信息科学与技术学院
School of Information Science and Technology

CS 110
Computer Architecture
Intro to RISC-V I

Instructors:

Siting Liu & Chundong Wang

Course website: https://toast-lab.sist.shanghaitech.edu.cn/courses/CS110@ShanghaiTech/Spring-2023/index.html

School of Information Science and Technology (SIST)

ShanghaiTech University

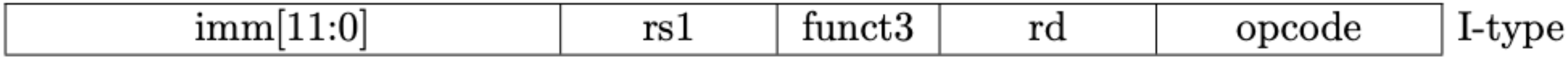
2023/2/6

Course Info

- Lab 3 will be released after class (10 a.m.), get yourself prepared before going to lab sessions!
- Our project 1.1 will be available this weekend, and will be marked in lab sessions. Deadline March 13th.
- Next week discussion on RISC-V related materials and assembly coding.

Assembly Instructions

- Different types of instructions



imm[11:0]	rs1	funct3	rd	opcode	I-type

- I-type
  - Register-Immediate type
  - Has two operands (one accessed from source register, another a constant/ immediate) and one output (saved to destination register)
  - Can do arithmetic/logic/load from main memory/jump (covered later)

RV32I I-type Arithmetic

immediate => 12 bits

immediate 立即数

- Syntax of instructions: assembly language
  - Addition: `addi rd, rs1, imm`

    two's complement

    Adds `imm` to `rs1`, stores the result to `rd`, and `imm` is a signed number.
  - Example: `addi x5, x4, 10` $x_5 = x_4 + 10$

    `addi x6, x4, -10` → 1111···10110

    x_4: 0000···00011
- Similarly, `andi/ori/xori/slti/sltiu`
  - All the imm's are sign-extended ✗

    符号位拓展

sltiu rd rs1 1 ⇒ 判断一个数是否为0

- `slli/srli/srai` are special (defined in RV64I), and can be extended to RV32I usage (RTFM)

Registers

0	x0/zero
0x12340000	x1
0x00006789	x2
0xFFFFFFFF	x3
0x3	x4
0xd	x5
0xFFFFFFF9	x6
	x7

x_6 1111····1001 (-7)

RV32I Arithmetic/Logic Test

```
addi x1, x2, -1
or   x2, x2, x1
add  x3, x1, x2
slt  x4, x3, x1
sra  x5, x3, x4
sub  x0, x5, x4
```

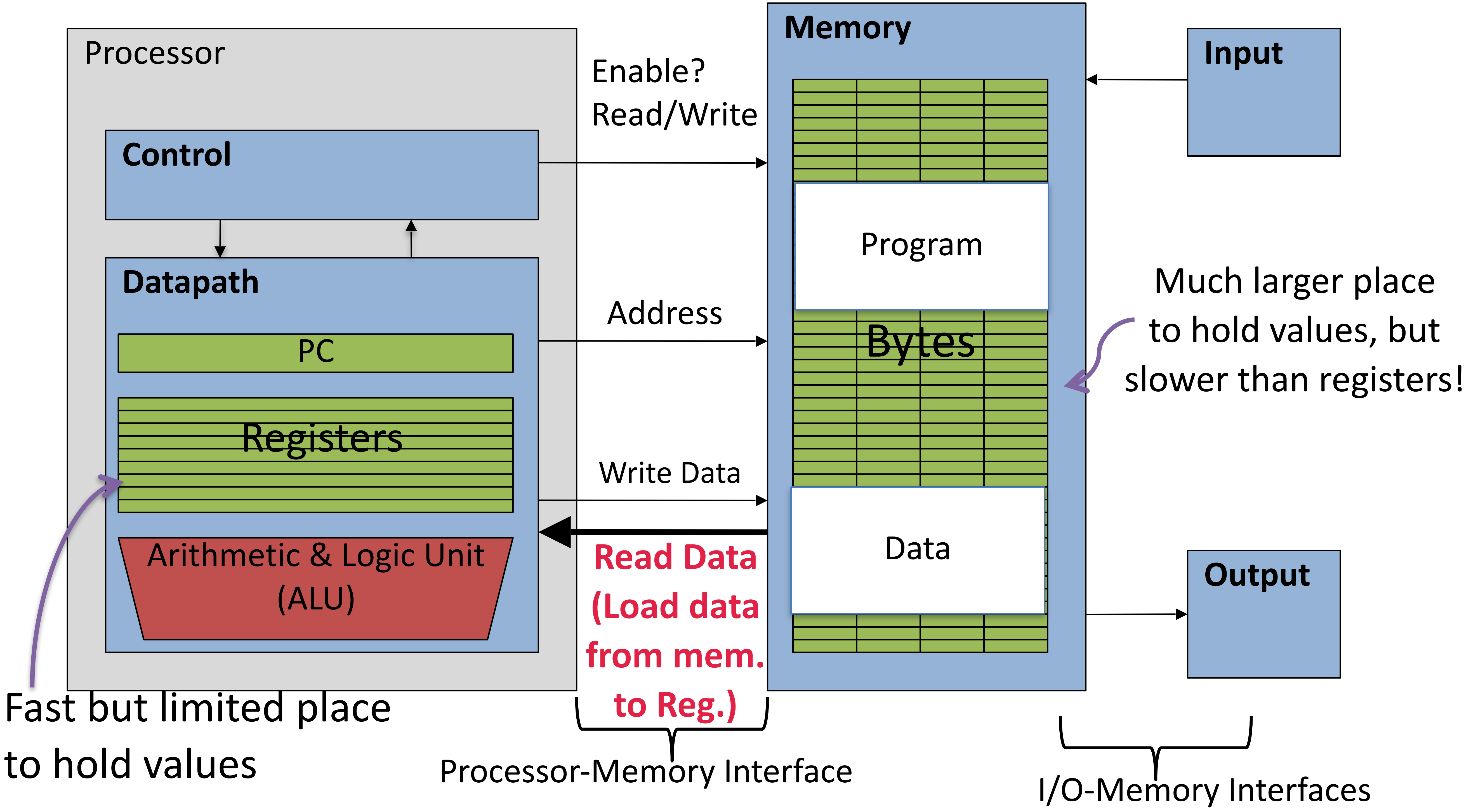
x0不会被改变
始终为0

- Register zero (`x0`) is 'hard-wired' to 0;
- By **convention** RISC-V has a specific no-op instruction…
  - **`add x0 x0 x0`**
  - You may need to replace code later: No-ops can fill space, align data, and perform other options
  - Practical use in jump-and-link operations (covered later)

Registers ⇒ 0

0 → 0xFFFFFFFE	x0/zero
0 → 0xFFFFFFFF	x1 ⇒ -1
0 → 0xFFFFFFFF	x2
0 → 0xFFFFFFFE	x3 ⇒ -
0 → 0x1	x4
0 → 0xFFFFFFFF	x5
0	x6
0	x7

RV32I I-type Load



Much larger place to hold values, but slower than registers!

Fast but limited place to hold values

en.wikipedia.org/wiki/Big_endian

Big Endian vs. Little Endian

Big-endian and little-endian from Jonathan Swift's *Gulliver's Travels*

- **The order in which BYTES are stored in memory**
- **Bits always stored as usual. (E.g., 0xC2=0b 1100 0010)**

Consider the number 1025 as we normally write it:

BYTE3	BYTE2	BYTE1	BYTE0
00000000	00000000	00000100	00000001

大端 Big Endian

ADDR3	ADDR2	ADDR1	ADDR0
BYTE0	BYTE1	BYTE2	BYTE3
00000001	00000100	00000000	00000000

低 → 高

Examples

Names in the West (e.g. Siting, Liu)

Java Packages: (e.g. org.mypackage.HelloWorld)

Dates done correctly ISO 8601 YYYY-MM-DD (e.g. 2020-03-22)

Eating Pizza crust first

Unix file structure (e.g., /usr/local/bin/python)

”Network Byte Order”: most network protocols

IBM z/Architecture; very old Macs

Little Endian 小端

ADDR3	ADDR2	ADDR1	ADDR0
BYTE3	BYTE2	BYTE1	BYTE0
00000000	00000000	00000100	00000001

高 ← 低

Examples

Names in China (e.g. LIU Siting)

Internet names (e.g. sist.shanghaitech.edu.cn)

Dates written in England DD/MM/YYYY (e.g. 22/03/2020)

Eating Pizza skinny part first (the normal way)

默认RISC-V为小端

CANopen

Intel x86, **RISC-V** (can also support big-endian)

big-endian: MIPS, IA-64, PowerPC

Assembly Instructions—Load

- RV32I is a load-store architecture, where only load and store instructions access memory and arithmetic instructions only operate on CPU registers.

imm[11:0]	rs1	funct3	rd	opcode	
					I-type

Offset Base

- `lw rd, imm(rs1) :` Load word at `addr.` to register `rd`

`addr.= (number in rs1) + imm`

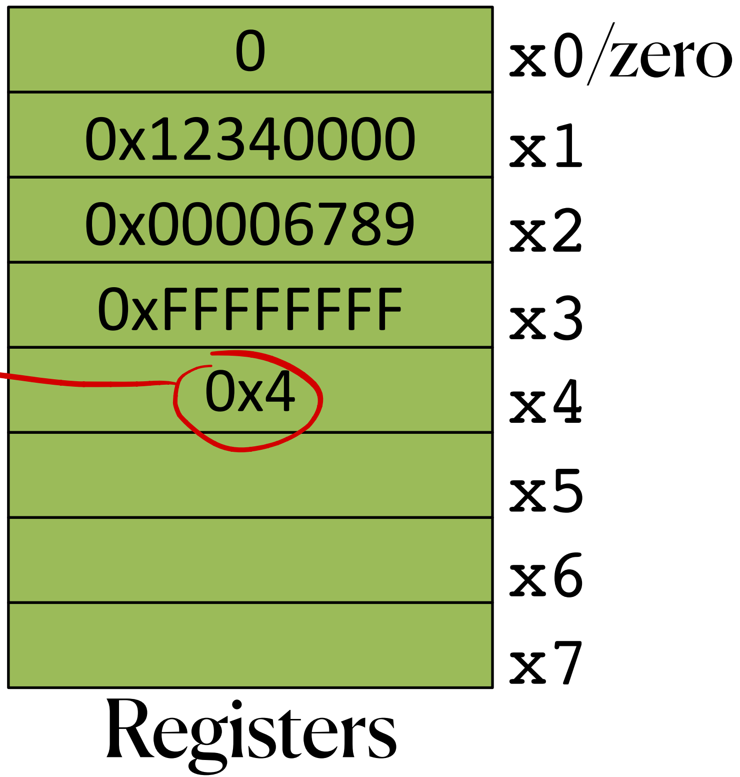
lw load word ⇒ 4 bytes

- Example

`lw x1, 12(x4)`

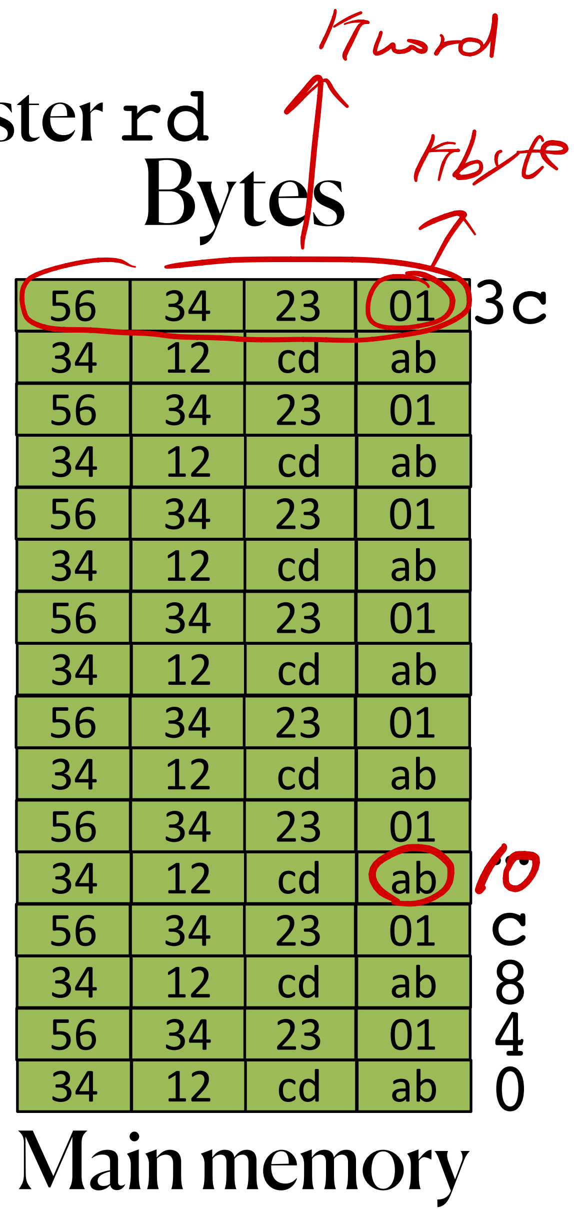
`addr.= 4 + 12 = (10)`$_{\text{HEX}}$

$\Rightarrow x_1 = 0x3412cdab$



0	x0/zero
0x12340000	x1
0x00006789	x2
0xFFFFFFFF	x3
0x4	x4
	x5
	x6
	x7

Registers



1 word

1 byte

Bytes

56	34	23	01	3c
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	10
56	34	23	01	c
34	12	cd	ab	8
56	34	23	01	4
34	12	cd	ab	0

Main memory

Assembly Instructions—Load

- RV32I is a load-store architecture, where only load and store instructions access memory and arithmetic instructions only operate on CPU registers.

imm[11:0]	rs1	funct3	rd	opcode	I-type

- `lw rd, imm(rs1) :` Load word at `addr.` to register `rd`

`addr.= (number in rs1) + imm`

- Example

```
lw x1, 12(x4)
```

`addr.= 4 + 12 =` (10)$_{\text{HEX}}$

- C code example

```
int A[100];
/*assume &A[0] = 4*/
G = A[3];
/*load G to x1/
```

lw G 3x4(&A[0])

Registers

0	x0/zero
0x12340000	x1
0x00006789	x2
0xFFFFFFFF	x3
0x4	x4
	x5
	x6
	x7

Bytes

Main memory

56	34	23	01	3c
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	…
56	34	23	01	
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	10
56	34	23	01	c
34	12	cd	ab	8
56	34	23	01	4
34	12	cd	ab	0

Assembly Instructions—Load

- RV32I is a load-store architecture, where only load and store instructions access memory and arithmetic instructions only operate on CPU registers.

imm[11:0]	rs1	funct3	rd	opcode	I-type

lb : load byte　　1 byte = 8 bits，前24位的填充

- `lb/lbu rd, imm(rs1) :` Load signed/unsigned byte at `addr.` to register `rd`

lb　　lbu 全0

取决于byte第1位

`addr.= (number in rs1) + imm`

- Example

```
lb x1, 12(x4)
addr.= 4 + 12 = (10)HEX
```

$x_1 = 0x\underline{FFFFFF}\,\underline{AB}$

```
lbu x1, 12(x4)
```

$x_1 = 0x000000AB$

Registers	
0	x0/zero
0x12340000	x1
0x00006789	x2
0xFFFFFFFF	x3
0x4	x4
	x5
	x6
	x7

Main memory				
56	34	23	01	3c
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	…
56	34	23	01	
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	10
56	34	23	01	c
34	12	cd	ab	8
56	34	23	01	4
34	12	cd	ab	0

1010
1011

Assembly Instructions—Load

- RV32I is a load-store architecture, where only load and store instructions access memory and arithmetic instructions only operate on CPU registers.

imm[11:0]	rs1	funct3	rd	opcode	I-type

load halfword

1 word → 4 byte → 32 bits

- `lh/lhu rd, imm(rs1)` : Load signed/unsigned halfword at `addr.` to register `rd` (similar to lb/lbu)

16 bits

`addr.= (number in rs1) + imm`

- Example

`lh x1, 12(x4)`

`addr.= 4 + 12 = (10)HEX`

0xFFFF CDAB

`lhu x1, 12(x4)`

Registers	
0	x0/zero
0x12340000	x1
0x00006789	x2
0xFFFFFFFF	x3
0x4	x4
	x5
	x6
	x7

Main memory				
56	34	23	01	3c
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	…
56	34	23	01	
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	10
56	34	23	01	c
34	12	cd	ab	8
56	34	23	01	4
34	12	cd	ab	0

c

1100

Assembly Instructions—S-Type Store

- RV32I is a load-store architecture, where only load and store instructions access memory and arithmetic instructions only operate on CPU registers.

imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type

- `sw rs2, imm(rs1) :` Store word at `rs2` to memory `addr.`

```
addr.= (number in rs1) + imm
```

- Example

```
sw x1, 12(x4)
```

`addr.= 4 + 12 =` $(10)_{\text{HEX}}$

- C code example

```
int  A[100];
/* &A[0] => x4 */
A[3] = h;
/* h in rs2 => x1 */
```

0	x0/zero
0x12340000	x1
0x00006789	x2
0xFFFFFFFF	x3
0x4	x4
	x5
	x6
	x7

Registers

56	34	23	01	3c
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	…
56	34	23	01	
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	10
56	34	23	01	c
34	12	cd	ab	8
56	34	23	01	4
34	12	cd	ab	0

Main memory

Assembly Instructions—S-Type Store

- RV32I is a load-store architecture, where only load and store instructions access memory and arithmetic instructions only operate on CPU registers.

imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type

Store word

- `sw rs2, imm(rs1)` : Store word at `rs2` to memory `addr.`

`addr.= (number in rs1) + imm`

- Example

`sw x1, 12(x4)`

`addr.= 4 + 12 =` $(10)_{HEX}$

- Similarly,

`sh`: Store lower 16 bits at `rs2`

`sb`: Store lower 8 bits at `rs2`

0	x0/zero
0x12340000	x1
0x00006789	x2
0xFFFFFFFF	x3
0x4	x4
	x5
	x6
	x7

Registers

56	34	23	01	3c
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	…
56	34	23	01	
34	12	cd	ab	0000 0004
56	34	23	01	
34	12	cd	ab	10
56	34	23	01	c
34	12	cd	ab	8
56	34	23	01	4
34	12	cd	ab	0

Main memory

Memory Alignment

- RISC-V *does not require* that integers be word aligned…
  - But it can be very very bad if you don't make sure they are…
- Consequences of unaligned integers
  - Slowdown: The processor is allowed to be a lot slower when it happens
    - In fact, a RISC-V processor may natively only support aligned accesses, and do unaligned-access in software!
      An unaligned load could take hundreds of times longer!
    - Lack of atomicity: The whole thing doesn't happen at once… can introduce lots of very subtle bugs

不强制 align

- So in practice, RISC-V *recommends* integers to be aligned on 4- byte boundaries; halfword 2-byte boundaries

Question! What's in x12?

0100 1111 0110

x_{11} = 0x0000 04F6

```
addi x11,x0,0x4F6
sw x11,0(x5)
lb x12,1(x5)
```

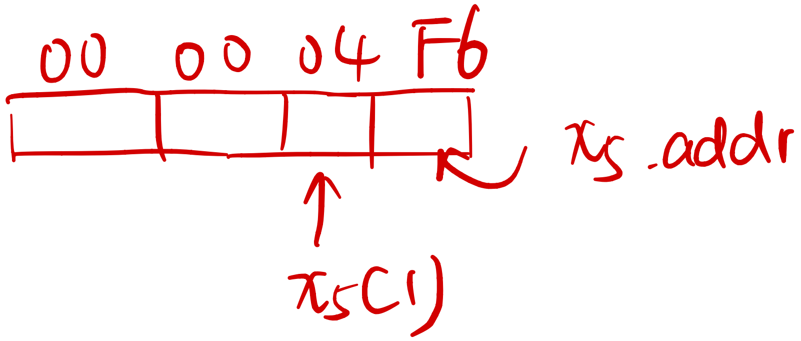
A: 0x0

B: [✓ 0x4]

C: 0x6

D: 0xF

E: 0xFFFFFFFF



Question! What's in x12?

0xFFFF85F6

1000 0101 1111 0110

```
addi x11,x0,0x85F6
sw x11,0(x5)
lb x12,1(x5)
```

A: 0x8

B: 0x85

C: 0xC

D: 0xBC

E: 0XFFFFFF85

F: 0XFFFFFFF8

G: 0XFFFFFFFC

H: 0XFFFFFFBC

Summary

- RISC-V ISA basics: (32 registers, referred to as `x0-x31, x0=0`)
- Simple is better
- One instruction (simple operation) per line (RISC-V assembly)
- Fixed-length instructions (for RV32I)
- 6 types of instructions (depending on their format)
- Instructions for arithmetics, logic operations, register-memory data exchange (load/store word/halfword/byte)
- RISC-V is little-endian
- Load-store architecture

CS 110
Computer Architecture
Intro to RISC-V II
Computer Decision Making

Instructors:

Siting Liu & Chundong Wang

Course website: https://toast-lab.sist.shanghaitech.edu.cn/courses/CS110@ShanghaiTech/Spring-2023/index.html

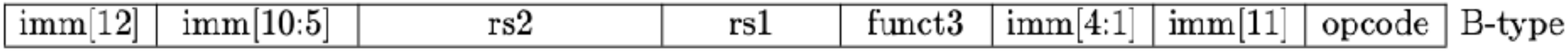
School of Information Science and Technology (SIST)

ShanghaiTech University

2023/2/6

Computer Decision Making—Branch

- Normal operation: execute instructions in sequence
- In C: `if/while/for`-statement; function call
- RISV-V provides conditional branch (B-type) & unconditional jump (j)



imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode	B-type

- RISC-V: similar `if`-statement instruction

  `beq rs1, rs2, L(imm/label)` branch equal

  means: go to statement labeled if (value in `rs1`) == (value in `rs2`);

  otherwise, go to next statement
- `beq` stands for branch if equal
- Similarly, `bne` for branch if not equal

Computer Decision Making—Branch

- Example:
  `beq rs1, rs2, L(imm/label)`
- C code

```
int main(void) {
    int i=5;
    if (i!=6){
        i++;
    }
    else i--;
   return 0;
}
```

- Assembly

```
  addi x2, x0, 5     5→x2
  addi x3, x0, 6     6→x3
  bne  x2, x3, L1
  beq  x2, x3, L2
L1:addi x2, x2, 1        i++
  ret (kind of jump)
L2:addi x2, x2, -1       i--
  ret
```

- Label can also point to data (more in discussion)

Computer Decision Making—Branch

- Example:
  `beq rs1, rs2, L(imm/label)`

- C code

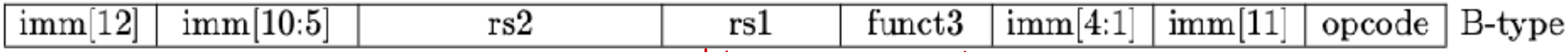
```
int main(void) {
    int i=5;
    if (i!=6){
        i++;
    }
    else i--;
   return 0;
}
```

- Assembly (real stuff in ARM64)

```
    mov w8, #5
Ltmp3:
    .loc1 10 9 is_stmt 0
    subs w8, w8, #6
    b.eq LBB0_2
    b    LBB0_1
LBB0_1:
Ltmp4:
    .loc1 11 10 is_stmt 1
    ldr w8, [sp, #8]
    add w8, w8, #1
    str w8, [sp, #8]
    .loc1 12 5
    b   LBB0_3
Ltmp5:
LBB0_2:
    .loc1 13 11
    ldr  w8, [sp, #8]
    subs w8, w8, #1
    str  w8, [sp, #8]
    b    LBB0_3
Ltmp6:
LBB0_3:
    .loc1 0 11 is_stmt 0
    mov w0, #0
    .loc1 14 5 is_stmt 1
    add sp, sp, #16
    ret
```

Computer Decision Making—Branch

- Normal operation: execute instructions in sequence
- In programming languages: `if/while/for`-statement
- RISV-V provides conditional branch & unconditional jump



imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode	B-type

less than greater than or equal

- RISC-V: `if`-statement instructions are

  `blt/bltu/bge/bgeu rs1, rs2, L(imm/label)`

  means: go to statement labeled L1 if (value in `rs1`) </≥ (value in `rs2`) using singed/unsigned comparison; otherwise, go to next statement

C Loop Mapped to RISC-V Assembly

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

```
# Assume x8 holds pointer to A
# Assign x10=sum                        x8
  add x9, x8, x0   # x9=&A[0]
  add x10, x0, x0  # sum=0
  add x11, x0, x0  # i=0 → x11
  addi x13,x0, 20  # x13=20 → i<20
Loop:     i ≥ 20
  bge x11,x13,Done
  lw x12, 0(x9)     # x12=A[i]
  add x10,x10,x12  # sum+=
  addi x9, x9,4     # &A[i+1]
  addi x11,x11,1    # i++
  j Loop
Done:
  → jump
```

Optimization

- The simple translation is sub-optimal!
  - Inner loop is now 4 instructions rather than 7
  - 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
  - See that i is only used as an index in a loop

```
# Assume x8 holds pointer to A
# Assign x10=sum
add  x10, x0, x0 # sum=0
add  x11, x8, x0 # ptr = A
addi x12,x11, 80 # end = A + 80
Loop:
    lw   x13,0(x11)   # x13 = *ptr
    add  x10,x10, x13 # sum += x13
    addi x11,x11, 4   # ptr++
blt x11, x12, Loop:  # ptr < end
```

- This optimization is not required
- Line by line translation is good
- Correctness first, performance second

Arrays and Pointers

```
int  i;
int  array[10];

for (i = 0; i < 10; i++)
{
  array[i] = …;
}
```

```
int *p;
int  array[10];

for (p = array; p < &array[10]; p++)
{
  *p = …;
}
```

These code sequences have the same effect!

Translate Assembly

```
addi   x10, x0, 0x7
add    x12, x0, x0
label_a:
andi   x14, x10, 1
beq    x14, x0, label_b
add    x12, x10, x12
label_b:
addi   x10, x10, -1
bne    x10, x0, label_a
```

```
x10 = 7
x12 = 0
label_a: x14 = x10 & 1
if (x14!=0)
{x12 = x10+x12;}
 label_b: x10 = x10-1;
if (x10!=0)
{go to label_a;}
```

Call a Function—Unconditional Jump

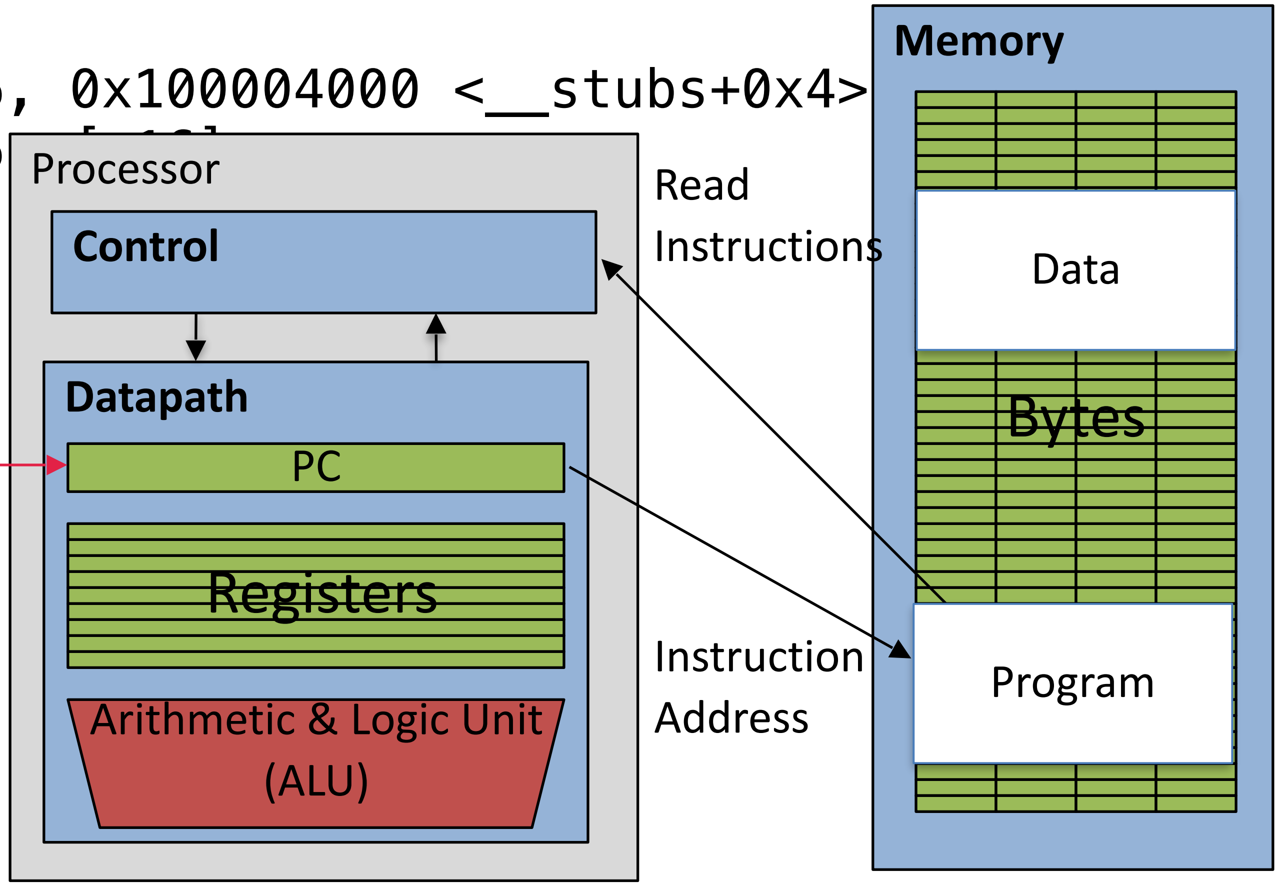
```
0000000100003f40 <_main>:
100003f40: ff c3 00 d1   sub sp, sp, #48
… …
100003f58: 48 9a 80 52   mov w8, #1234
100003f5c: a8 83 1f b8   stur w8, [x29, #-8]
100003f60: 28 1c 82 52   mov w8, #4321
100003f64: a8 43 1f b8   stur w8, [x29, #-12]
100003f68: a8 83 5f b8   ldur w8, [x29, #-8]
100003f6c: a9 43 5f b8   ldur w9, [x29, #-12]
100003f70: 08 01 09 0b   add w8, w8, w9
… …
100003f90: 05 00 00 94  bl  0x100003fa4 <_printf+0x100003fa4>
… …
Disassembly of section __TEXT,__stubs:
0000000100003fa4 <__stubs>:
100003fa4: 10 00 00 b0  adrp x16, 0x100004000 <__stubs+0x4>
100003fa8: 10 02 40 f9  ldr  x16
100003fac: 00 02 1f d6  br x16
```

program counter

Increase by 4 each time an instruction is executed

RISC-V 中 每条指令 4个 byte.

Except for branch/jump/ function call



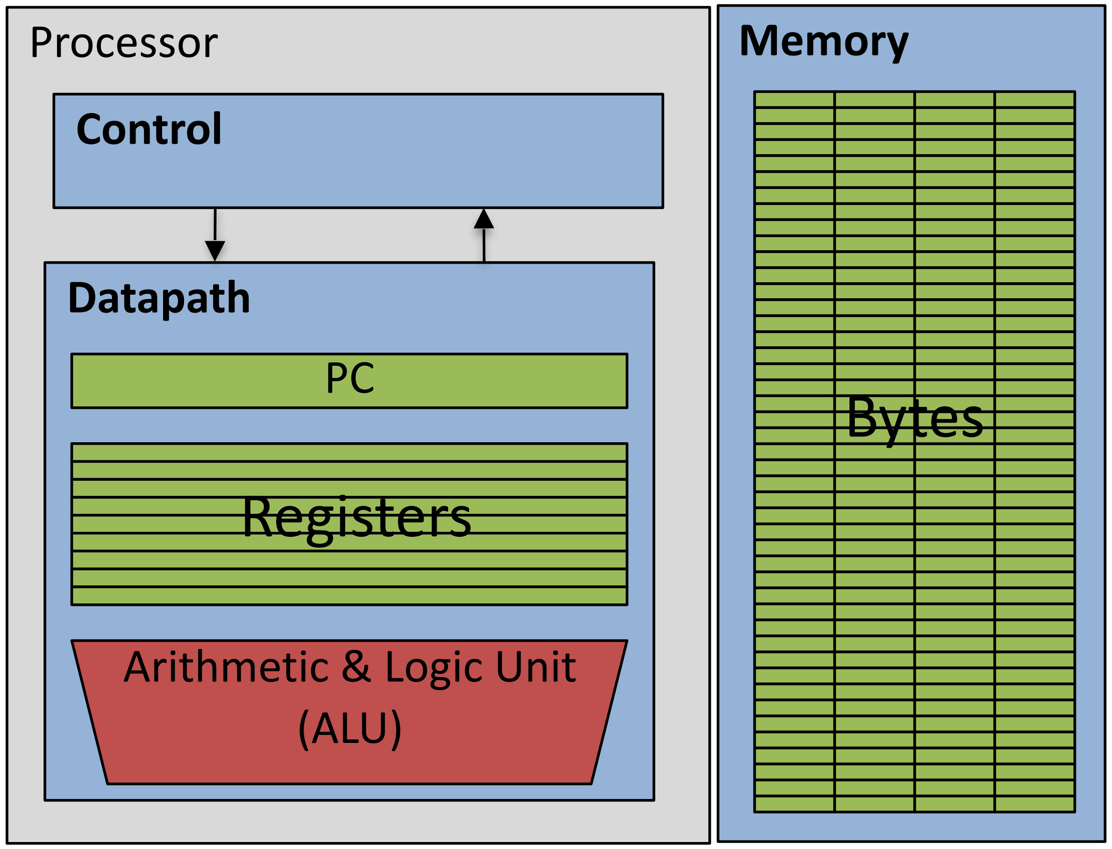
Call a Function

```
#include <stdio.h>
int sum_two_number(int a, int b)
{
    int y;
    return y=a+b;
}
int main(int argc, const char * argv[]) {
    int x=4321, y=1234;
    int a=1,b=2,c=3,d=4,e=5,f=6,g=0;
    y = sum_two_number(x,y);
    c = sum_two_number(a,b);
    f = sum_two_number(e,d);
    g = sum_two_number(c,f);
    printf("Sum is %d.\n",y);
    return 0;
}
```

1. Put parameters in a place where function can access them
2. Transfer control to function (PC jump to sum_two_number)

stack frame

3. Acquire (local) storage resources needed for function
4. Perform desired task of the function

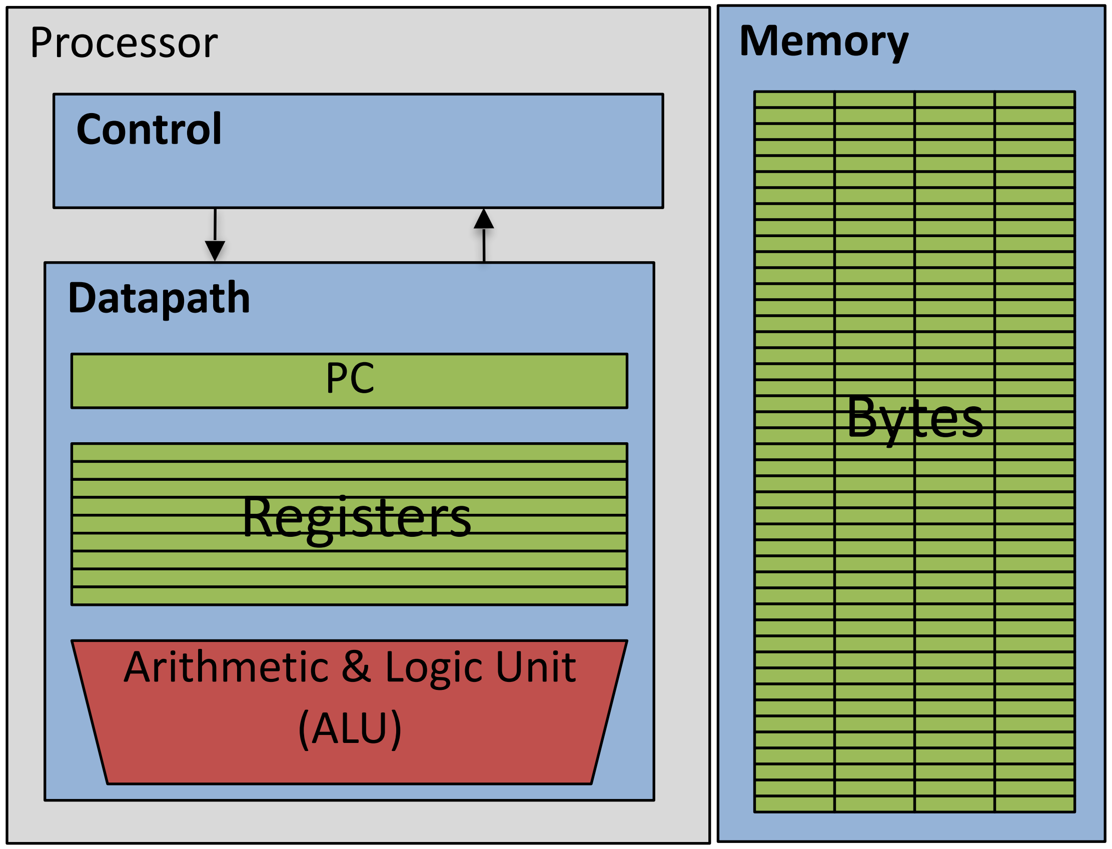


Call a Function

```
#include <stdio.h>
int sum_two_number(int a, int b)
{
    int y;
    return y=a+b;
}
int main(int argc, const char * argv[]) {
    int x=4321, y=1234;
    int a=1,b=2,c=3,d=4,e=5,f=6,g=0;
    y = sum_two_number(x,y);
    c = sum_two_number(a,b);
    f = sum_two_number(e,d);
    g = sum_two_number(c,f);
    printf("Sum is %d.\n",y);
    return 0;
}
```

6. Return control to point of origin, since a function can be called from several points in a program

5. Put result value in a place where calling code can access it and restore any registers you used



Processor

Control

Datapath

PC

Registers

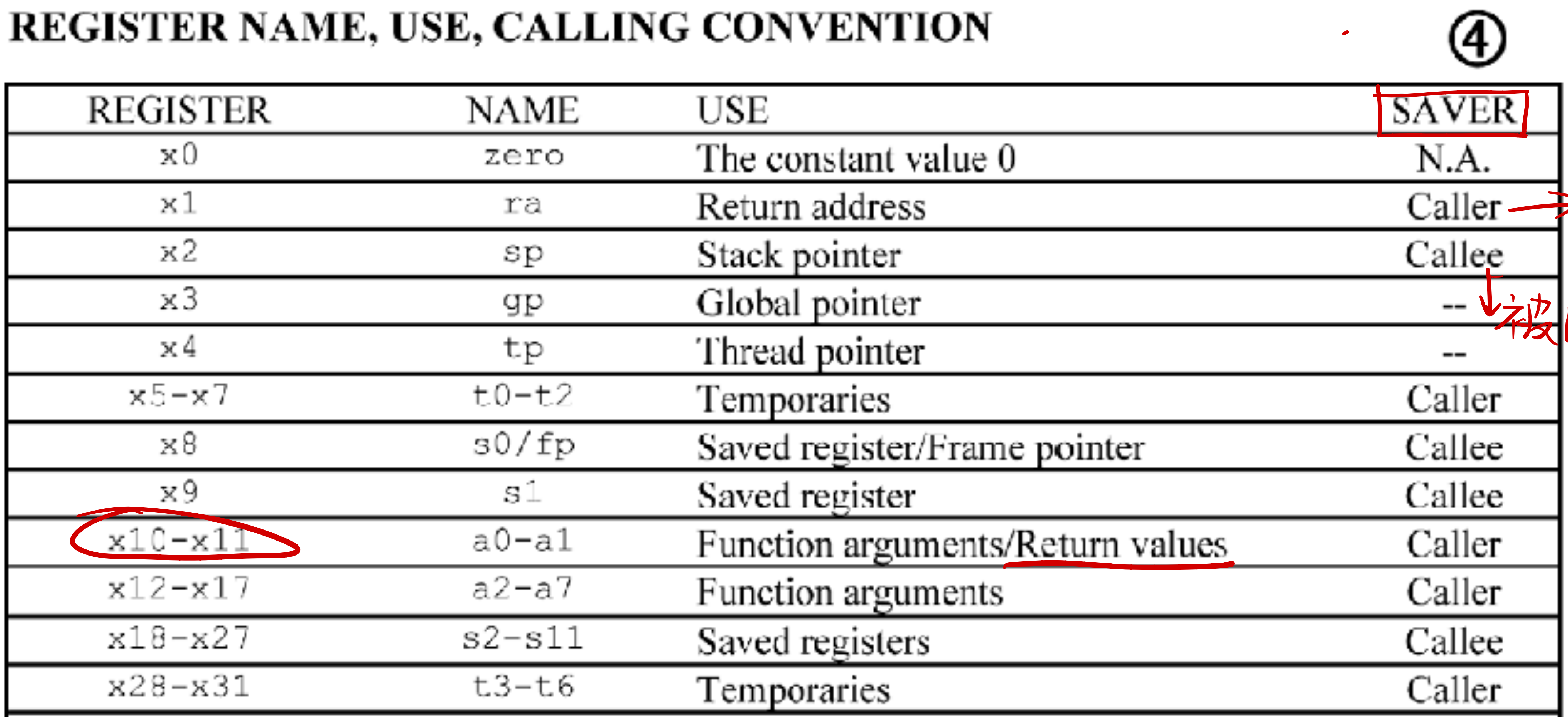
Arithmetic & Logic Unit (ALU)

Memory

Bytes

RISC-V Function Call Conventions

- Registers faster than memory, so use them as much as possible
- Give names to registers, conventions on how to use them



REGISTER NAME, USE, CALLING CONVENTION ④

REGISTER	NAME	USE	SAVER
x0	zero	The constant value 0	N.A.
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	--
x4	tp	Thread pointer	--
x5-x7	t0-t2	Temporaries	Caller
x8	s0/fp	Saved register/Frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Function arguments/Return values	Caller
x12-x17	a2-a7	Function arguments	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller

Caller → 调用别人

Callee ↓ 被调用

Older version: https://riscv.org/wp-content/uploads/2015/01/riscv-calling.pdf

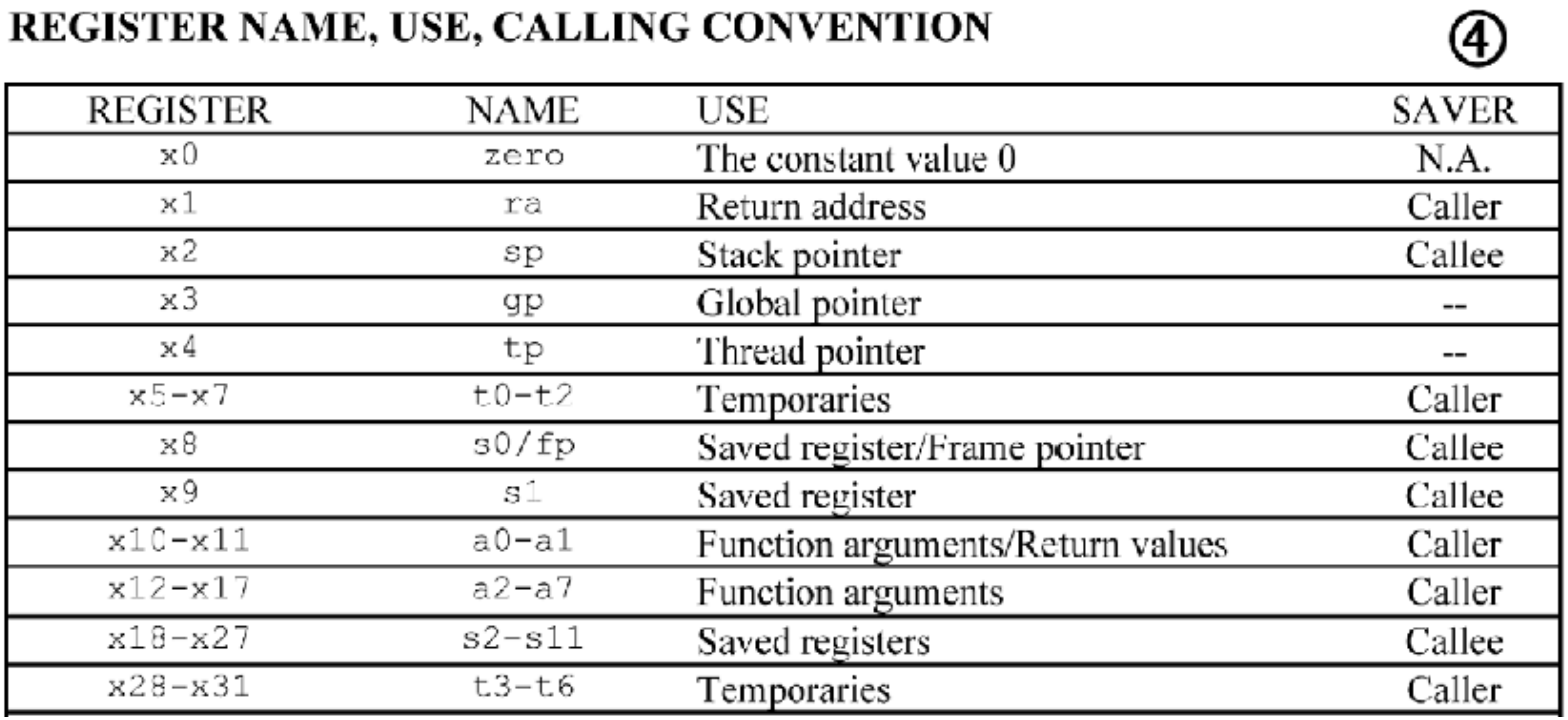
Latest draft: https://github.com/riscv-non-isa/riscv-elf-psabi-doc/releases/tag/draft-20230220-87f4a72d5aeaf048b35a230e0ba5accd1bfcf072

RISC-V Function Call Conventions

- `a0—a7 (x10-x17)`: eight argument registers to pass parameters and return values `(a0-a1)`

return address

- `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)



REGISTER NAME, USE, CALLING CONVENTION ④

REGISTER	NAME	USE	SAVER
x0	zero	The constant value 0	N.A.
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	--
x4	tp	Thread pointer	--
x5-x7	t0-t2	Temporaries	Caller
x8	s0/fp	Saved register/Frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Function arguments/Return values	Caller
x12-x17	a2-a7	Function arguments	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller

Call a Function

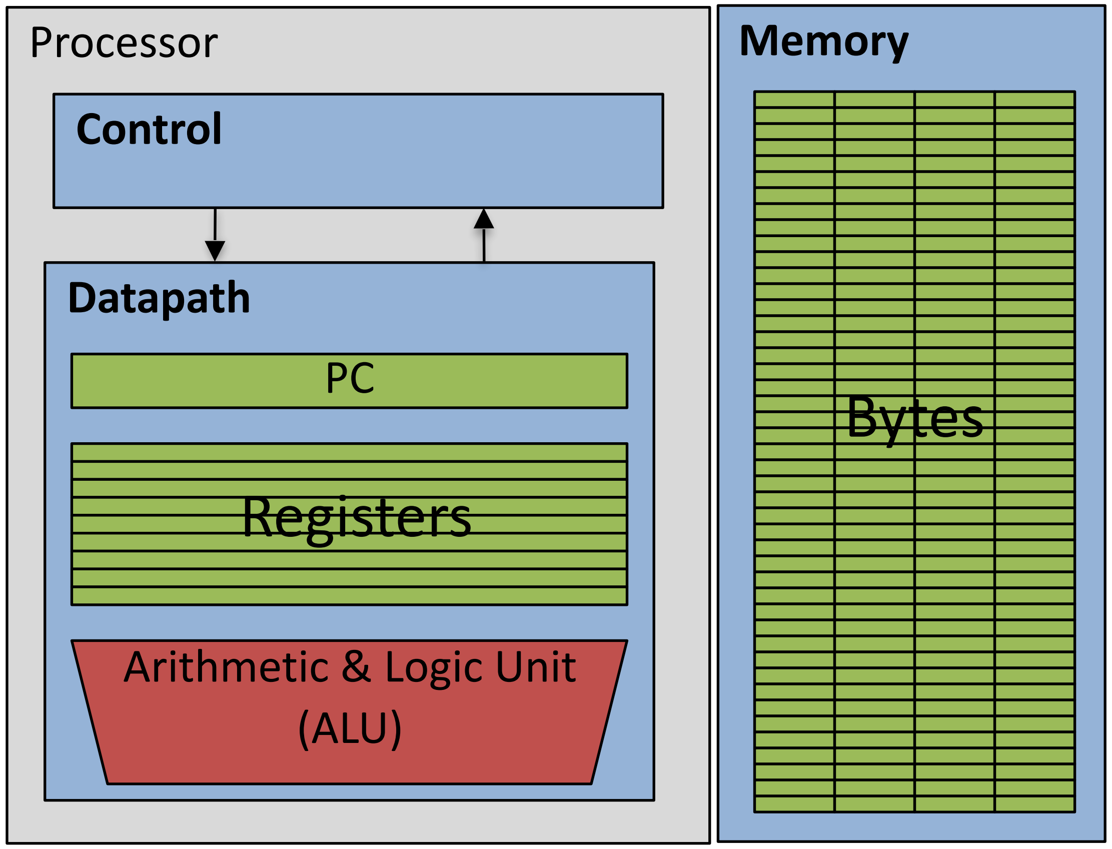
```
#include <stdio.h>
int sum_two_number(int a, int b)
{
    int y;
    return y=a+b;
}
int main(int argc, const char * argv[]) {
    int x=4321, y=1234;
    int a=1,b=2,c=3,d=4,e=5,f=6,g=0;
    y = sum_two_number(x,y);
    c = sum_two_number(a,b);
    f = sum_two_number(e,d);
    g = sum_two_number(c,f);
    printf("Sum is %d.\n",y);
    return 0;
}
```

y is returned function argument;

Can be put in registers `a0-a1`

x and y are function arguments;

Can be put in registers `a0-a7`



Processor

Control

Datapath

PC

Registers

Arithmetic & Logic Unit (ALU)

Memory

Bytes

Call a Function

```
#include <stdio.h>
int sum_two_number(int a, int b)
{    Func_called:
     0x2000  //one instruction
     0x2004  //another instruction
}    ... ...      //need jump back to main()
int main(int argc, const char * argv[]) {
     Start:
     0x1000  //one instruction
     0x1004  //another instruction
     0x1008  //a third instruction
     0x100c  //PC jump to 0x2000 (call function
     sum_two_number)
     0x1010   //next instruction... ...
     ... ...
}
```

PC = 0x100C

PC = 0x1010

Save this value to register ra

Call a Function—Jump

- JAL: Jump & Link, jump to function
- Unconditional jump (J-type)

ra

31	30 21	20	19 12	11 7	6 0
imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode
1	10	1	8	5	7

```
jal rd label
```

返回：jal rd

Jump to label (imm+PC, explain later) and save return address (PC+4) to rd;

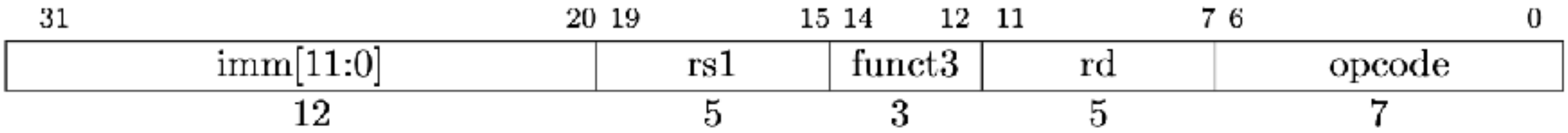
rd is x1 (ra) by convention; sometimes can be x5.

When rd is x0, it is simply unconditional jump (j) without recording PC+4.

任何指令对x0无效

Return—Jump

- JALR: Jump & Link Register
- Unconditional jump (**I-type**)



31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7

```
jalr rd label
```

Jump to label `(imm+rs1)&~1` and save return address `(PC+4)` to `rd`

`rs1` can be the return address we just saved to `ra`

When `rd` is `x0,` it is simply unconditional jump (j) without recording PC+4.

Jump

`–jal rd offset  –jalr rd rs offset`

- Jump and Link
  - Add the immediate value to the current address in the program (the "Program Counter"), go to that location
    - The offset is 20 bits, sign extended and left-shifted one (not two)
  - At the same time, store into rd the value of PC+4
    - So we know where it came from (need to return to)
  - `jal offset == jal x1 offset` (pseudo-instruction; x1 = ra = return address)
  - `j offset == jal x0 offset` (jump is a pseudo-instruction in RISC-V)
- Two uses:
  - Unconditional jumps in loops and the like
  - Calling other functions

Jump and Link Register

- The same except the destination
    - Instead of PC + immediate it is rs + immediate
        - Same immediate format as I-type: 12 bits, sign extended
- 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

Notes on Functions

- Calling program (caller) puts parameters into registers `a0-a7` and uses `jal X` to invoke (callee) at address labeled `X`
- Must have register in computer with address of currently executing instruction
  - Instead of Instruction Address Register (better name), historically called Program Counter (PC)
  - It's a program's counter; it doesn't count programs!
- What value does `jal X` place into `ra`?
- `jr ra` puts address inside `ra` back into PC

= ret

Call a Function

1. Put parameters in a place where function can access them

2. Transfer control to function (PC jump to function)

3. Acquire (local) storage resources needed for function

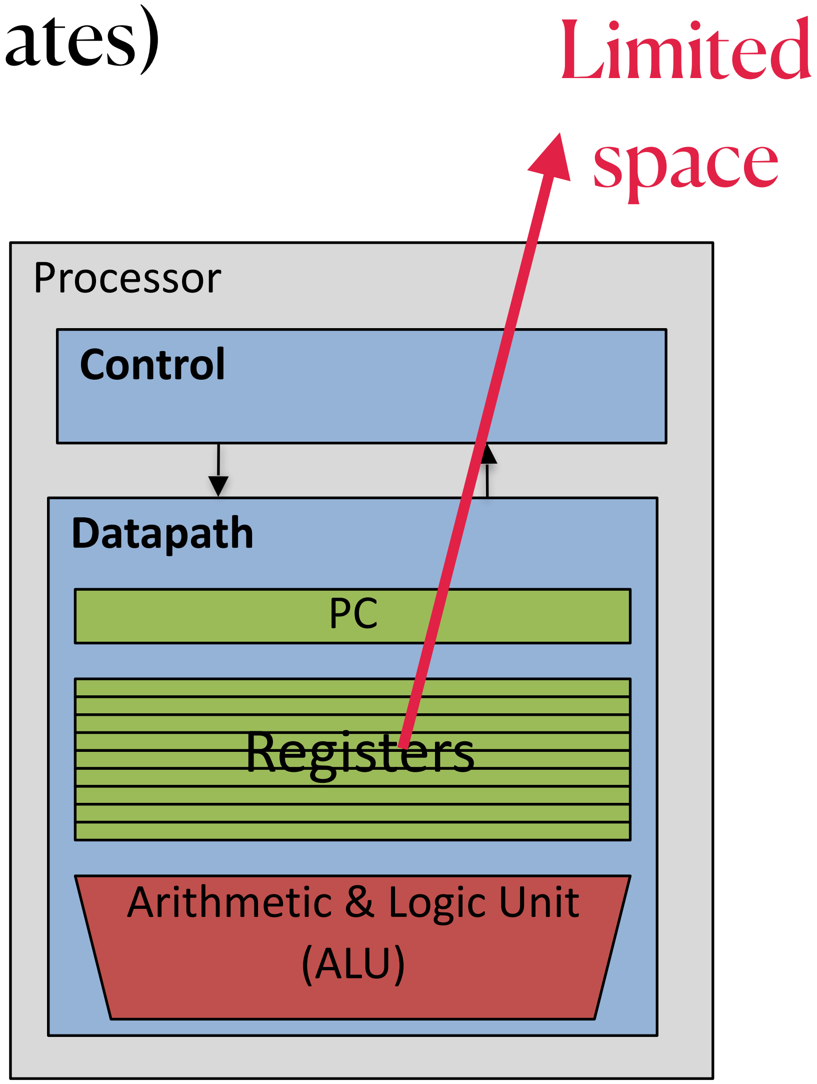
4. Perform desired task of the function

5. Put result value in a place where calling code can access it and restore any registers you used

6. Return control to point of origin, since a function can be called from several points in a program

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

- Need a place to save old values before call function, restore them when return, and delete
- 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`)
- Convention is grow from high to low addresses
  - Push decrements `sp`, Pop increments `sp`



Stack

- Stack frame may include:
    - Return "instruction" address
    - Parameters (spill)
    - Space for other local variables
- Stack frames contiguous; stack pointer (`sp/x2`) tells where bottom of stack frame is
- When procedure ends, stack frame is tossed off the stack; frees memory for future stack frames; `sp` restores

Example

- Leaf function: a function that calls no function

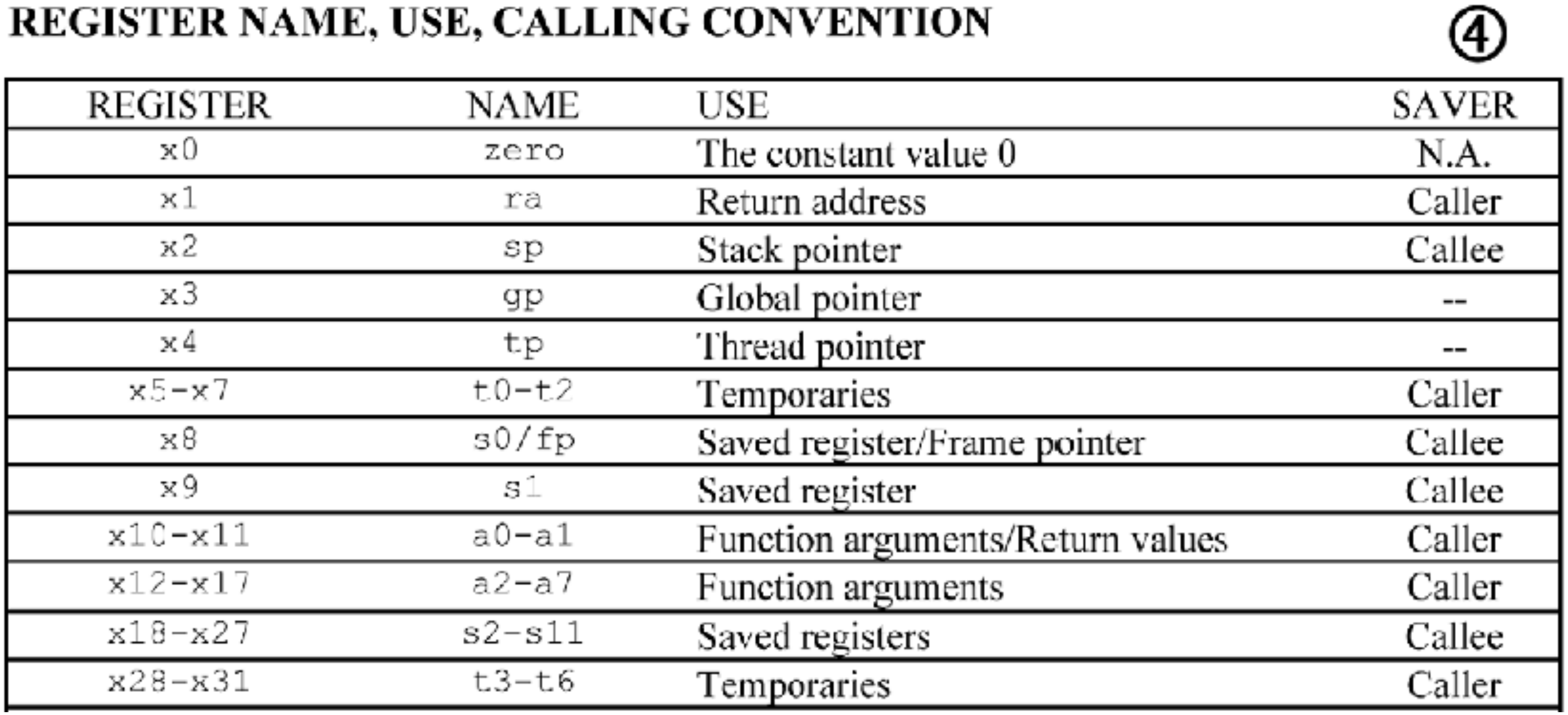
```
int Leaf (int g, int h, int i, int j)
{
   int f; f = (g + h) – (i + j);
   return f;
}
int main (void){
     int a=1, b=2, c=3, d=4, e;
     e = Leaf(a,b,d,c);
     return e;
} /*a function called by OS*/
```

0	x0/zero
ra	x1
sp	x2
… …	
s1	x9
a0	x10
a1	x11
a2	x12
a3	x13
a4	x14
… …	

- Parameter variables g, h, i, and j in argument registers `a1`, `a2`, `a3`, and `a4`, and f in `a0` when returned, and assume e in `s1`, later should be copied to `a0` when returned
- Register `ra` consideration

Stack Before, During, After Function

- Caller needs to save old values of `a1`, `a2`, `a3` and `a4`
- Callee needs to save old value of `s1` (and any other callee saved registers), and makes sure they are not changed after return

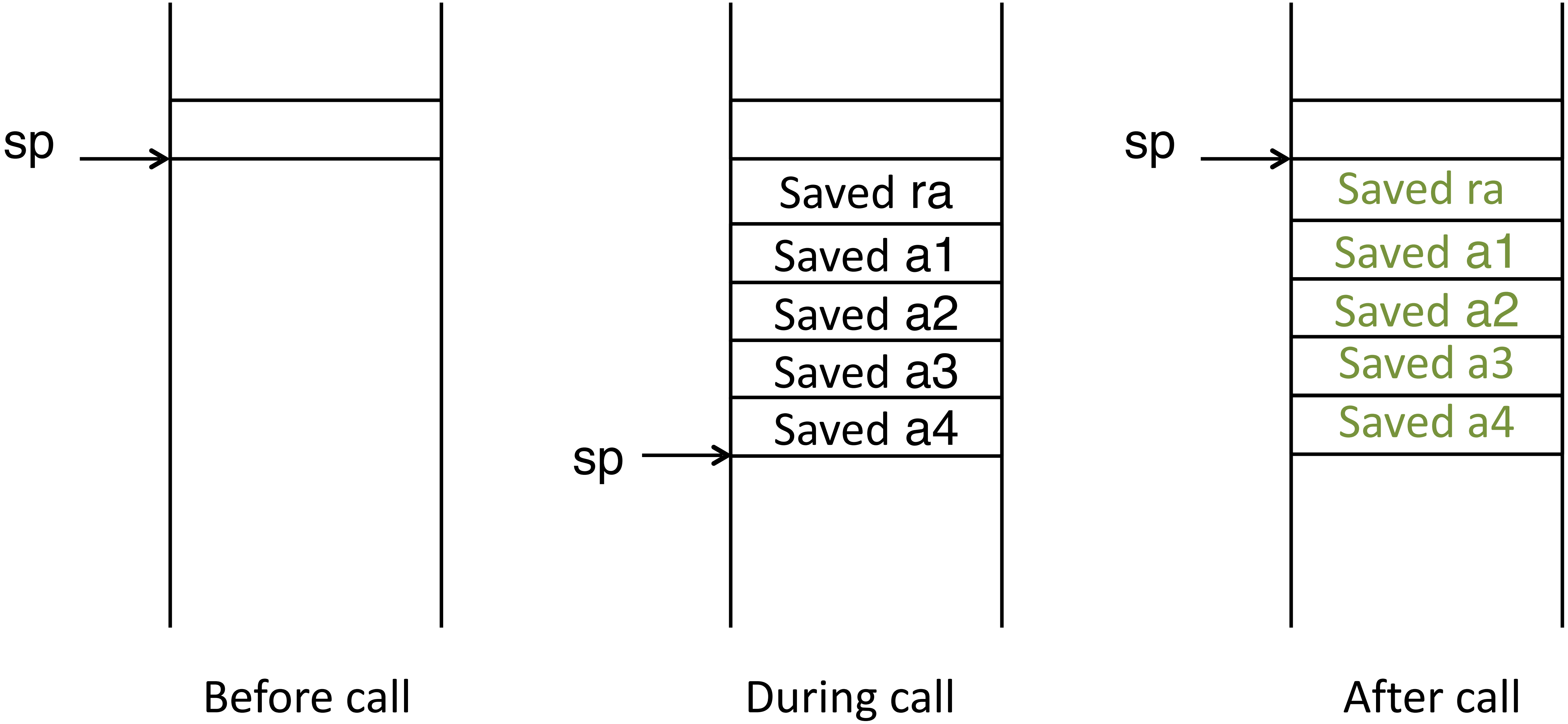


REGISTER NAME, USE, CALLING CONVENTION ④

REGISTER	NAME	USE	SAVER
x0	zero	The constant value 0	N.A.
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	--
x4	tp	Thread pointer	--
x5-x7	t0-t2	Temporaries	Caller
x8	s0/fp	Saved register/Frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Function arguments/Return values	Caller
x12-x17	a2-a7	Function arguments	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller

Stack Before, During, After Function

- Need to save old values of ra, a1, a2, a3 and a4 (caller-saved)
- W.r.t. main()

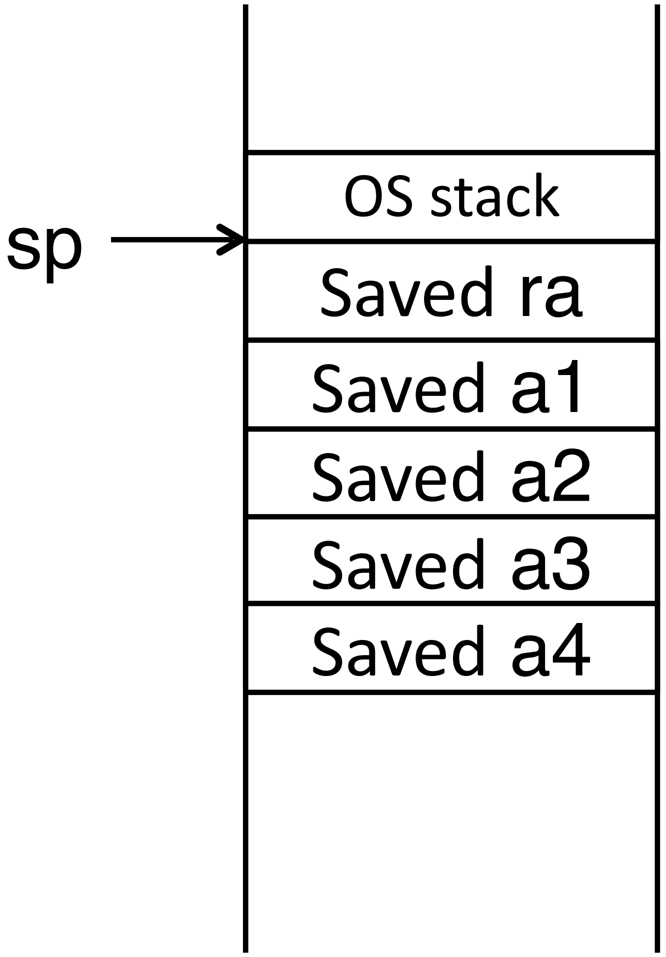


RISC-V Code for Main()/Leaf()

```
Main:
     addi     sp, sp, -20   # adjust stack for 5 items, 4 int & 1 ra pointer/address
    sw   ra, 16(sp)     # save ra for use afterwards (return to OS)
    sw   a1, 12(sp)     # save a1 for use afterwards, these are all caller-saved
    sw   a2, 8(sp)      # save a2 for use afterwards
    sw   a3, 4(sp)      # save a3 for use afterwards
    sw   a4, 0(sp)      # save a4 for use afterwards

    jal  ra, Leaf       # save a1 for use afterwards

    lw   a1, 12(sp)     # restore register a1
    lw   a2, 8(sp)      # restore register a2
    lw   a3, 8(sp)      # restore register a2
    lw   a4, 8(sp)      # restore register a2
    lw   ra, 8(sp)      # restore register ra
    addi     sp, sp, 20    # adjust stack to delete 5 items
    mv   a0, a0         # move result to return register
    jr   ra             # return
```

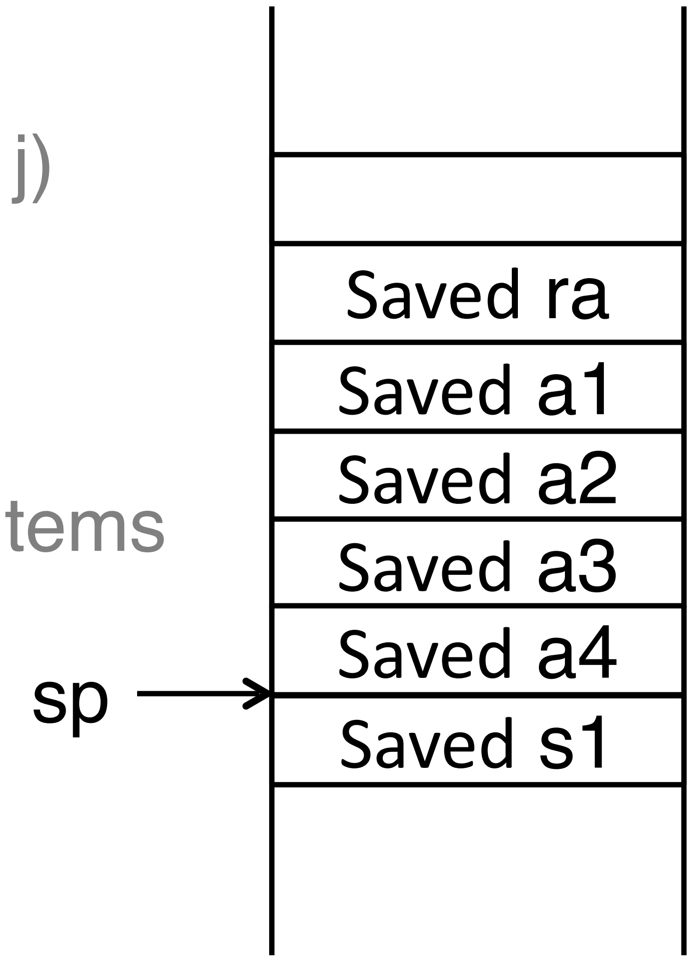


Stack
During call

RISC-V Code for Main()/Leaf()

```
Leaf:
     addi sp, sp, -4      # adjust stack for 1 items, callee saved s1
      sw   s1, 0(sp)       # save callee saved s1 to stack
      add a1, a0, a1      # g = g + h
      add a2, a2, a3      # j = i + j
      sub  s1, a0, a2     # calculate result (g + h) – (i + j)
      mv   a0, s1        # return value (g + h) – (i + j)

      lw    s1, 0(sp)     # restore register s1 for caller
      addi       sp, sp, 4       # adjust stack to delete 1 items
      jr     ra                   # jump back to caller
                                    (pseudo-assembly: ret)
```



sp

Saved ra

Saved a1

Saved a2

Saved a3

Saved a4

Saved s1

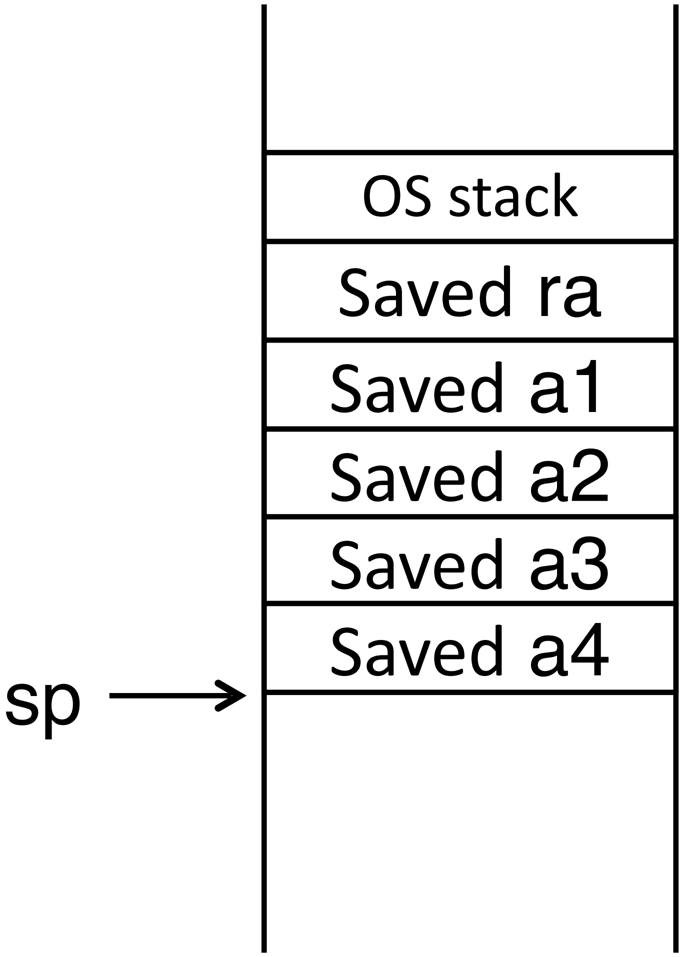
Stack
During call

RISC-V Code for Main()/Leaf()

```
Main:
     addi    sp, sp, -20   # adjust stack for 5 items, 4 int & 1 ra pointer/address
    sw   ra, 16(sp)     # save ra for use afterwards (return to OS)
    sw   a1, 12(sp)     # save a1 for use afterwards, these are all caller-saved
    sw   a2, 8(sp)      # save a2 for use afterwards
    sw   a3, 4(sp)      # save a3 for use afterwards
    sw   a4, 0(sp)      # save a4 for use afterwards

    jal  ra, Leaf       # save a1 for use afterwards

    lw   a1, 12(sp)     # restore register a1
    lw   a2, 8(sp)      # restore register a2
    lw   a3, 8(sp)      # restore register a2
    lw   a4, 8(sp)      # restore register a2
    lw   ra, 8(sp)      # restore register ra
    addi    sp, sp, 20    # adjust stack to delete 5 items
    mv   a0, a0         # move result to return register
    jr   ra             # return
```



Stack
During call

Call a Function

1. Caller put parameters in a place where function can access them (a0-a7, or stack when registers not avail.), and then save caller-saved registers to stack
2. Transfer control to function (PC jump to function): JAL, ra is changed to where caller left
3. Acquire (local) storage resources needed for function: change sp (size decided when compiling);

   Push callee-saved registers to stack (e.g., s0-s11)
4. Perform desired task of the function
5. Put result value in a place where calling code can access it (a0, a1), and restore callee-saved registers (s0-s11, sp)
6. Return control to point of origin, since a function can be called from several points in a program (jr); caller restores caller-saved registers