CSC 3210 Computer organization and programming

Chunlan Gao

Georgia State University

Attendance Number

2

Last Class: Logical Operation and conditional operation

- AND
- OR
- XOR
- slli

Conditional operation : if else/ while loop

- bqe
- bne

More Conditional Operations

- `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
  - if (a > b)： a += 1;
  - a in x22, b in x23

```
    bge  x23, x22, Exit      // branch if b >= a
    addi x22, x22, 1
Exit:
```

Signed vs. Unsigned

- Signed comparison: blt, bge
- Unsigned comparison: bltu, bgeu
- Example
  - x22 = 1111 1111 1111 1111 1111 1111 1111 1111
  - x23 = 0000 0000 0000 0000 0000 0000 0000 0001
  - `x22 < x23 // signed`
    - –1 < +1
  - `x22 > x23 // unsigned`
    - +4,294,967,295 > +1

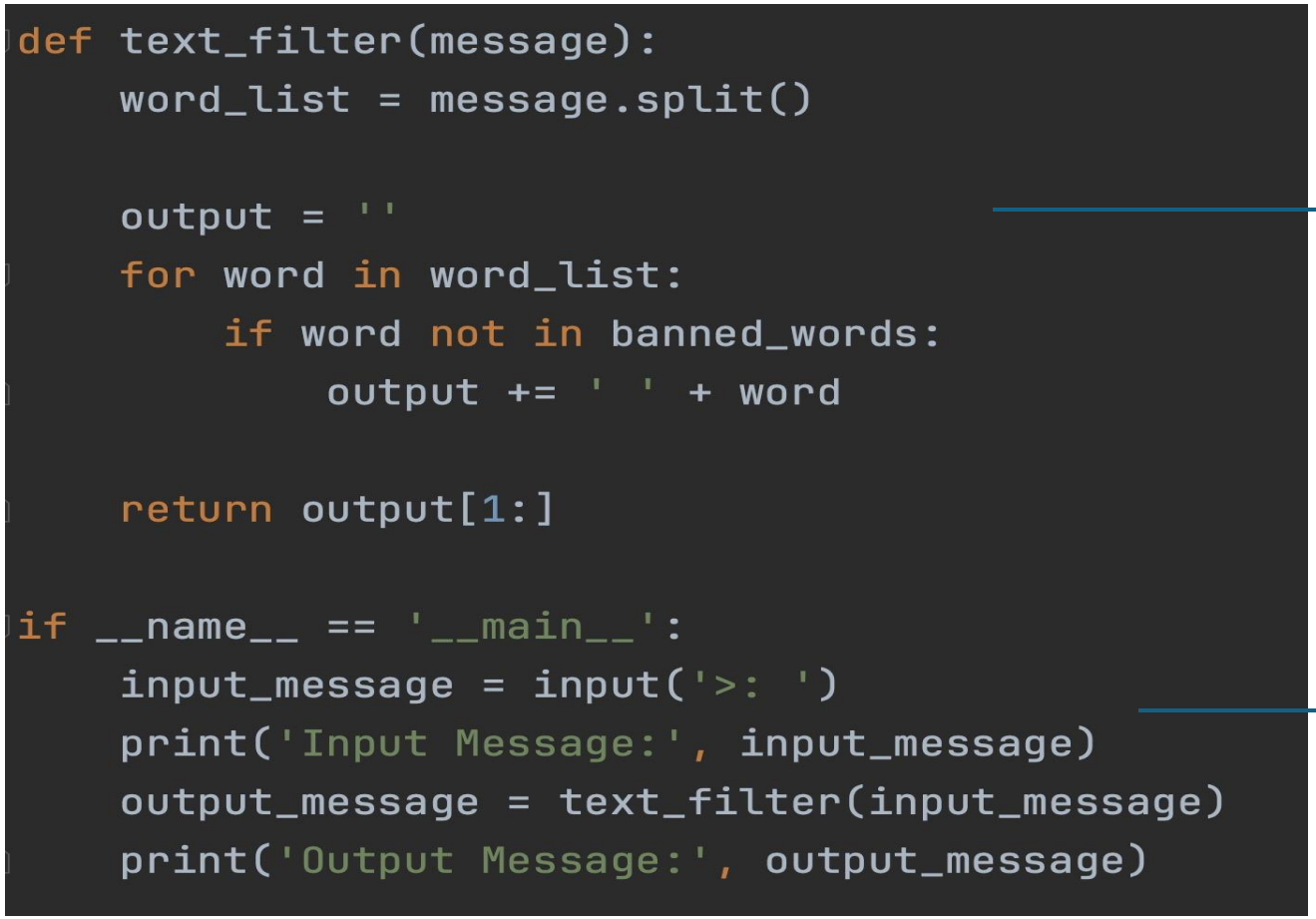
What is a Procedure?

Procedure: A stored subroutine that performs a specific task based on the **parameters** with which it is provided.

Used for: procedure or function is one tool programmers use to structure programs, both to make them easier to understand and to allow code to be reused.

def in Python

Procedure Calling



Procedure Calling

- Steps required
  1. Place parameters in registers x10 to x17
  2. Transfer control to procedure
  3. Acquire storage for procedure
  4. Perform procedure's operations
  5. Place result in register for caller
  6. Return to place of call (address in x1)

Register	Name	Mnemonic	Usage
`x0`	zero	`zero`	Always **0** (hardwired)
`x1`	ra	`ra` (return address)	Stores return address (`jal`, `jalr`)
`x2`	sp	`sp` (stack pointer)	Points to the top of the stack
`x3`	gp	`gp` (global pointer)	Global variables (OS, not common)
`x4`	tp	`tp` (thread pointer)	Used for thread-local storage
`x5` - `x7`	t0-t2	Temporary registers	Used for intermediate values
`x8`	s0/fp	`s0` or `fp` (frame pointer)	Stack frame pointer
`x9`	s1	`s1`	Saved register
`x10` - `x11`	a0-a1	Argument registers	Function arguments, return values
`x12` - `x17`	a2-a7	Argument registers	More function arguments
`x18` - `x27`	s2-s11	Saved registers	Preserved across function calls
`x28` - `x31`	t3-t6	Temporary registers	Used for calculations, not preserved

Stack ！ ！

- The **stack** is a region of memory used to manage **function calls, local variables, and register storage**. Even though **RISC-V relies heavily on registers for efficiency**, the **stack is still necessary** in many cases. (last in first out)
- Why we need stack?

(1) Storing Return Addresses in Nested Function Calls

If a function **calls another function**, the return address stored in ra (**x1**) gets overwritten. To preserve it, the function must save ra on the **stack** before making another call.（return address: pc+4, pc program counter）

(2) Preserving Register Values

Some registers **must be preserved across function calls** (e.g., s0-s11). If a function modifies them, it should **store them on the stack before modifying them and restore them before returning**.

(3) Handling More Arguments Than Available Registers

RISC-V **only provides a0-a7 (x10-x17) for function arguments**. If a function has **more than 8 arguments**, the extra arguments **must be stored on the stack**

(4) Allocating Local Variables

If a function has **local variables** (e.g., arrays, structs, large data), they must be stored **on the stack**, since registers are limited.

(5) Recursive Functions

Each recursive function call **must store its own state (return address, local variables, arguments)**. This is naturally handled using the **stack**.

Procedure Call Instructions

- Procedure call: jump and link

  `jal x1, ProcedureLabel`

  - Address of following instruction put in x1
  - Jumps to target address

- Procedure return: jump and link register

  `jalr x0, 0(x1)`

  - Like jal, but jumps to 0 + address in x1
  - Use x0 as rd (x0 cannot be changed)
  - Can also be used for computed jumps
    - e.g., for case/switch statements

Leaf Procedure Example

- C code:

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

  - Arguments g, …, j in x10, …, x13
  - f in x20
  - temporaries x5, x6
  - Need to save x5, x6, x20 on stack

g	h	i	j	f
x10	x11	x12	x13	x20

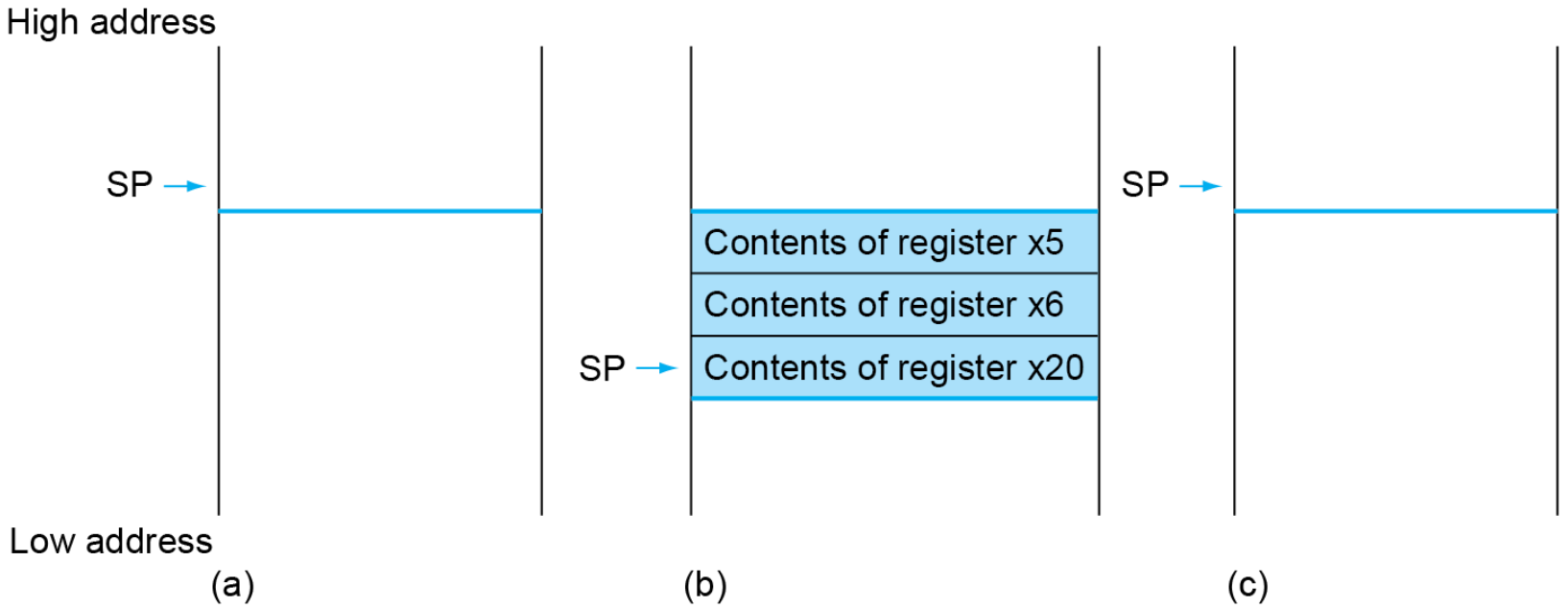
Leaf Procedure Example

```
leaf_example:
    addi sp, sp, -12  # Allocate 12 bytes stack space
    sw   x5, 8(sp)   # Save x5 to stack (offset 8)
    sw   x6, 4(sp)    # Save x6 to stack (offset 4)
    sw  x20, 0(sp)   # Save x20 to stack (offset 0)

    add  x5, x10, x11 # x5 = x10 + x11
    add  x6, x12, x1  # x6 = x12 + x1
    sub  x20, x5, x6  # x20 = x5 - x6
    addi x10, x20, 0  # Store result in x10 (return value)

    lw   x20, 0(sp)   # Restore x20 from stack
    lw   x6, 4(sp)    # Restore x6 from stack
    lw   x5, 8(sp)   # Restore x5 from stack
    addi sp, sp, 12   # Free stack space
```

why do I have to put value in the stack???

This function is **modifying x5, x6, and x20**, which may contain important values **used by the caller**.
Since these values **must be restored before returning**, they are **saved onto the stack first**.

Data on the Stack



Register Usage

- x5 – x7, x28 – x31: temporary registers
  - Not preserved by the callee

- x8 – x9, x18 – x27: saved registers
  - If used, the callee saves and restores them

Preserved	Not preserved
Saved registers: x8-x9, x18-x27	Temporary registers: x5-x7, x28-x31
Stack pointer register: x2(sp)	Argument/result registers: x10-x17
Frame pointer: x8(fp)	
Return address: x1(ra)	
Stack above the stack pointer	Stack below the stack pointer

Non-Leaf Procedures

- Procedures that call other procedures (caller)
- 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 fact (int n)
{
  if (n < 1) return f;
  else return n * fact(n - 1);
}
```

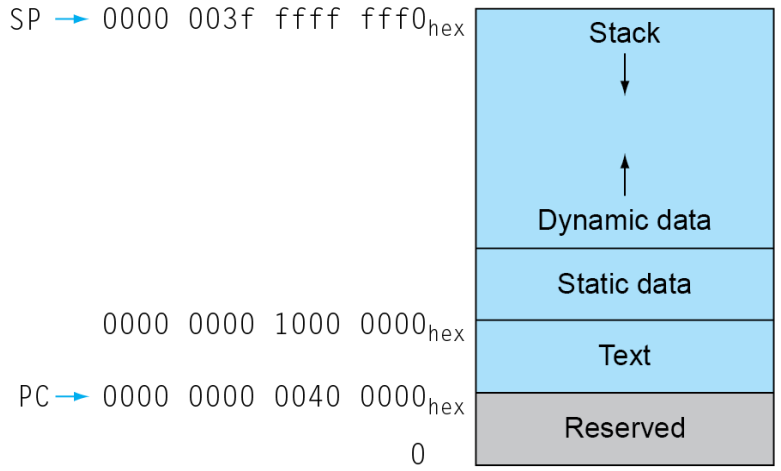
  - Argument n in x10
  - Result in x10

Non-Leaf Procedure Example

```
addi sp, sp, -8    // adjust stack for 2 items
  sw x1, 4(sp)     // save the return address
  sw x10, 0(sp)    // save the argument n

addi     x5, x10, -1        // x5 = n - 1
bge      x5, x0, L1         // if (n - 1) >= 0, go to L1

 L1: addi x10, x10, -1  // n >= 1: argument gets (n – 1)
     jal  x1, fact      // call fact with (n – 1)

 addi  x6, x10, 0    // return from jal: move result of fact
                        (n - 1) to x6:
 lw    x10, 0(sp)    // restore argument n
 lw    x1, 4(sp)     // restore the return address
 addi  sp, sp, 8     // adjust stack pointer to pop 2 items

 mul      x10, x10, x6      // return n * fact (n – 1)

jalr      x0, 0(x1)      // return to the caller
```

Memory Layout

- Text: program code
- Static data: global variables
  - e.g., static variables in C, constant arrays and strings
  - x3 (global pointer) initialized to address allowing ±offsets into this segment
- Dynamic data: heap
  - E.g., malloc in C, new in Java
- Stack: automatic storage



Example of stack

```
void FunctionA()
{
   int a = 10;
   FunctionB();
}

void FunctionB()
{
   int b = 20;
}

void Main()
{
   FunctionA();
}
```

Execution Process & Stack Changes:

1. Main() starts execution, and its **stack frame** is pushed onto the stack.
2. Main() calls FunctionA(), pushing FunctionA()'s **stack frame**.
3. FunctionA() declares int a = 10;, storing a on the stack.
4. FunctionA() calls FunctionB(), pushing FunctionB()'s **stack frame**.
5. FunctionB() declares int b = 20;, storing b on the stack.
6. FunctionB() finishes, and its stack frame is popped.
7. FunctionA() finishes, and its stack frame is popped.
8. Main() finishes, and its stack frame is popped, clearing the stack.

Example of stack