C Code

```
void f4(void) { f5(); }
void f3(void) { f4(); }
void f2(void) { f3(); }
void f1(void) { f2(); }
int main() { f1(); return 0; }
```



Example: libasan Stack Unwinding

```
mp2099@ilab1: ~/cs211/pa3
mp2099@ilab1:~/cs211/pa3$ make sanitize
/common/users/shared/cs211_s25_5678/toolchain_glibc3/bin/riscv64-unknown-linux-gnu-gcc -O0 -g -Wall -Wextra -Werror -Wshadow -pedantic -fno-omit-frame-p
ointer -Isrc -std=c11 -march=rv64i -mabi=lp64 -misa-spec=2.2 -fsanitize=address -c -MMD -MF build/pa3.d.asan src/pa3.c -o build/pa3.o.asan
/common/users/shared/cs211_s25_5678/toolchain_glibc3/bin/riscv64-unknown-linux-gnu-gcc -Wl,-rpath=/common/users/shared/cs211_s25_5678/toolchain_glibc3/s
ysroot/lib -fsanitize=address build/main.o.asan build/pa3.o.asan build/fs_generator.o.asan build/pa3_test.o.asan build/grading.o.asan -o pa3.asan
ASAN_OPTIONS=detect_leaks=0 QEMU_RESERVED_VA=256G QEMU_GUEST_BASE=0x14000 qemu-riscv64 -L /common/users/shared/cs211_s25_5678/toolchain_glibc3/sysroot p
a3.asan
[INFO] testing filesystem: "filesystems/fs0_root_only_rw.bin"
================================================================
==931516==ERROR: AddressSanitizer: attempting free on address which was not malloc()-ed: 0x003ffcfe4020 in thread T0
    #0 0x3fff1d65ae in free.part.0 (/common/users/shared/cs211_s25_5678/toolchain_glibc3/sysroot/lib/libasan.so.8+0xc15ae)
    #1 0x1260e in pa3_deserialize_node src/pa3.c:103
    #2 0x11902 in main src/main.c:49
    #3 0x3ffefde714 in __libc_start_call_main (/common/users/shared/cs211_s25_5678/toolchain_glibc3/sysroot/lib/libc.so.6+0x2b714)
    #4 0x3ffefde7bc in __libc_start_main@GLIBC_2.27 (/common/users/shared/cs211_s25_5678/toolchain_glibc3/sysroot/lib/libc.so.6+0x2b7bc)
    #5 0x1164e in _start (/common/home/mp2099/cs211/pa3/pa3.asan+0x1164e)

Address 0x003ffcfe4020 is located in stack of thread T0 at offset 32 in frame
    #0 0x116ce in main src/main.c:10

  This frame has 2 object(s):
    [32, 40) 'deserialized_root' (line 48) <== Memory access at offset 32 is inside this variable
    [64, 80) 'cfgs' (line 29)
HINT: this may be a false positive if your program uses some custom stack unwind mechanism, swapcontext or vfork
      (longjmp and C++ exceptions *are* supported)
SUMMARY: AddressSanitizer: bad-free (/common/users/shared/cs211_s25_5678/toolchain_glibc3/sysroot/lib/libasan.so.8+0xc15ae) in free.part.0
==931516==ABORTING
make: *** [Makefile:46: sanitize] Error 1
mp2099@ilab1:~/cs211/pa3$
```

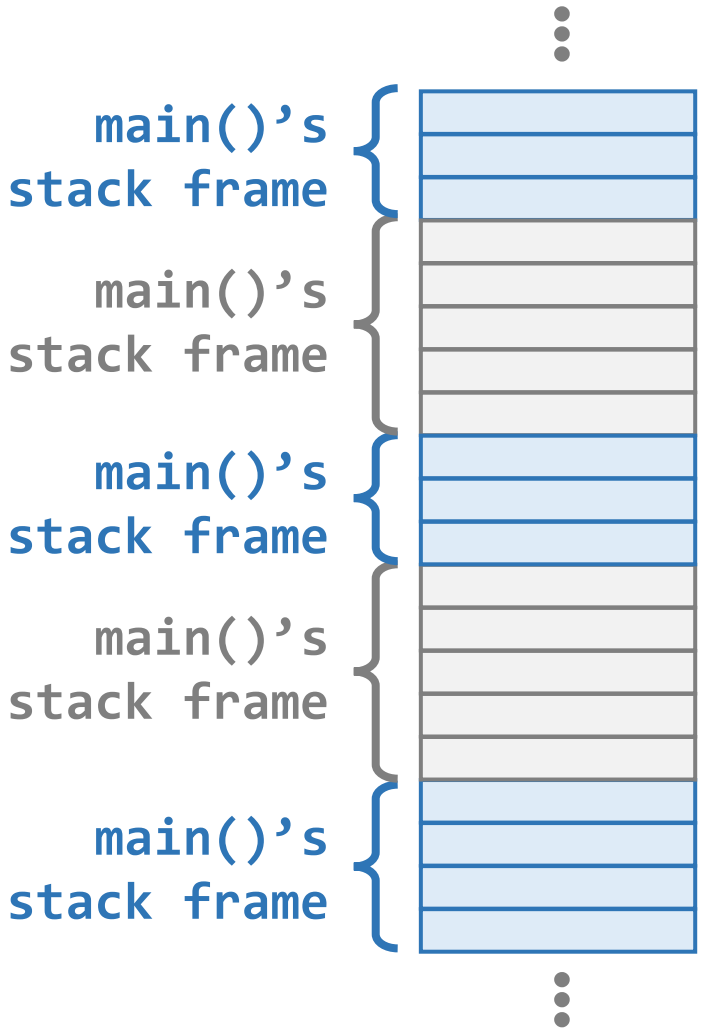
Recursive Function Calls

- Every step of the recursion creates a **new stack frame**

- **"Stack overflow"**: run out of stack!
  - Usually deep or infinite recursion
  - Can also just be deeply-nested functions

C Code

```
int main()
{
   main();
   return 0;
}
```

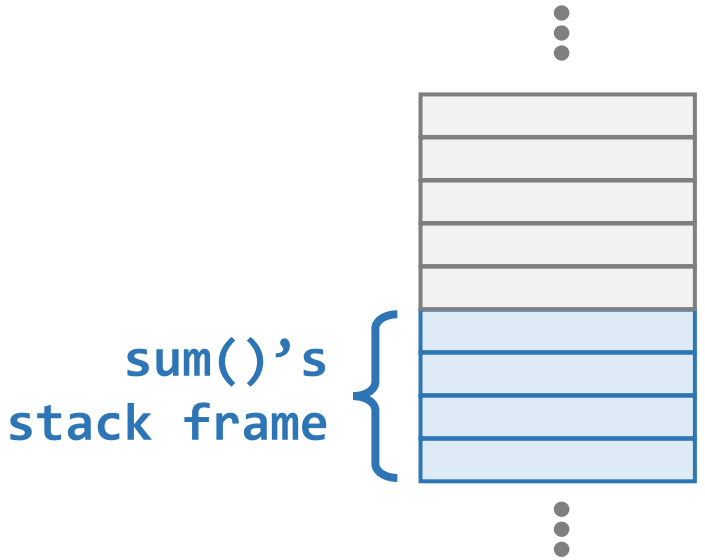


Tail Recursion

- Optimization: **no need to save ra** if recursion is the **final step** of a function
  - We're never coming back to that stack frame!
  - Can **reuse** it for subsequent calls (overwrite arguments, local variables, etc.)

C Code

```
uint64_t sum(uint64_t n, uint64_t result)
{
   if (n <= 1)
      return result;
   return sum(n - 1, n + result);
}
```



C and Assembly Type Interplay

- All C objects are widened to XLEN bits in a register

C Type	Bytes	Register Representation
`(unsigned) char`	1	zext
`signed char`	1	sext
`unsigned short`	2	zext
`(signed) short`	2	sext
`unsigned int`	4	zext
`(signed) int`	4	sext
`unsigned long`	8	-
`(signed) long`	8	-

Mixing C and Assembly

- tl;dr: it works just fine.
  - This is how many standard libraries are written
  - E.g., math library, many C standard library functions, etc.

- Make sure you follow **function calling/stack conventions**
  - Otherwise, you and the compiler will be doing different things
- In a pure assembly program, you can do **anything you want**

RV64i Summary

Instructions

Arithmetic

```
add(i) rd, rs1, rs2/imm
sub(i) rd, rs1, rs2/imm
and(i) rd, rs1, rs2/imm
or(i) rd, rs1, rs2/imm
xor(i) rd, rs1, rs2/imm
sll(i) rd, rs1, rs2/imm
srl(i) rd, rs1, rs2/imm
sra(i) rd, rs1, rs2/imm
```

Comparison

```
slt(u)(i) rd, rs1, rs1/imm
```

Loads/Stores

```
l{b/h/w/d} rd, offset(rs)
lu{b/h/w/d} rd, offset(rs)
s{b/h/w/d} rd, offset(rs)
```

Branches

```
beq rs1, rs2, <label>
bne rs1, rs2, <label>
blt(u) rs1, rs2, <label>
bge(u) rs1, rs2, <label>
```

Jumps

```
jal rd, <label>
jalr rd, offset(rs1)
```

Calculating Offsets

```
lui rd, imm
auipc rd, imm
```

Pseudo-Instructions

Constants/Initialization

```
.asciz <C-style string>
mv rd, rs1
li rd, imm
la rd, <label>
```

Arithmetic

```
neg rd, rs
not rd, rs
```

Function Calls

```
call label
tail label
j label
ret
```

Branches

```
ble(u) rd, <label>
bgt(u) rd, <label>
b{cond}z rd, <label>
```

CS 211: Intro to Computer Architecture

10.2: RISC-V Function Calls and Memory

Minesh Patel

Spring 2025 – Thursday 3 April