COMPUTER ORGANIZATION

Lecture 3
RISC-V Procedure

2024 Spring

Recap

- Lecture2
  - RISC-V
    - Introduction
      - CISC
      - RISC
    - Register file
      - 32-32bits registers
      - x0~x31
    - Composition of Instruction
      - operator
      - operand
    - Design principles (regularity, small, common case, compromise
    - RISC-V is Little endian
      - LSByte at LSAddress
  - Instruction Type
    - Arithmetic
      - immediate (sign extended
    - Data transfer
      - byte offset
    - Logical
    - Branch/Jump
  - Data representation
    - unsigned/signed binary
    - 2's complement

Recap: Exercise

- For the following C statement, what is the corresponding RISC-V RV32 assembly code? Assume that the variables i is assigned to registers x13, respectively. Assume that the base address of the arrays A and B are in registers x16 and x17, respectively.

$$B[8] = A[i-3];$$

Conditional Operations

- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- Conditional branch
  - *beq rs1, rs2, L1*

    if (rs1 == rs2) branch to instruction labeled L1;
  - *bne rs1, rs2, L1*
    - if (rs1 != rs2) branch to instruction labeled L1;
- Unconditional branch
  - *beq x0, x0, L1*
    - unconditional jump to instruction labeled L1

Labels in Assembly

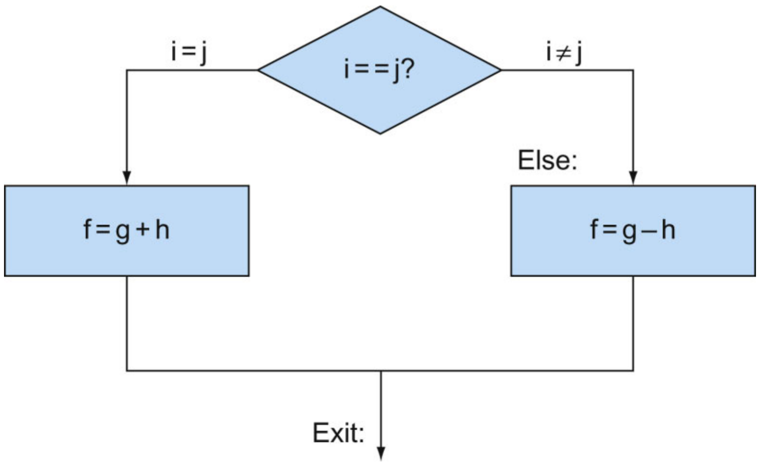
- We commonly see "labels" in the code
  - foo: add x2, x1, x0
- The assembler converts these into positions in the code
  - At what address in the code is that label …
- Labels give control flow instructions, such as jumps and branches, a place to go ...
  - e.g. bne x0, x2, foo
- The assembler in outputting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

Compiling If Statements

- C code

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

  - i and j are in x22 and x23,
  - f,g and h are in x19, x20 and x21



- Compiled RISC-V code:

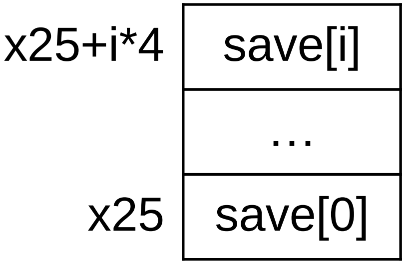
```
      bne x22, x23, Else  # go to Else if i ≠ j
      add x19, x20, x21   # f=g+h, skipped if i ≠ j
      beq x0, x0, Exit    # unconditional go to Exit
Else: sub x19, x20, x21   # f=g-h, skipped if i = j
Exit:
```

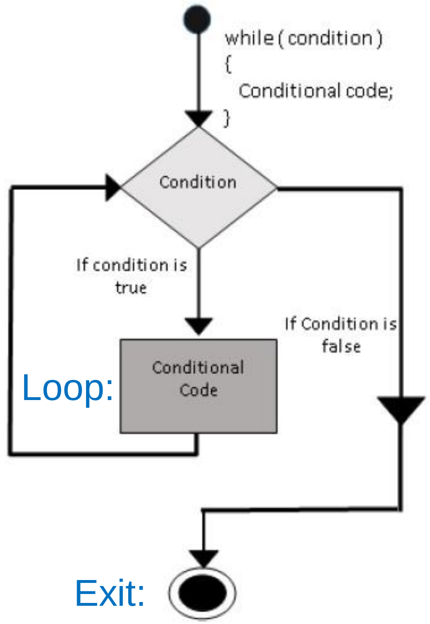
Compiling Loop Statements

- C code:

```
while (save[i] == k)
  i += 1;
```

  - i in x22, k in x24, address of save in x25





- Compiled MIPS code:

```
Loop: sll   x10, x22, 2     # Temp reg x10 = i * 4
      add   x10, x10, x25   # x10 = address of save[i]
      lw    x9, 0(x10)      # Temp reg x9 = save[i]
      bne   x9, x24, Exit   # go to Exit if save[i]≠k
      addi  x22, x22, 1     # i = i + 1
      j     Loop            # go to Loop
Exit:
```

More Conditional Operations

- Signed comparison
  - `blt rs1, rs2, L1`
    - if (rs1 < rs2) branch to instruction labeled L1
  - `bge rs1, rs2, L1`
    - if (rs1 >= rs2) branch to instruction labeled L1
  - Example, C to RISC-V
    - `if (a > b) a += 1;`
    - a in x22, b in x23

```
    bge  x23, x22, Exit # signed comparison
    addi x22, x22, 1
Exit:
```

- Unsigned comparison
  - `bltu, bgeu`

What if we need more instructions?

- RISC-V doesn’t have "branch if greater than” or “branch if less than or equal”
- Instead you can reverse the arguments, as:
  - A > B is equivalent to B < A
  - A <= B is equivalent to B >= A
- The assembler defines pseudo-instructions for your convenience:

  `bgt x2, x3, foo (pseudo)` will become
  `blt x3, x2, foo (basic)`

Pseudo-instructions

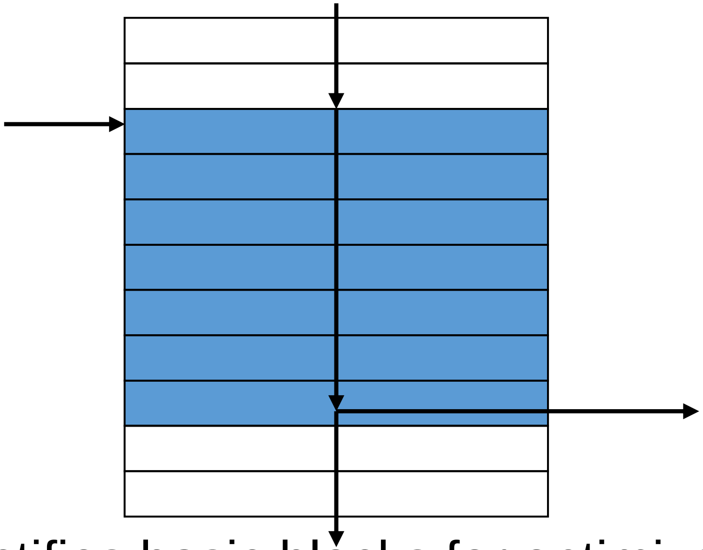
- For more pseudo-instructions, refer to RARS Help (see in lab).

Basic Instructions | Extended (pseudo) Instructions | Directives | Syscalls | Exc

lhu t1,10000000	Load Halfword Unsigned : Set t1 to zero-extended 16-bit value
lhu t1,label	Load Halfword Unsigned : Set t1 to zero-extended 16-bit value f
li t1,-100	Load Immediate : Set t1 to 12-bit immediate (sign-extended)
li t1,10000000	Load Immediate : Set t1 to 32-bit immediate
lui t1,%hi(label)	Load Upper Address : Set t1 to upper 20-bit label's address
lw t1,%lo(label)(t2)	Load from Address
lw t1,(t2)	Load Word : Set t1 to contents of effective memory word addres
lw t1,-100	Load Word : Set t1 to contents of effective memory word addres
lw t1,10000000	Load Word : Set t1 to contents of effective memory word addres
lw t1,label	Load Word : Set t1 to contents of memory word at label's addres
mv t1,t2	MoVe : Set t1 to contents of t2
neg t1,t2	NEGate : Set t1 to negation of t2
nop	NO OPeration
not t1,t2	Bitwise NOT (bit inversion)

Basic Blocks

- A basic block is a sequence of instructions with
  - No embedded branches (except at end)
  - No branch targets (except at beginning)



  - A compiler identifies basic blocks for optimization
  - An advanced processor can accelerate execution of basic blocks

C to RISC-V Example

```c
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
    slli 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++
    beq x0, x0, loop         # iterate
    exit:
```

C to RISC-V Example Optimized

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

Loop now has 6 instructions

```
# 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++
beq x0, x0, loop       # iterate
exit:
```

C to RISC-V Example Optimum

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

```
# 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
```

Loop now has 4 instructions

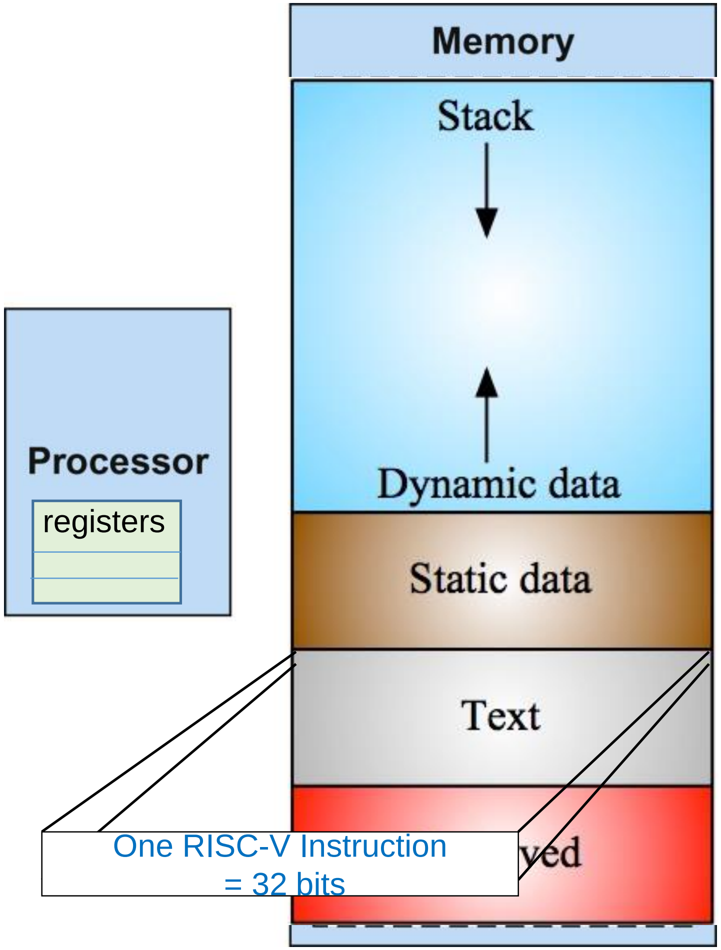
- Directly increment ptr into A array
- And only 1 branch/jump rather than two
  - Because first time through is always true so can move check to the end
  - The compiler will often do this automatically for optimization

Procedure Calling

- A procedure or function is one tool used by the programmers to structure programs
  - Benefit: easy to understand, reuse code
- We can think of a procedure like a spy
  - acquires resources → performs task → covers his tracks → returns back with desired result
- Six Steps of Calling a Function
  1. Put parameters in a place where the procedure can access them.
  2. Transfer control to the procedure.
  3. Acquire the storage resources needed for the procedure.
  4. Perform the desired task.
  5. Put the result value in a place where the calling program can access it.
  6. Return control to the point of origin, since a procedure can be called from several points in a program.

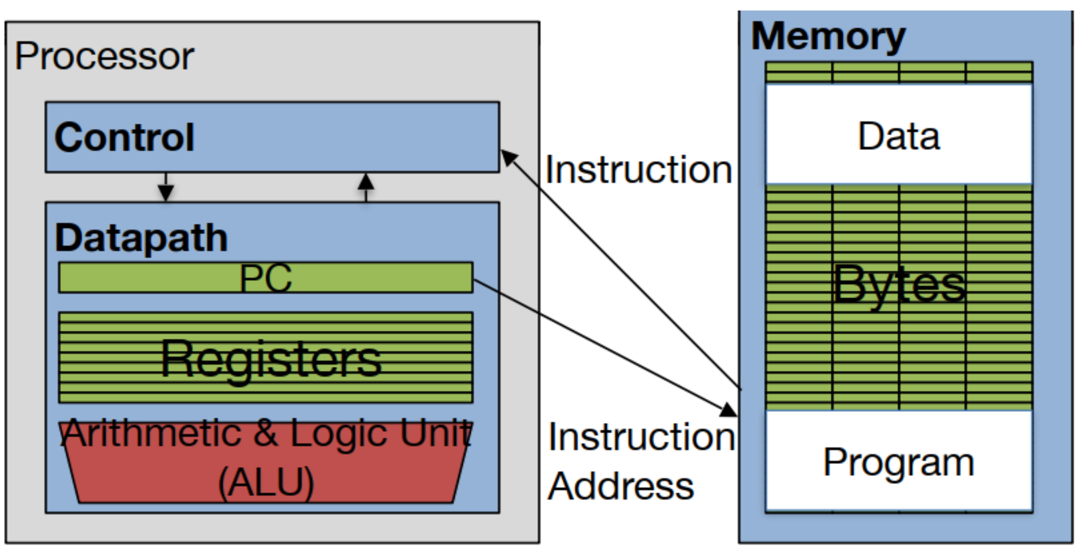
Recall: How Program is Stored

- Instructions(programs) are represented in binary, just like data
- Programs are stored in *Text Segment*
- Constants and other static variables are stored in *Static data segment*
- Dynamic data: *Heap*
  - E.g., malloc in C, new in Java
- Automatic data: *Stack*



Program Execution

- PC (program counter) is special internal register inside processor holding **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 program counter (default is add +4 bytes to PC, to move to next sequential instruction)



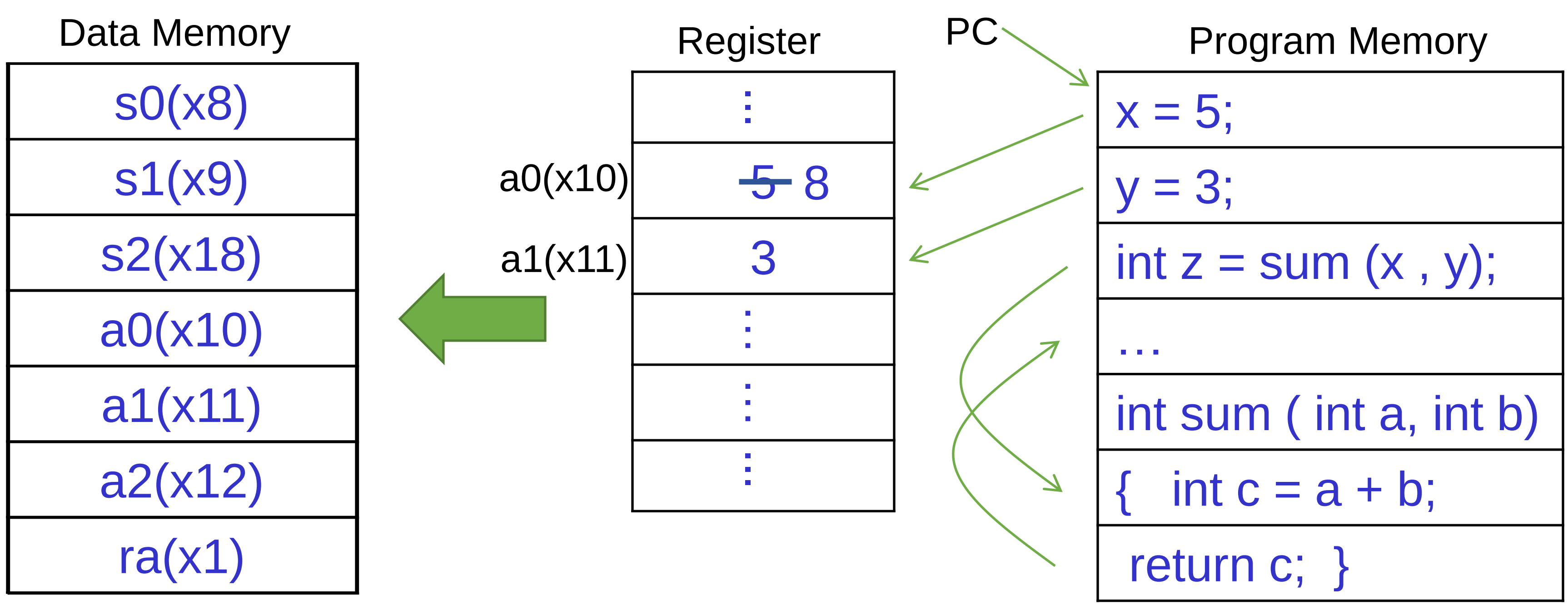
Procedure Calling

Caller:

```
int x = 5;
int y = 3;
int z = sum (x , y);
x = x + 7;
…
```

callee:

```
int sum ( int a, int b)
{
    int c = a + b;
    return c;
}
```



Procedure Calling: Question1

- Step 1, 5 and 6: Where should we put the arguments and return values?
  - Registers way faster than memory, so use them whenever possible
  - Symbolic register names
    - E.g., a0-a7 (x10-x17) for function arguments, a0-a1 for return values
    - E.g., ra (x1) for return address, used to save where a function is called from so we can get back
  - If need extra space, use memory (the stack!)

RISC-V Registers and Convention

- ABI: Application Binary Interface, defines “Calling Convention” – How to call other functions
- So going forward, no more x3, x6... type nomenclature

Register	ABI name	Description	Saved by Callee?
x0	zero	the constant value 0	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	Temporaries	No
x8	s0 / fp	Saved register/Frame pointer	Yes
x9	s1	Saved register	Yes
x10 – 17	a0 – 7	Function arguments/Return values	No
x18 – 27	s2 – 11	Saved registers	Yes
x28 – 31	t3 – 6	Temporaries	No

Procedure Calling: Question2

- Step 2 and 6： How do we Transfer Control?
  - Procedure call: jump and link (the real order is link and jump)

    jal ra, target #pseudo code: jal target
    - Address of following instruction (return address) put in ra
    - Jumps to target address
    - Used by Caller
  - Procedure return: jump register

    jalr zero, 0(ra) #pseudo code: jr ra
    - Similar to “Jump and Link” except in specification of target
    - Jump to ra and simultaneously saves the address of following instruction in zero register (value put into zero is meant to be throw away)
    - Used by Callee

Transfer Control

Caller:

```
int x = 5;  // x in s0
int y = 3;  // y in s1
int z = sum (x , y);
...
```

callee: // a in a0, b in a1, c in a0

```
int sum ( int a, int b) {
    int c = a + b;
    return c;
}
```

```
address (shown in decimal)
1000 mv a0,s0     # parse argument x → a
1004 mv a1,s1     # parse argument y → b
1008 jal sum      # ra=1012, goto sum
1012 …            # next instruction
…
2000 sum: add a0,a0,a1
2004 jr    ra     # next instr at 1012…
```

Note: these are pseudo code

PC's decimal value: …→1000→1004→1008→2000→2004→1012→…

Transfer Control

```
Caller:
        int x = 5;  // x in s0
        int y = 3;  // y in s1
        int z = sum (x , y);
        …
```

```
callee: // a in a0, b in a1, c in a0
    int sum ( int a, int b) {
        int c = a + b;
        return c;
    }
```

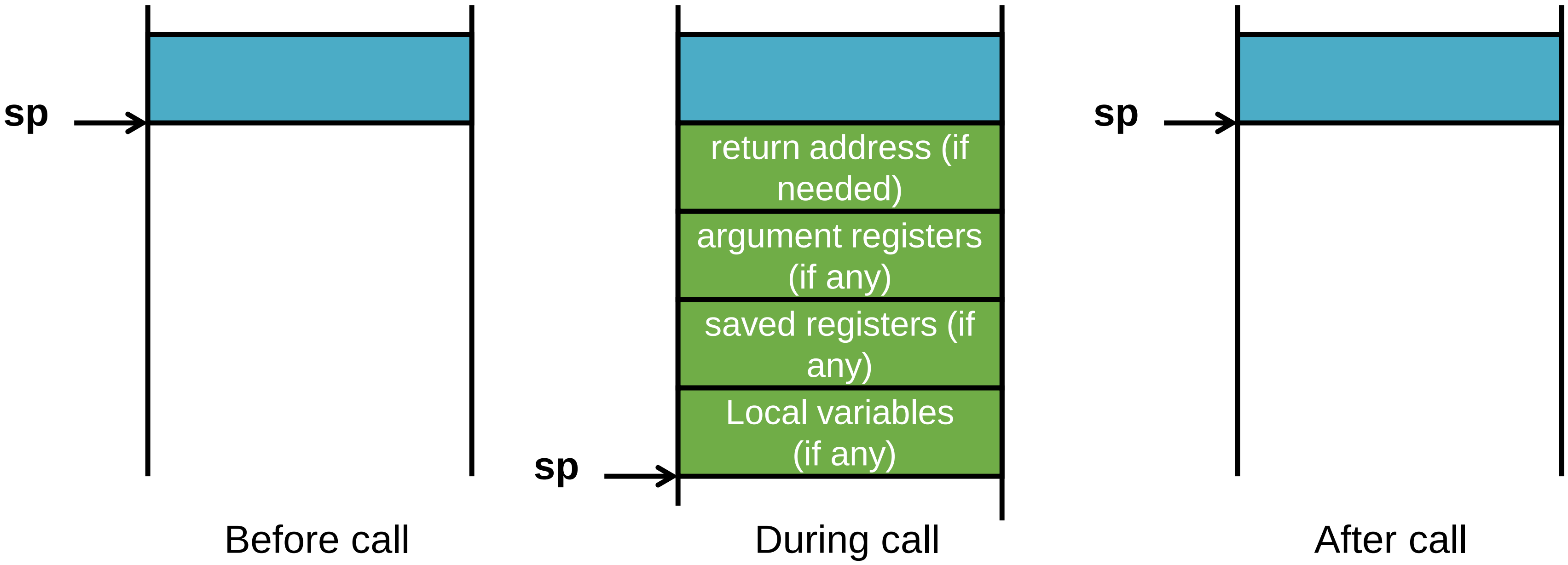
- Question: Why use jr here? Why not use j?
- Answer: sum might be called by many places, so we can't return to a fixed place. The calling proc to sum must be able to say "return here" somehow.

```
2000 sum: add a0,a0,a1
2004 jr   ra      # jump to the return address
```

Procedure Calling: Question3

- Step 3: What's the Local storage for variables?
  - C has three storage classes:
    - Stack: automatic variables are local to function and discarded when function exits
    - Static: variables exist across exits from and entries to procedures
    - Heap: variables declared dynamically via malloc
  - Use stack for automatic (local) variables that don't fit in registers
    - Push: placing data onto stack
    - Pop: removing data from stack
  - sp (x2) (stack pointer) points to the last used place at the stack
    - Push decrements sp
    - Pop increments sp

Stack Before, During, After Call



Procedure Calling: Question 4

- Step 4: Function Calling Conventions?
  - It is effectively a contract between functions
  - By convention, registers are classified as one of …
  - **caller-saved**
    - 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**
    - The function invoked must restore them before returning (if used)

The Calling Convention

- Caller saved:
  - **a0–a7** (**x10-x17**): eight argument registers to pass parameters and two return values (**a0-a1**)
  - **t0-t6** Temporaries
  - **ra**: one return address register for return to the point of origin
- Callee saved:
  - **s0-s11** Saved registers: Preserved across function calls
- If both the Caller and Callee obey the procedure conventions, there are significant benefits
  - People who have never seen or even communicated with each other can write functions that work together
  - Recursion functions work correctly

Leaf Procedure 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.
- Return variable f uses register a0
- Assume we compute f by using s0 and s1
- s0 and s1 are callee saved, so it’s the responsibility of “Leaf” to save and restore

Leaf Procedure Example

- RISC-V code:

a0, a1, a2, a3

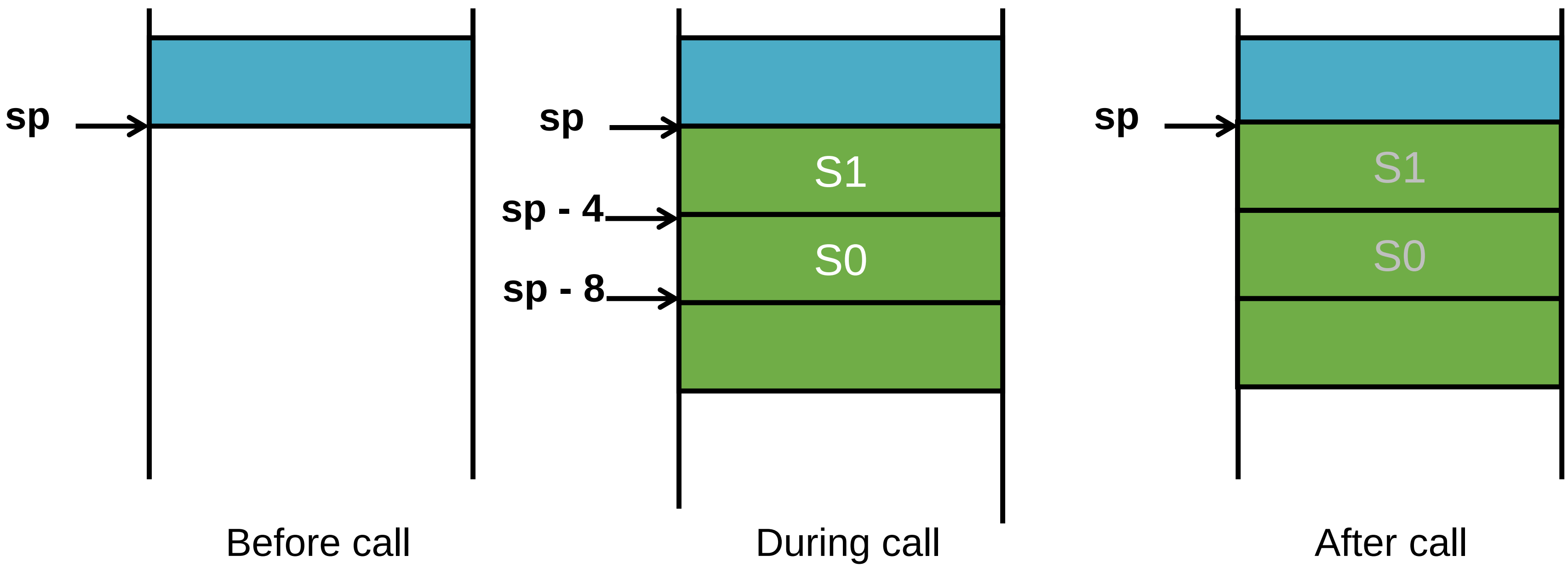
```c
int Leaf(int g, int h, int i, int j)
  { int f;
    f = (g + h) - (i + j);
    return f;
  }
```

a0 s0 s1

```
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
              # psudo of jalr zero, 0(ra)
```

Leaf Procedure Example: Stack

- In the previous example we need to save old values of s0 and s1
- Push doesn't actually delete content from memory, it just increase the stack pointer



Leaf Procedure Example: Observation

- This is a "leaf function": it calls no other function
  - We didn't need to save ra (because leaf didn't call any other function and therefore ra never changed
  - 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)
jr ra
```

Caller saved:
a0–a7
t0-t6
ra
Callee saved:
s0-s11

Non-Leaf Procedures

- Procedures that call other procedures (nested call)
  - Note: this could mean a function calling itself – recursion
- For nested call, caller needs to save on the stack:
  - Its return address
  - Any arguments and temporaries needed after the call
- Restore from the stack after the call

Non-Leaf Procedure Example

- C code:

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

- Function **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

Non-Leaf Procedure Example

a0, a1

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

a0, a1

```
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      # prepare 2nd argument for mult
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:
```

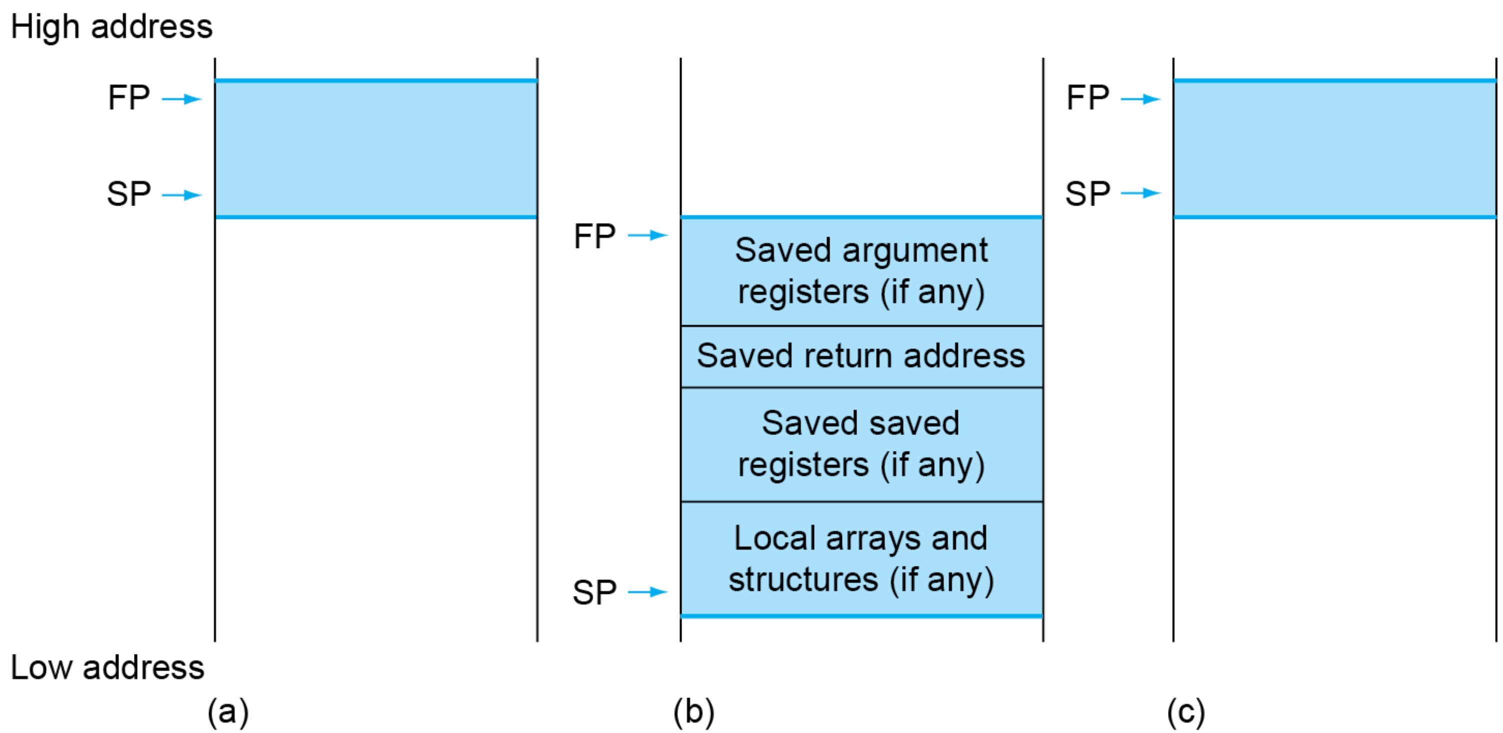
Push

Pop

Register Conventions Summary

- CalleR must save arguments and temporary registers it is using onto the stack before making a procedure call
- CalleR can trust saved registers to maintain values
- CalleE must save any saved registers it intends to use by putting them on the stack before overwriting their values
- Notes:
  - CalleR and calleE only need to save the **appropriate using** (not all!)
  - Don't forget to restore the values later

Local Data on the Stack



- Local data allocated by callee
  - e.g., C automatic variables
- Procedure frame (activation record)
  - Used by some compilers to manage stack storage

Summary

- Functions called with jal, return with jalr (jr)
- 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
  - Memory
  - Conditional Branches
  - Unconditional Branches (Jumps)
- Registers

Non-Leaf Procedure Example 2

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

```
fact:
  addi      sp, sp, -8
  sw        ra, 4(sp)
  sw        a0, 0(sp)
  addi      t0, a0, -1
  bge       t0, zero, L1
    addi    a0, zero, 1
    addi    sp, sp, 8
    jr      ra
L1:
  addi      a0, a0, -1
  jal       fact
  addi      t1,a0,0
  lw        a0, 0(sp)
  lw        ra, 4(sp)
  addi      sp, sp, 8
  mul       a0, a0, t1
  jr        ra
```

Compare n<1

Return 1

Fact(n-1)

Return n*fact(n-1)

Notes:

The caller saves a0 and ra in its stack space.

Temps are never saved.

Non-Leaf Procedure Example 2

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

- RISC-V code: suppose n=2

```
     fact:
1       addi   sp, sp, -8
2       sw     ra, 4(sp)
3       sw     a0, 0(sp)
4       addi   t0, a0, -1
5       bge    t0, zero, L1
6       addi   a0, zero, 1
7       addi   sp, sp, 8
8       jr     ra
9    L1:addi   a0, a0, -1
10      jal    fact
11      mv     t1,a0
12   Y: lw     a0, 0(sp)
13      lw     ra, 4(sp)
14      addi   sp, sp, 8
15      mul    a0, a0, t1
16      jr     ra
```

X: return address of caller (not shown in the code)
Y: the address of line 12