CMPE110 Lecture 08
Instruction Set Architecture 5

Heiner Litz

https://canvas.ucsc.edu/courses/19290

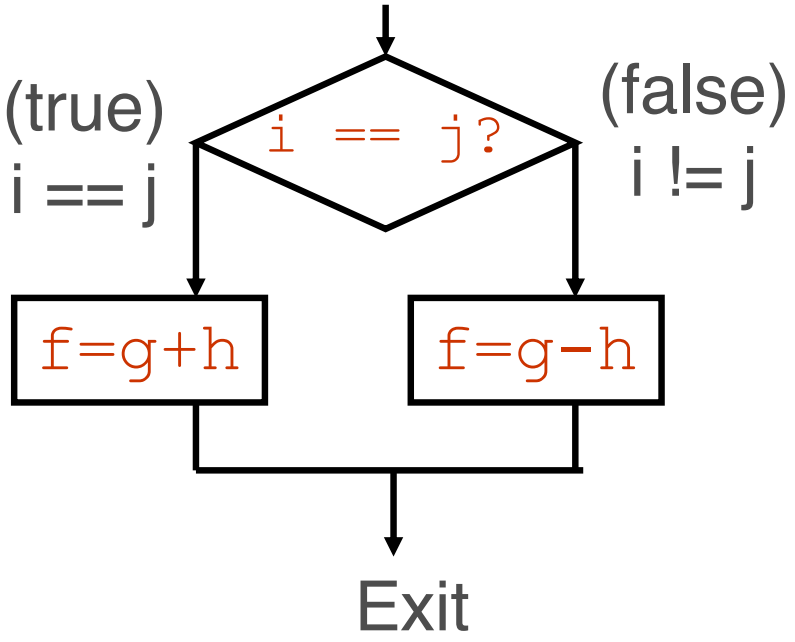
Announcements

Review

If-then-else Example

Consider the code

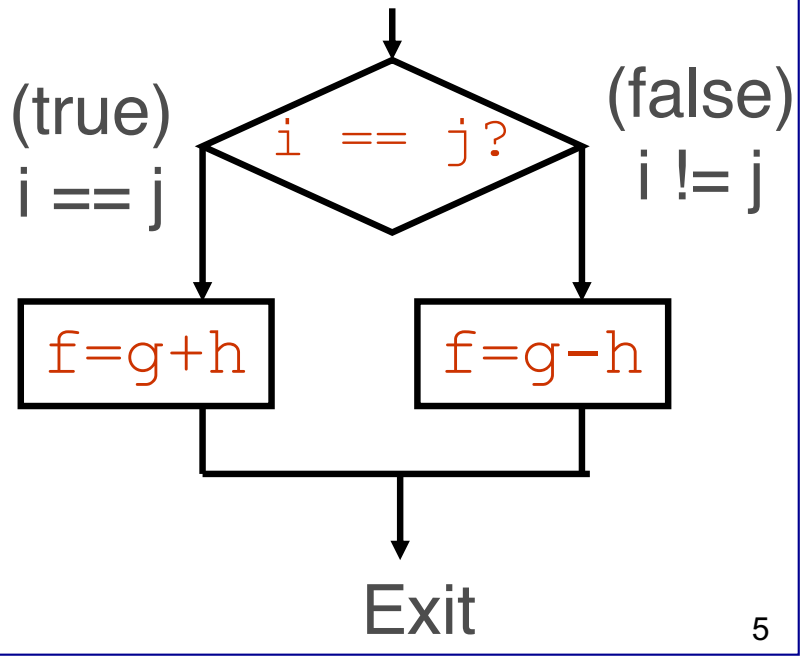
```
if (i == j) f = g + h;
else f = g - h;
```



If-then-else Example

Create labels and use equality instruction

```
        beq x3, x4, True    # Branch if i == j
False:  subu x0, x1, x2     # f = g - h
        j Exit              # Go to Exit
True:   add x0, x1, x2      # f = g + h
Exit:
```



RISC-V Jumps & Branches

Instruction	Example	Meaning
jump	j L	goto L
jump register	jr x1	goto value in x1
jump and link	jal L	goto L and set xra
jump and link register	jalr x1	goto x1 and set xra

xra = return address register

branch equal	beq x1, x2, L	if (x1 == xs2) goto L
branch not eq	bne x1, x2, L	if (x1 != x2) goto L
branch l.t. 0	bltz x1, L	if (x1 < 0) goto L
branch l.t./eq 0	blez x1, L	if (x1 <= 0) goto L
branch g.t. 0	bgtz x1, L	if (x1 > 0) goto L
branch g.t./eq 0	bgez x1, L	if (x1 >= 0) goto L

Branch on Other Comparisons

How do we branch on other comparisons?

<, >, ≤, ≥

Two instruction solution

A comparison instruction with binary outputs (e.g., slt = set less than)

An equality/inequality branch on the comparison output

Consider the following C code

```
if (f < g) goto Less;      # f in x1, g in x2
```

Solution

Remember: x0 is fixed to zero

```
slt xt0, x1, x2            # xt0 = 1 if x1 < x2
bne xt0, x0, Less          # Goto Less if xt0 != 0
```

RISC-V Comparisons

Instruction	Example	Meaning	Comments
set less than	`slt x1, x2, x3`	x1 = (x2 < x3)	comp less than signed
set less than imm	`slti x1, x2, 100`	x1 = (x2 < 100)	comp w/const signed
set less than uns	`sltu x1, x2, x3`	x1 = (x2 < x3)	comp < unsigned
set l.t. imm. uns	`sltiu x1, x2, 100`	x1 = (x2 < 100)	comp < const unsigned

$$A < B = B > A$$
$$!(A < B) = A \geq B$$

C

```
if (a < 8)
```

Assembly

```
slti xt0,xa,8              # xt0 = a < 8
beq  xt0, x0, Exceed       # goto Exceed if xt0 == 0
```

While loop in C

Consider a `while` loop

```
while (A[i] == k)
      i = i + j;
```

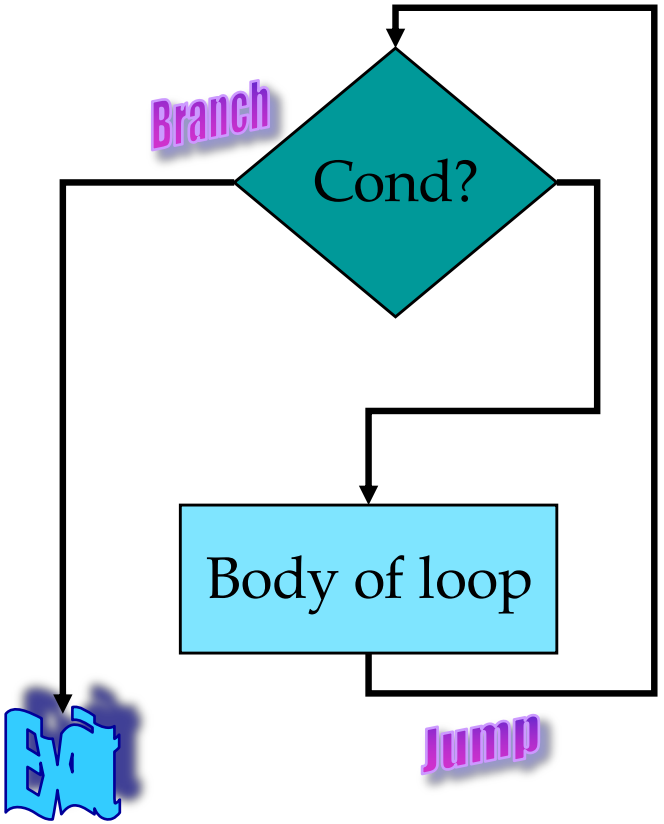
Any idea?

assume `i in x1, j in x4, k in x2, &A[0] in x3`

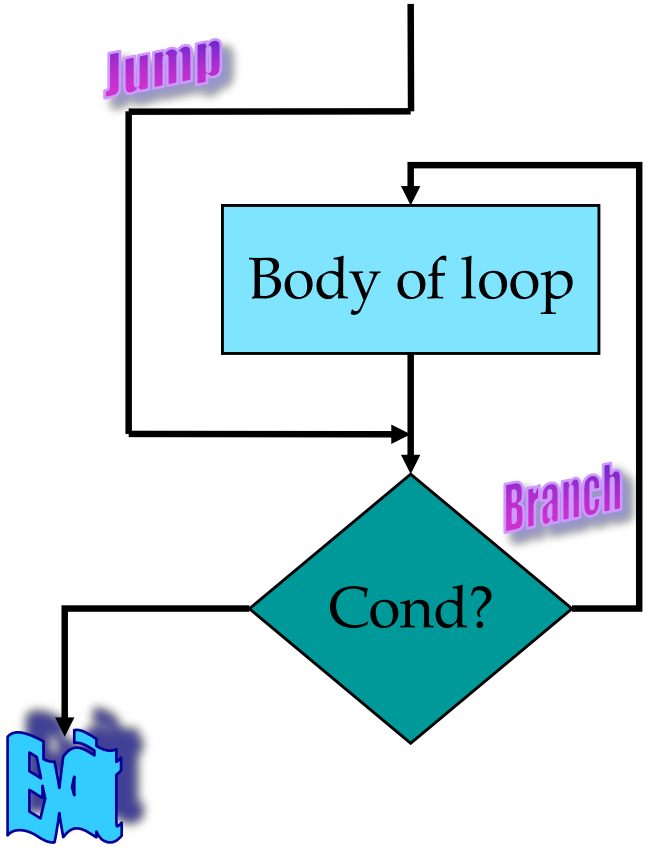
```
Loop:  sll xt0, x1, 2          # xt0 = 4 * i
       addu xt1, xt0, x3       # xt1 = &(A[i])
       lw xt2, 0(xt1)          # xt2 = A[i]
       bne xt2, x2, Exit       # goto Exit if !=
       addu x1, x1, x4         # i = i + j
       j Loop                  # goto Loop
Exit:
```

Improve Loop Efficiency

Two branches/iteration:



Better structure:



Improved Loop Solution

Remove extra jump from loop body

```
        j Cond                  # goto Cond
Loop:   addu x1, x1, x4         # i = i + j
Cond:   sll xt0, x1, 2          # xt0 = 4 * i
        addu xt1, x1, x3        # xt1 = &(A[i])
        lw xt2, 0(xt1)          # xt2 = A[i]
        beq xt2, x2, Loop       # goto Loop if ==
Exit:
```

Reduced loop from 6 to 5 instructions

Even small improvements important if loop executes many times

Switch Statements


```
typedef enum {ADD, MULT, MINUS, DIV,
   MOD, BAD}  op_type;

char unparse_symbol(op_type op)
{
  switch (op) {
  case ADD :
    return '+';
  case MULT:
    return '*';
  case MINUS:
    return '-';
  case DIV:
    return '/';
  case MOD:
    return '%';
  case BAD:
    return '?';
  }
}
```

- Implementation Options:

1. Series of conditional branches
   - Good if few cases; slow if many

2. Jump table to lookup branch target
   - Use j to unconditionally jump to address stored in a register
     - j dest # Jump to xdest
   - Avoids conditional branches
   - Possible when cases are small integer constants and dense

- C compiler picks based on case structure

Jump Table Structure

Switch Form

```
switch(op) {
  case val_0:
    Block 0
  case val_1:
    Block 1
    • • •
  case val_n-1:
    Block n–1
}
```

Jump Table

jtab:

Targ0
Targ1
Targ2
• • •
Targn-1

Jump Targets

Targ0: Code Block 0

Targ1: Code Block 1

• • •

Targn-1: Code Block n–1

Implementation

```
target = JTab[op];
goto *target;
```

Switch Statement Code


```
typedef enum
  {ADD, MULT, MINUS, DIV, MOD, BAD}
    op_type;
char unparse_symbol(op_type op)
{
  switch (op) {
    • • •
  }
}
```

Enumerated Values

ADD	**0**
MULT	**1**
MINUS	**2**
DIV	**3**
MOD	**4**
BAD	**5**

```
        slt    xt0, xa0, x0        # xt0 = 1 if op < 0
        bne    xt0, x0, Exit       # if op < 0 goto Exit
        slti   xt0, xa0, 6         # xt0 = 1 if op < 6
        beq    xt0, x0, Exit       # if op >= 6 goto Exit
        slli   xt1, xa0, 2         # xt1 = 4 * op
        add    xt2, xt1, xt4       # xt2 =&(JumpTable[op])
                                   # assumes xt4=JumpTable
        lw     xt3, 0(xt2)         # xt3 = JumpTable[op]
        jr     xt3                 # jump to JumpTable[op]
```

Procedure Call and Return

- Procedures are required for structured programming
  - Aka: functions, methods, subroutines, …
- Implementing procedures in assembly requires several things
  - Memory space must be set aside for local variables
  - Arguments must be passed in and return values passed out
  - Execution must continue after the call
- Just a single new instruction needed
  - A variation of jump (jump and link)
  - The rest is software (using instructions we already know)

Procedure Call Steps

1. Place parameters in a place where the procedure can access them
2. Transfer control to the procedure
   - Some kind of jump
3. Allocate the memory resources needed for the procedure
4. Perform the desired task (procedure body)
5. Place the result value in a place where the calling program can access it
6. Free the memory allocated in (3)
7. Return control to the point of origin

Call and Return Implementation

- Call: jump & link instruction (`jal target`)
  - Store PC+4 in register x1 (xra)
  - Jump to target
- Return: jr xra
- Arguments: x10 – x17
  - By software convention
- Return values: x10, x11
  - By software convention

Complications

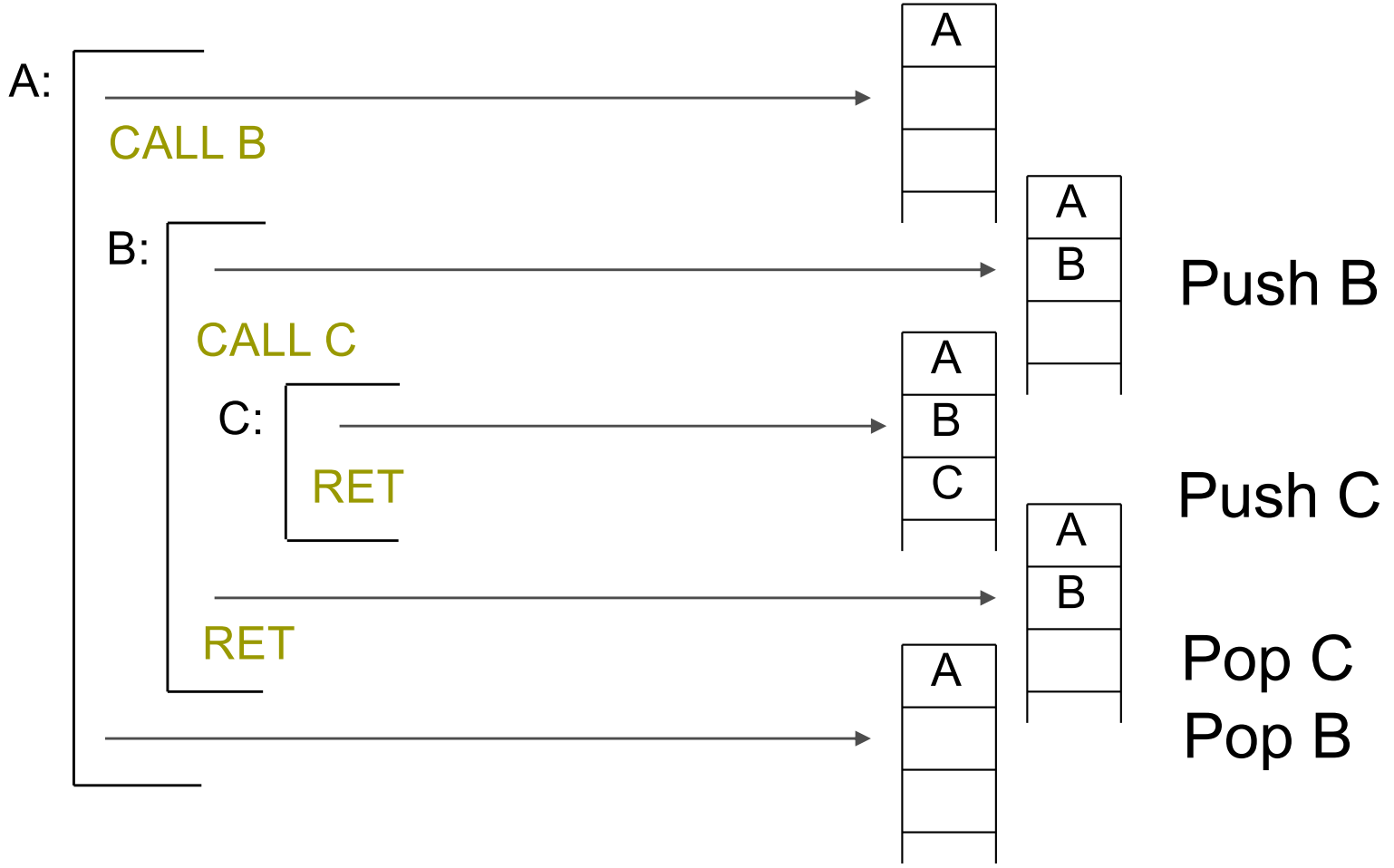
- What if function has >4 arguments?
  - Or an unknown number of arguments
- What if function returns >8 bytes?
- What if the function calls another function
  - A() → B() → C() →
  - Recursion: A() → A() → A() →

Stack-Based Languages

- Languages that support recursion (C, C++, Java, …)
  - Code must be “reentrant”
    - Multiple simultaneous instantiations of single procedure
  - Need someplace to store state of each instantiation
    - Arguments, local variables, return pointer
- Stack discipline (last in, first out)
  - State for given procedure needed for limited time
    - From when called to when return
  - Callee returns before caller does → LIFO
    - Stack allocated in frames

Nested Stacks

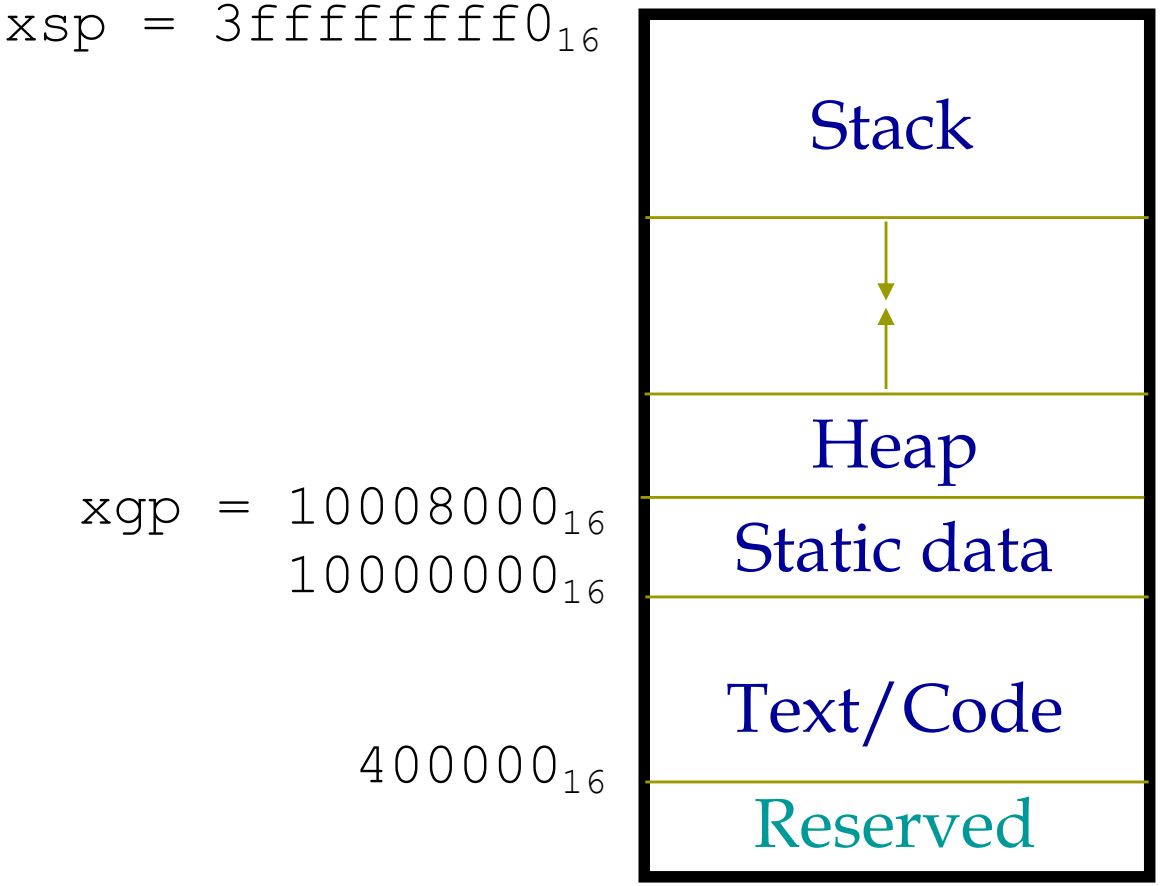
- The stack grows downward and shrinks upward



Call/Return Conventions

- Need common rules and layouts
  - To allow multi-procedure programs to work robustly
  - Procedures may be compiled and distributed separately
- Common conventions for RISC-V
  - Stack grows downwards
  - Callee is responsible for register spilling
    - Upon entry, push registers to stack
    - One register spill for each local variable
    - Before return, restore (pop) registers from stack
- No special connection to HW here
  - All implemented with regular instructions
  - Alternative conventions would probably work as well

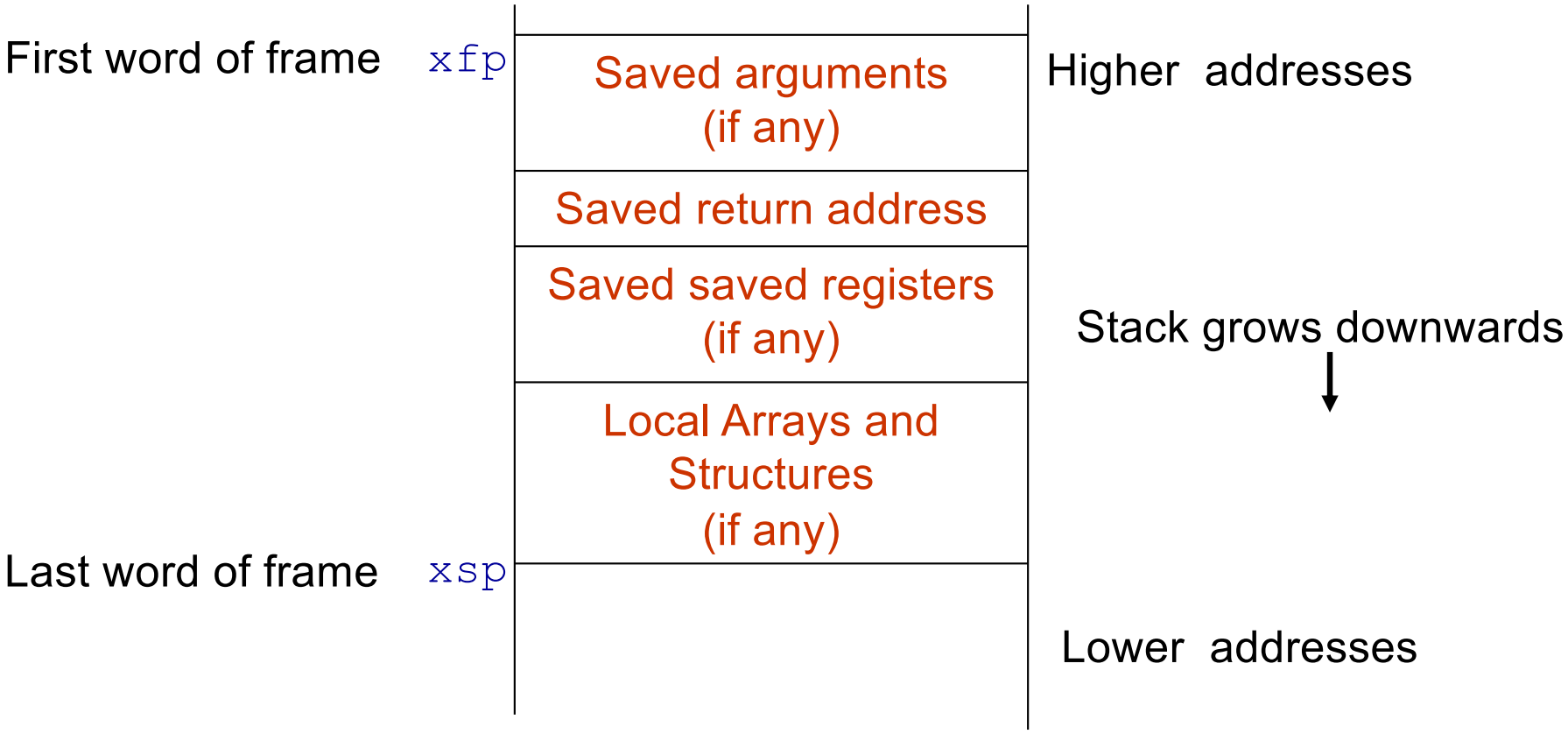
RISC-V Storage Layout (Software Convention)



Stack and heap grow towards one another to maximize storage use before collision

Procedure Activation Record or Stackframe

- Each procedure creates an activation record on stack



RISC-V Calling Convention

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
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

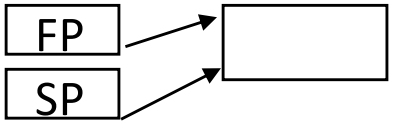
Table 20.2: RISC-V calling convention register usage.

Call and Return: the Details

- Caller
  - Save caller-saved registers `(ra, temporaries)` as needed
  - Load arguments in `x10-x17`, rest passed on stack
  - Execute `jal`
- Callee setup
  1. Allocate memory for new frame (`xsp` = `xsp` - frame)
  2. Save callee-saved registers `non-temporal`, `xfp`, `xra` **as needed**
  3. Set frame pointer (`xfp` = `xsp` + frame size - 4)
- Callee return
  - Place return value in `x10` and `x11`
  - Restore any callee-saved registers
  - Pop stack (`xsp` = `xsp` + frame size)
  - Return by `jr xra`
- Caller
  - Restore any caller-saved registers as needed

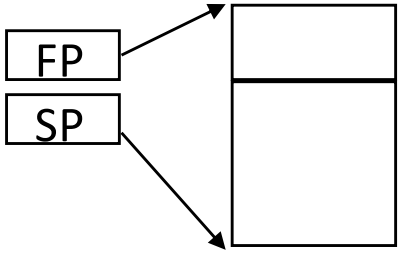
Calling Convention Steps

Before call:



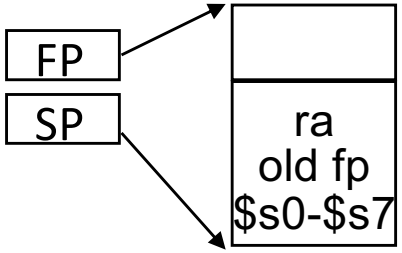
First four arguments passed in registers

Callee setup; step 1



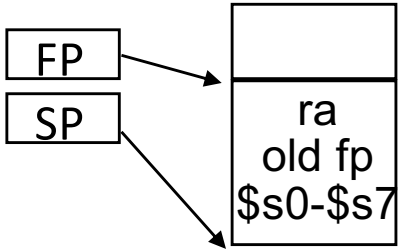
Adjust SP

Callee setup; step 2



Save registers as needed

Callee setup; step 3



Adjust FP

Simple Example


```
int foo(int num)
{
   return(bar(num + 1));
}

int bar(int num)
{
   return(num + 1);
}
```

```
foo:
        addiu   xsp, xsp, -32 # push frame
        sw      xra, 28(xsp)  # Save xra
        sw      xfp, 24(xsp)  # Save xfp
        addiu   xfp, xsp, 28  # Set new xfp
        addiu   x10, x10, 1   # num + 1
        jal     xra, bar      # call bar
        lw      xfp, 24(xsp)  # Restore xfp
        lw      xra, 28(xsp)  # Restore xra
        addiu   xsp, xsp, 32  # pop frame
        jr      xra           # return

bar:
        addiu   x10, x10, 1   # leaf procedure
        jr      xra           # with no frame
```

Factorial Example


```
int fact(int n)
{
   if (n <= 1)
      return(1);
   else
      return(n*fact(n-1));
}
```

```
fact:   slti  xt0, xa0, 2       # n < 2
        beq   xt0, xzero, skip  # goto skip
        ori   xv0, xzero, 1     # Ret val = 1
        jr    xra               # Return
skip:   addiu xsp, xsp, -32     # xsp down 32
        sw    xra, 28(xsp)      # Save xra
        sw    xfp, 24(xsp)      # Save xfp
        addiu xfp, xsp, 28      # Set up xfp
        sw    xa0, 20(xsp)      # Save n
        addui xa0, xa0, -1      # n - 1
        jal   fact              # Call recursive
link:   lw    xa0, 20(xsp)      # Restore n
        mul   xv0, xv0, xa0     # n * fact(n-1)
        lw    xra, 28(xsp)      # Load xra
        lw    xfp, 24(xsp)      # Load xfp
        addiu xsp, xsp, 32      # Pop stack
        jr    xra               # Return
```