CS 110
Computer Architecture
RISC-V II

Instructors:
Siting Liu & Chundong Wang
Course website: https://toast-lab.sist.shanghaitech.edu.cn/courses/CS110@ShanghaiTech/Spring-2024/index.html
School of Information Science and Technology (SIST)
ShanghaiTech University

2024/3/12

Clarifications

- R-type compare and how to identify a negative number

RV32I R-type Compare

- Syntax of instructions
  - SLT/SLTU: `slt/sltu rd, rs1, rs2`

  Compare the value stored in register `rs1` and that of `rs2`, sets `rd=1`, if `rs1<rs2` otherwise `rd=0`, equivalent to a = b < c ? 1 : 0, a ⇔ `rd`, b ⇔ `rs1`, c ⇔ `rs2`. Treat the numbers as signed/unsigned with `slt/sltu`.

  - Example:

```
slt  x5, x2, x1
slt  x4, x3, x1
sltu x5, x3, x1
```

0

x2<x1 → x5=1 → Signed.

x3<x1 → x4=1

x3>x1 → x5=0 → unsigned

  - Overflow detection (unsigned)

```
add x5, x3, x3
sltu x6, x5, x3
```

  - Overflow detection (signed)?

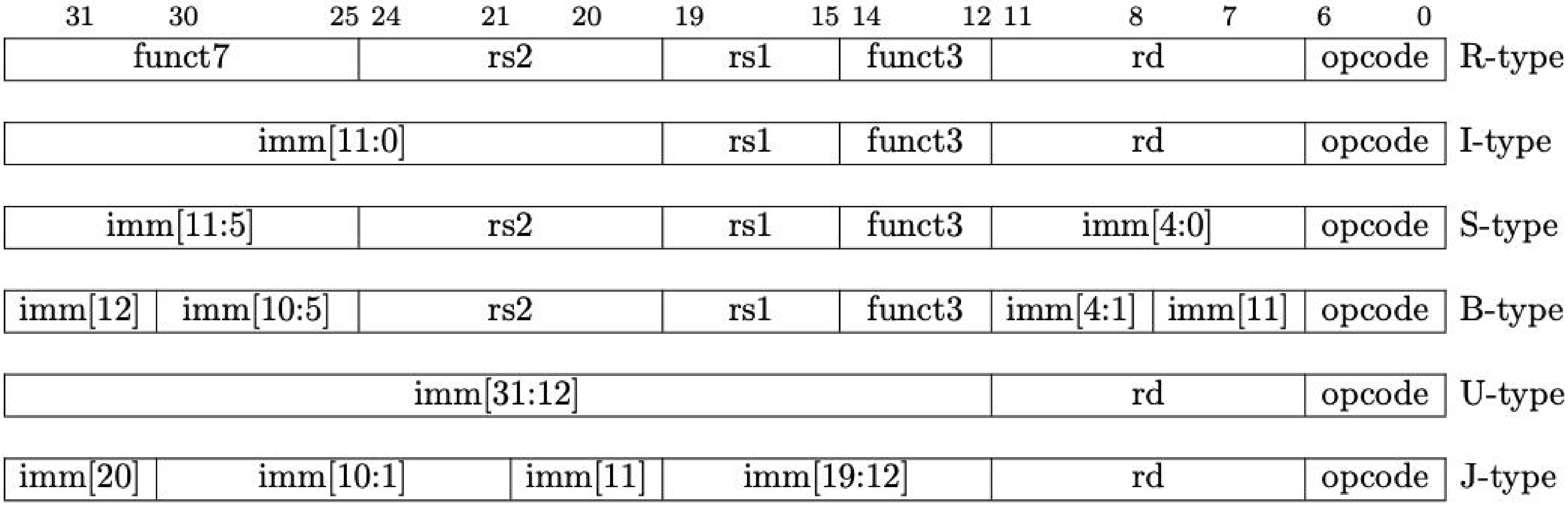
```
add t0, t1, t2
slti t3, t2, 0
slt t4, t0, t1
bne t3, t4, overflow
```

Registers

0	x0/zero
0x12340000	x1
0x00006789	x2
0xFFFFFFFF	x3
0x1	x4
0x 0 00... 1	x5 or 0x1 → 0x0
	x6
	x7

Clarifications

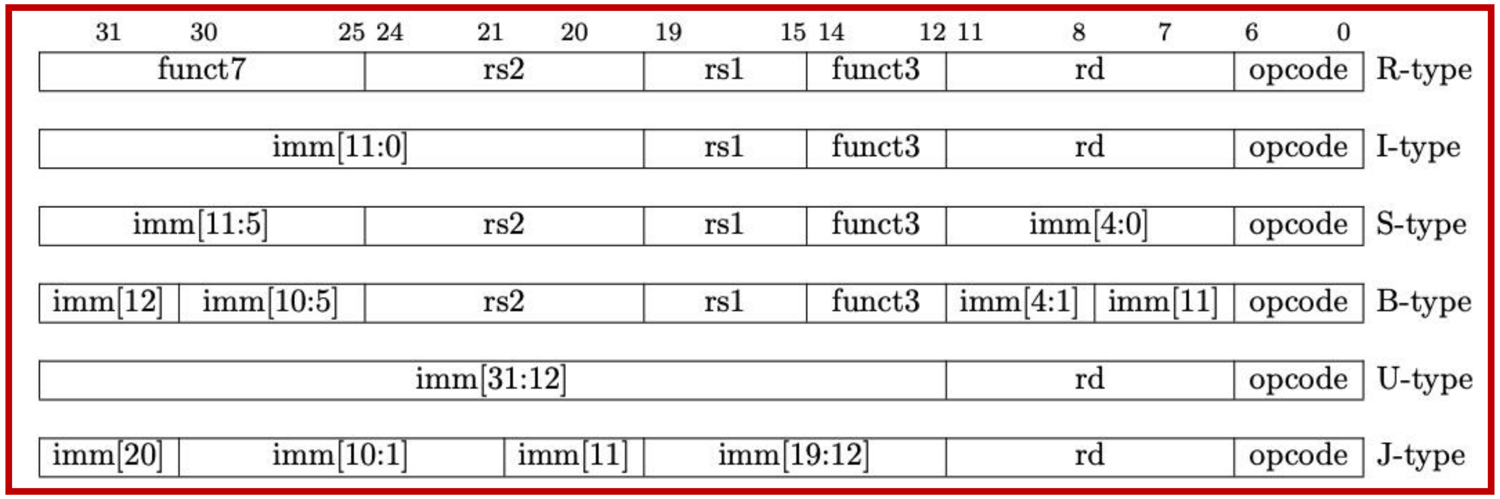
- R-type compare and how to identify a negative number
- `slt` has unsigned version as `sltu`, but not the other R-type instructions
- `rs1`, `rs2` and `rd` field in the code all have 5 bits to represent the 32 general purpose registers (GPRs)



31	30 25	24 21	20	19 15	14 12	11 8	7	6 0	
funct7		rs2		rs1	funct3	rd		opcode	R-type
imm[11:0]				rs1	funct3	rd		opcode	I-type
imm[11:5]		rs2		rs1	funct3	imm[4:0]		opcode	S-type
imm[12]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode	B-type
imm[31:12]						rd		opcode	U-type
imm[20]	imm[10:1]		imm[11]	imm[19:12]		rd		opcode	J-type

Clarifications

- R-type compare and how to identify a negative number
- `slt` has unsigned version as `sltu`, but not the other R-type instructions
- `rs1`, `rs2` and `rd` field in the code all have 5 bits to represent the 32 general purpose registers (GPRs)

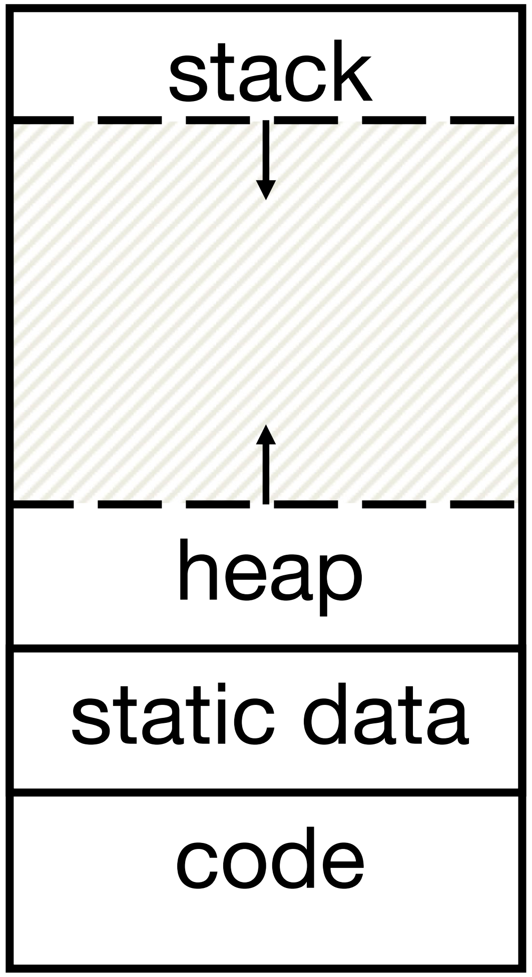


31	30 25	24 21	20	19 15	14 12	11 8	7	6 0	
funct7		rs2		rs1	funct3	rd		opcode	R-type
imm[11:0]				rs1	funct3	rd		opcode	I-type
imm[11:5]		rs2		rs1	funct3	imm[4:0]		opcode	S-type
imm[12]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode	B-type
imm[31:12]						rd		opcode	U-type
imm[20]	imm[10:1]		imm[11]	imm[19:12]		rd		opcode	J-type

- Your code text vs. the machine code

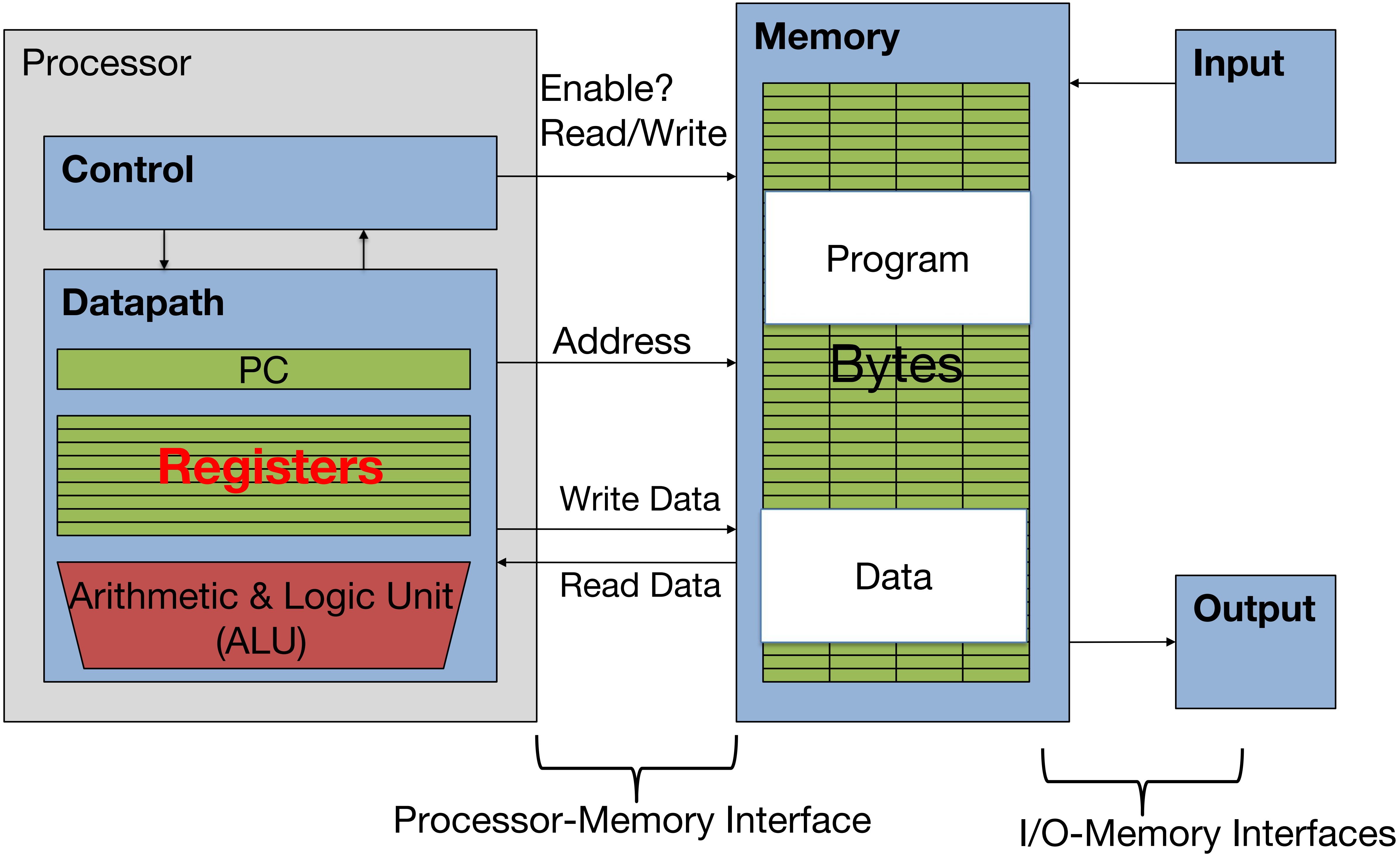
```
1c: a8 83 1f b8   stur w8, [x29, #-8]
20: 28 1c 82 52   mov w8, #4321
24: a8 43 1f b8   stur w8, [x29, #-12]
28: a8 83 5f b8   ldur w8, [x29, #-8]
2c: a9 43 5f b8   ldur w9, [x29, #-12]
30: 08 01 09 0b   add w8, w8, w9
```

Memory Address
(32 bits assumed here)



Clarifications

- This is only a simplified model!



Administrative

- HW2, due Mar. 22nd and Lab2 (with guidance to valgrind) are released
- Lab 3 with tutorial to Venus available today, play with venus to understand better about RISC-V assembly

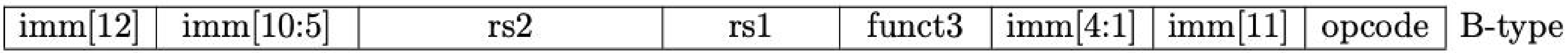
  https://venus.cs61c.org/
- Proj1.1 will be released Next week
- Discussion this week on memory management & valgrind (useful for your labs/HWs/projects) by TA Suting Chen at teaching center 301 on Monday
  - Discussion on Mar. 15th will be held at SPST 4-122 (only this one)
- Discussion next week on RISC-V assembly & CALL, useful for Proj1.1 by TA Chao Yang at teaching center 301

Outline

- Assembly instructions in RISC-V (RV32I)
  - R-type
  - I-type arithmetic and logic
  - I-type load
  - S-type store
  - Desicion-making instructions
    - Branch (B-type)
    - Function call
      - Unconditional jump (J-type)
      - Calling convention
      - Managing the stack

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-type)



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

- RISC-V: similar to if-statement instruction

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

meaning: 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)
```

```
1c: a8 83 1f b8
20: 28 1c 82 52
24: a8 43 1f b8
28: a8 83 5f b8
2c: a9 43 5f b8
30: 08 01 09 0b
```

- C code

Compile

- Assembly

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

```
   addi x2, x0, 5
   addi x3, x0, 6
   bne  x2, x3, L1
   beq  x2, x3, L2
L1:addi x2, x2, 1
   ret #kind of jump, psuedo-instruction
L2:addi x2, x2, -1
   ret
```

assembler

#imm = 8

- 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

- Example:

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

- C code

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

Five variables `f` through `j` correspond to the five registers `x19` through `x23`

- Assembly

```
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
#Jump to Exit

Else: sub x19, x20, x21
Exit: #Else branch & Exit
```

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

- RISC-V: if-statement instructions are

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

meaning: go to statement labeled L if (value in `rs1`) </≥ (value in `rs2`) using singed/unsigned comparison; otherwise, go to the 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
  add x9, x8, x0  # x9=&A[0]
  add x10, x0, x0 # sum=0
  add x11, x0, x0 # i=0
  addi x13,x0, 20 # x13=20
Loop:
  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:
  ret
```

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

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

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

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

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

These code sequences have the same effect!

Translate Assembly to C

```
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;}
```

Outline

- Assembly instructions in RISC-V (RV32I)
  - R-type
  - I-type arithmetic and logic
  - I-type load
  - S-type store
  - Desicion-making instructions
    - Branch (B-type)
    - Function call
      - Unconditional jump (J-type)
      - Calling convention
      - Managing the stack

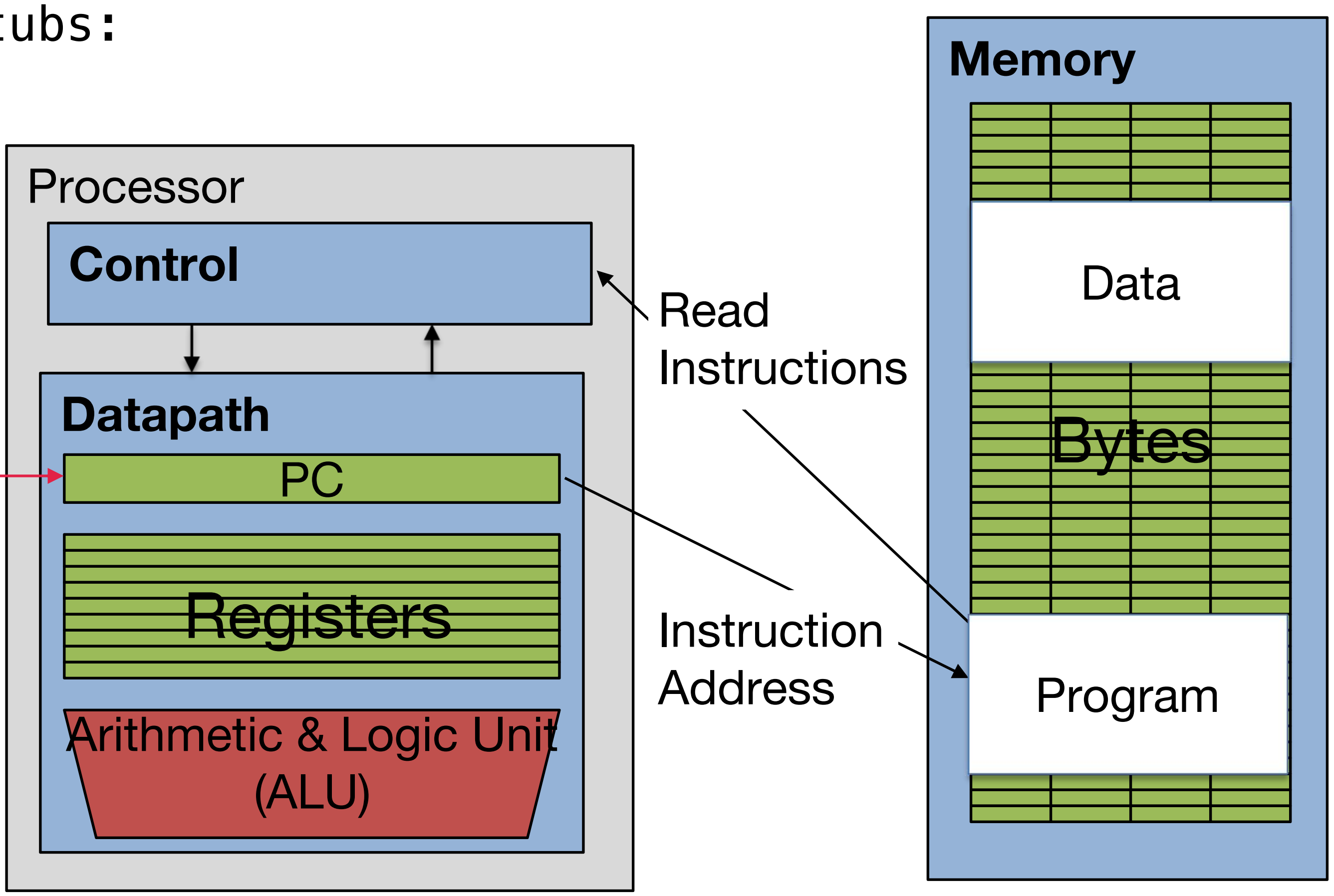
Call a Function — Unconditional Jump

```
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
100003fa8: 10 02 40 f9  ldr
100003fac: 00 02 1f d6  br
```

```
#include <stdio.h>
int main() {//compute 1234 + 4321
  int x = 1234, y = 4321;
  int z = x+y;
  printf("z=%d/n",z);
  return 0;
}
```

Increase by 4 each time an instruction is executed

Except for branch/jump/function call



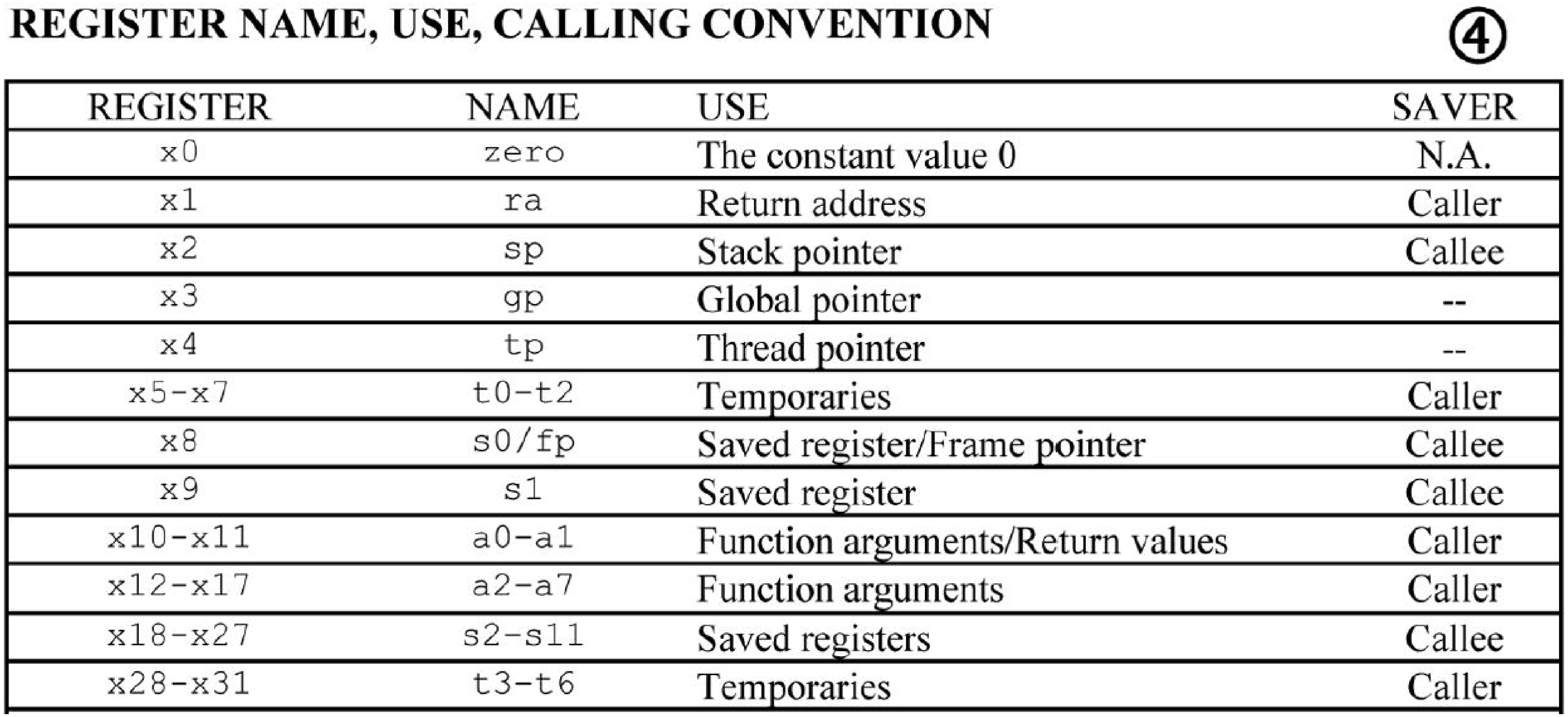
Call a Function

```c
#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)
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

RISC-V Function Call Conventions

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



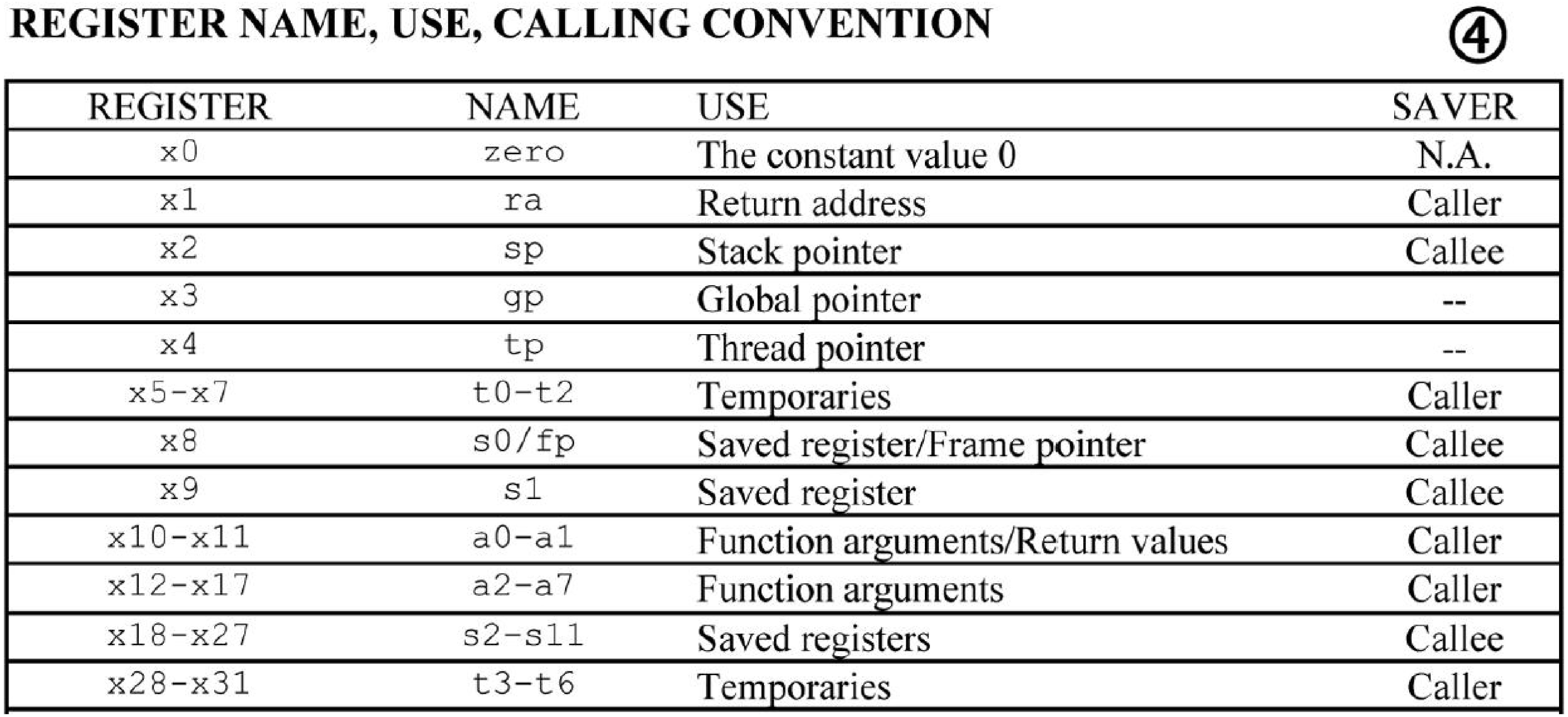
REGISTER NAME, USE, CALLING CONVENTION ④

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

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
Part of application binary interface (ABI)

RISC-V Function Call Conventions

- `a0–a7 (x10–x17)`: eight argument registers to pass parameters and return values (`a0–a1`)
- `ra`: one return address register to return to the point of origin (`x1`)
- Also `s0–s1 (x8–x9)` and `s2–s11` (`x18–x27`): saved registers



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

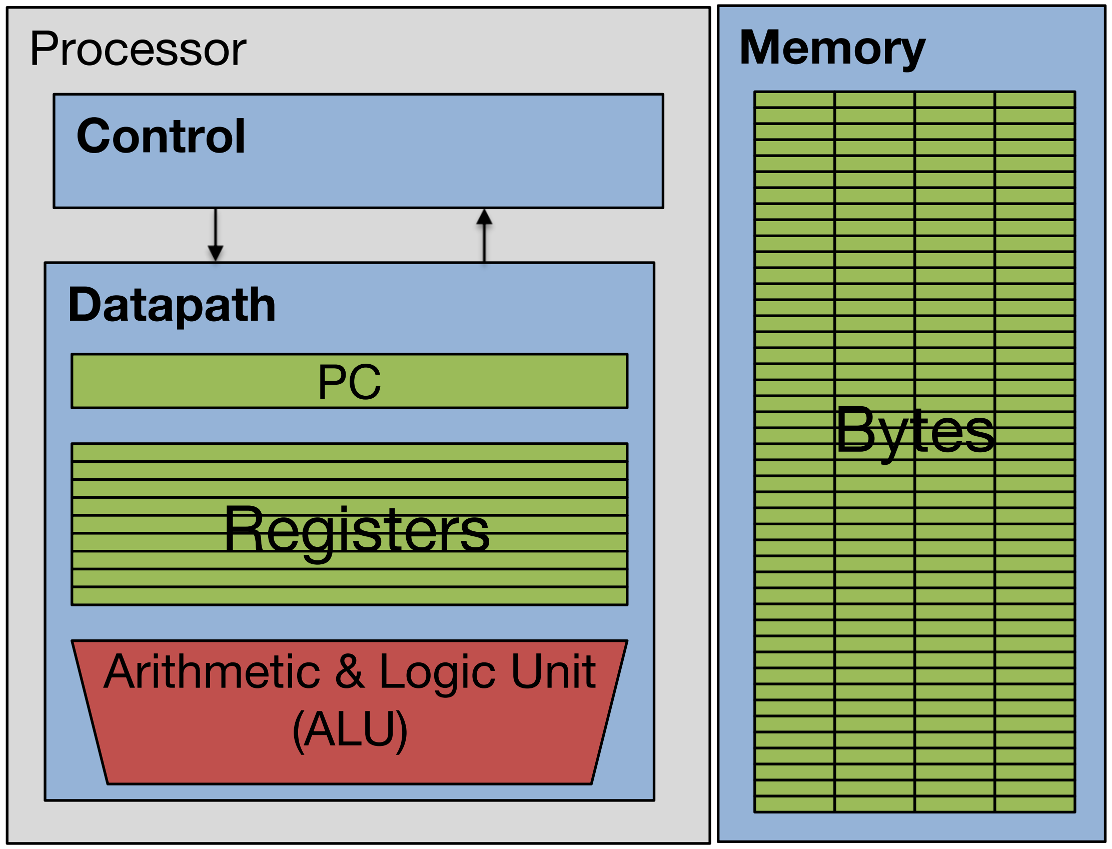
Call a Function

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

```
#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`



Call a Function

- ra: one return address register to return to the point of origin (x1)

```
#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… …
    … …

}
```

Save this value to register ra

Call a Function—J-type

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

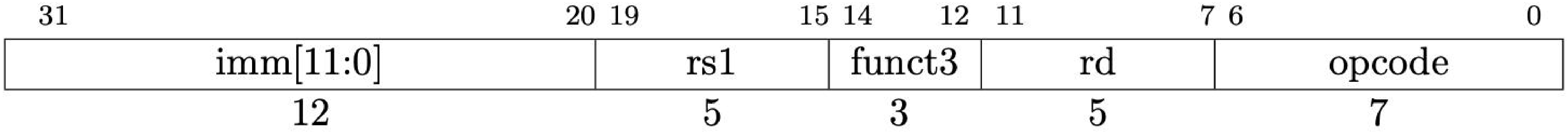
31	30 21	20	19 12	11 7	6 0
imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode
1	10	1	8	5	7

```
jal rd label
```

- 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.

Return—J-type

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



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 (`jal`)
  - 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 [value in `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

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;
}
```

```
int (*p)(void);
```

1. Put parameters in a place where function can access them (`addi/add` etc. to copy)
2. Transfer control to function (PC jump to sum_two_number, `jal/jalr`)
3. Acquire (local) storage resources needed for function
4. Perform desired task of the function (we have learned this)
5. Put result value in a place where calling code can access it and restore any registers you used (`addi/add` etc. to copy)
6. Return control to point of origin, since a function can be called from several points in a program (`jalr`)

Notes on Functions

- Calling program (caller) puts parameters into registers a0–a7 and uses jal X to invoke (callee) at address labeled X
- PC provides the returning point
- What value does jal X place into ra?
- jr ra puts address inside ra back into PC
- New problem, small # of GPRs

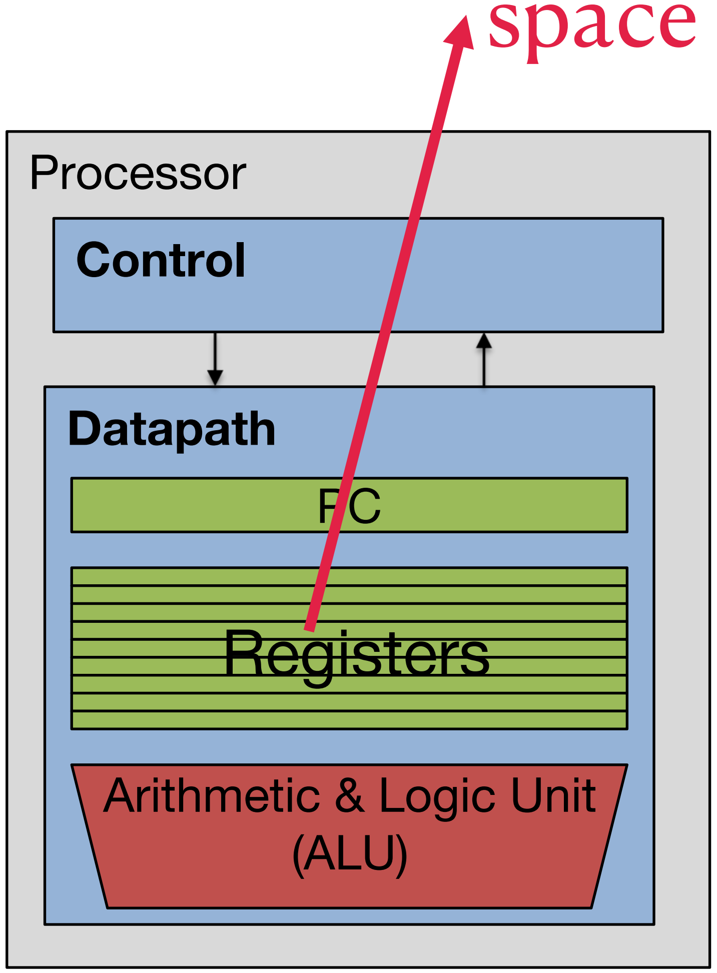
```
#include <stdio.h>
int sum_two_number(int a, int b)
{
    int y;
    return y=a+b;
}
int main(int argc, const char * argv[])
{
    int a=1,b=2,c=3,d=4,e=5,f=6,g=0;
    int h,i,j,k,l,m,n,cs110,cs110p,...;
    y = sum_two_number(x,y);
    ...
```

3. Acquire (local) storage resources needed for function

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; use the big main memory
- 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

Limited space

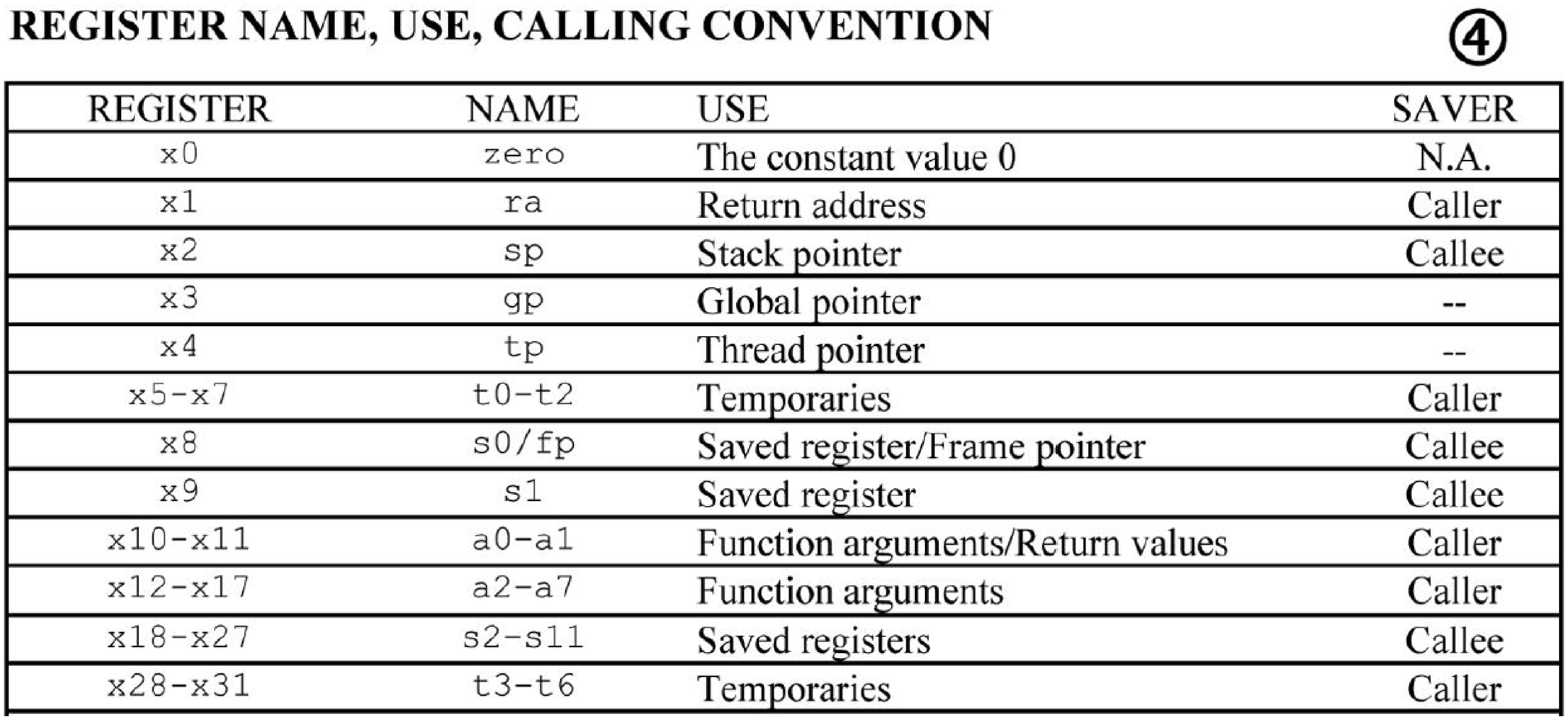


Stack

- 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

Calling Convention

- Could have saved all variables to main memory
- Calling convention to make things easier
- Callee-saved registers: callee clean the mess
  - Callee needs to save old value of `s1` (and any other callee saved registers), and makes sure they are not changed after return, and



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

Example

- Leaf function: a function that calls no function

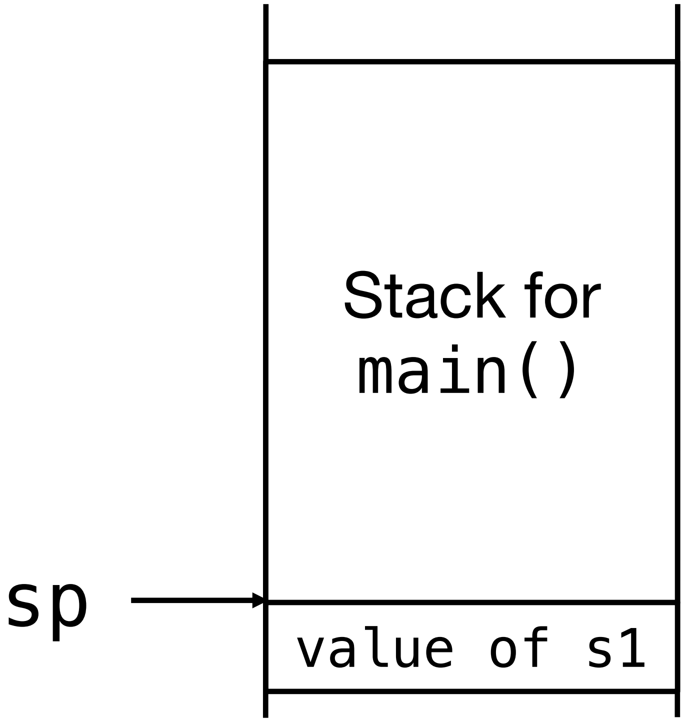
```
int Leaf (int g, int h, int i, int j)
{
   int f; f = (g + h) – (i + j);
   return f;
}
int main (void){
     int a=1, b=2, c=3, d=4, e, x;
     e = Leaf(a,b,d,c);
     ... ...
     return e;
} /*a function called by OS*/
```

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 x in s1
- Assume function Leaf use s1 for intermediate results
- Register ra consideration

RISC-V Code for Leaf()

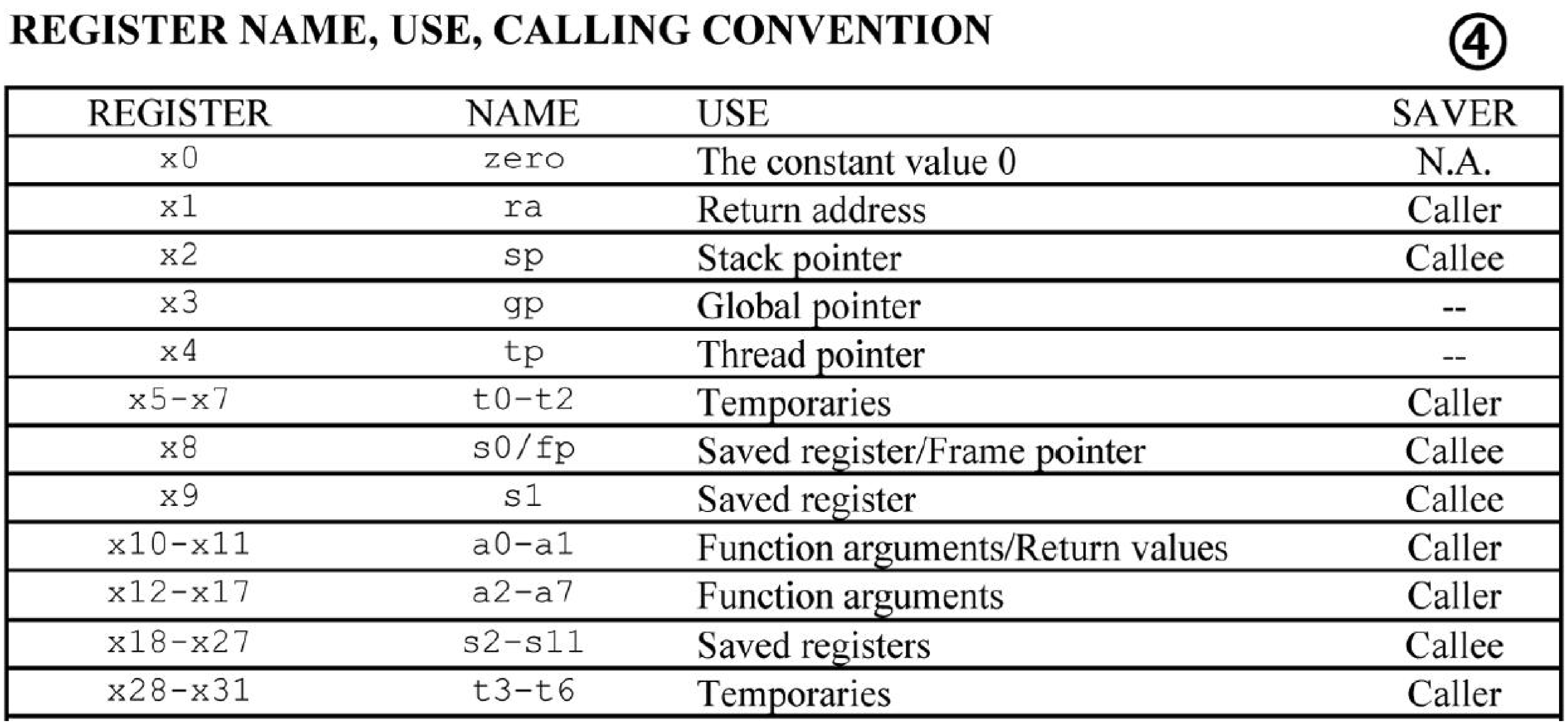
```
Leaf:
addi   sp, sp, -4 # adjust stack for 1 items, callee
saved s1
sw    s1, 0(sp)  # save callee saved s1 to stack
add    s1, a0, a1 # s1 = g + h
add  a2, a2, a3 # j = i + j
sub  a0, s1, a2 # calculate result (g + h) − (i + j)
              # return value (g + h) − (i + j)
lw   s1, 0(sp) # restore register s1 for caller
addi   sp, sp, 4 # adjust stack to delete 1 items
jr   ra    # jump back to caller (pseudo-assembly: ret)
```



Stack
During call

Optimization for `Leaf()`

- Caller-saved registers: caller clean the mess
  - Caller needs to save old value of caller-saved registers, and makes sure they are not changed for further use



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

Optimization for Leaf()

- Caller-saved registers: caller clean the mess
  - Caller needs to save old value of caller-saved registers, and makes sure they are not changed for further use

```
Leaf:
addi   sp, sp, -4 # adjust stack for 1 items, callee
saved s1
~~sw    s1, 0(sp)  # save callee saved s1 to stack~~
add   t1, a0, a1 # s1 = g + h
add  a2, a2, a3 # j = i + j
sub  a0, t1, a2 # calculate result (g + h) – (i + j)
              # 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)
```

Nested Call

```
int bar(int g,int h,int i,int j)
{
 int f = (g + h) - (i + j);
 return f;
}
int foo(int x)
{
//do stuff
 int x = bar(g, h, i, j);
//g, h, i, j for other use
return (x/2);
}
int main()
{
// do stuff
foo(x);
// do stuff
}
```

In **foo**, **g**, **h**, **i**, and **j** are in **s0–s3**

```
foo:
# do stuff (code omitted)
# save ra
```

(Prologue)

```
addi sp, sp, -4
sw ra, 0(sp)
# set up argument registers
add a0, s0, x0
add a1, s1, x0
add a2, s2, x0
add a3, s3, x0
```

```
jal bar
```

(Epilogue)

```
# restore ra
lw ra, 0(sp)
addi sp, sp, 4
slli a0, a0, 1
jr ra
```

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 callee function (`PC` jump to function): `jal/jalr, ra` is changed to where caller left
3. Acquire (local) storage resources needed for function: change `sp` (size decided when compiling);

   Callee 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 ra`); caller restores caller-saved registers