# Computer Architecture
# Chapter 2

## Instructions: Language of the computer
## 2

**Joonho Kong**
**School of EE, KNU**

# Instructions for Making Decisions

**Conditional branch / unconditional branch**

- RISC-V assembly codes are executed sequentially without branch instructions
  - Next instruction's memory address: PC+4

- Sometimes, we need to make a decision
  - Whether or not we execute certain instructions

- Branch instructions control program counter, not general purpose registers

**In C language, if-else, loop, switch-case, function call, goto, etc.**

# Instructions for Making Decisions

## Conditional branch instruction

- **beq (branch if equal)** **beq rs1, rs2, Label**
  : Go to the Label statement if the values in register rs1 and register rs2 are equal

- **bne (branch if not equal)** **bne rs1, rs2, Label**
  : Go to the Label statement if the values in register rs1 and register rs2 are not equal

- Label: designates a specific memory address that corresponds to a certain instruction

## SB-type instruction format

- Below is SB-type instruction format

| 7-bit | 5-bit | 5-bit | 3-bit | 5-bit | 7-bit |
|---|---|---|---|---|---|
| imm [12,10:5] | rs2 | rs1 | funct3 | imm [4:1,11] | opcode |

# Instructions for Making Decisions

**Example)**

**What is the compiled RISC-V code for following C if statement?**

```
if (i == j)                    f = g + h;
else                  f = g - h;
// corresponding register number is x19 ~ x23 for the variables f~j.
```

```
        bne     x22, x23, Else  // if (i != j), go to label  named Else
        add     x19, x20, x21   // f = g + h
        beq     x0, x0, Exit    // terminate the function
Else:   sub  x19, x20, x21          // Else label, f = g - h
Exit:                               //exit the statement.
```

# Instructions for Making Decisions

## Loop

- RISC-V provides instructions for the comparison (SB-type).

- **blt (branch if less than)** **blt rs1, rs2, Label**
  : takes the branch to Label if the value in rs1 is smaller than the value in rs2

- **bge (branch if greater than or equal)** **bge rs1, rs2, Label**
  : takes the branch to Label if the value in rs1 is greater or equal to the value in rs2

- **bltu and bgeu**
  : bltu and bgeu treat unsigned number

# Instructions for Making Decisions

**Loop example)**

There is a traditional loop in C:

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

( i and k correspond to register x22 and x24 and base address of the array save is in x25)

What is the RISC-V assembly code?

# Instructions for Making Decisions

## Case / Switch statement

### Branch address table (Branch table)

- A table of addresses of alternative instruction sequences
- The program needs only to index into the table
- The program will branch to the appropriate sequence

| Registers | Values |
|---|---|
| x8 | Stores the memory address of Label A |
| x9 | Stores the memory address of Label B |
| x10 | Stores the memory address of Label C |
| x11 | Stores the memory address of Label D |

```
switch(c) {
   case ‘A’: …
   case ‘B’: …
   case ‘C’: …
   case ‘D’: …
}
```

### Procedure of working using branch table

1) load the appropriate entry from the branch table into a register.
2) It branches using the address in the register.
   - RISC-V includes an indirect jump instruction
   - jalr or jal can be used for unconditional branch

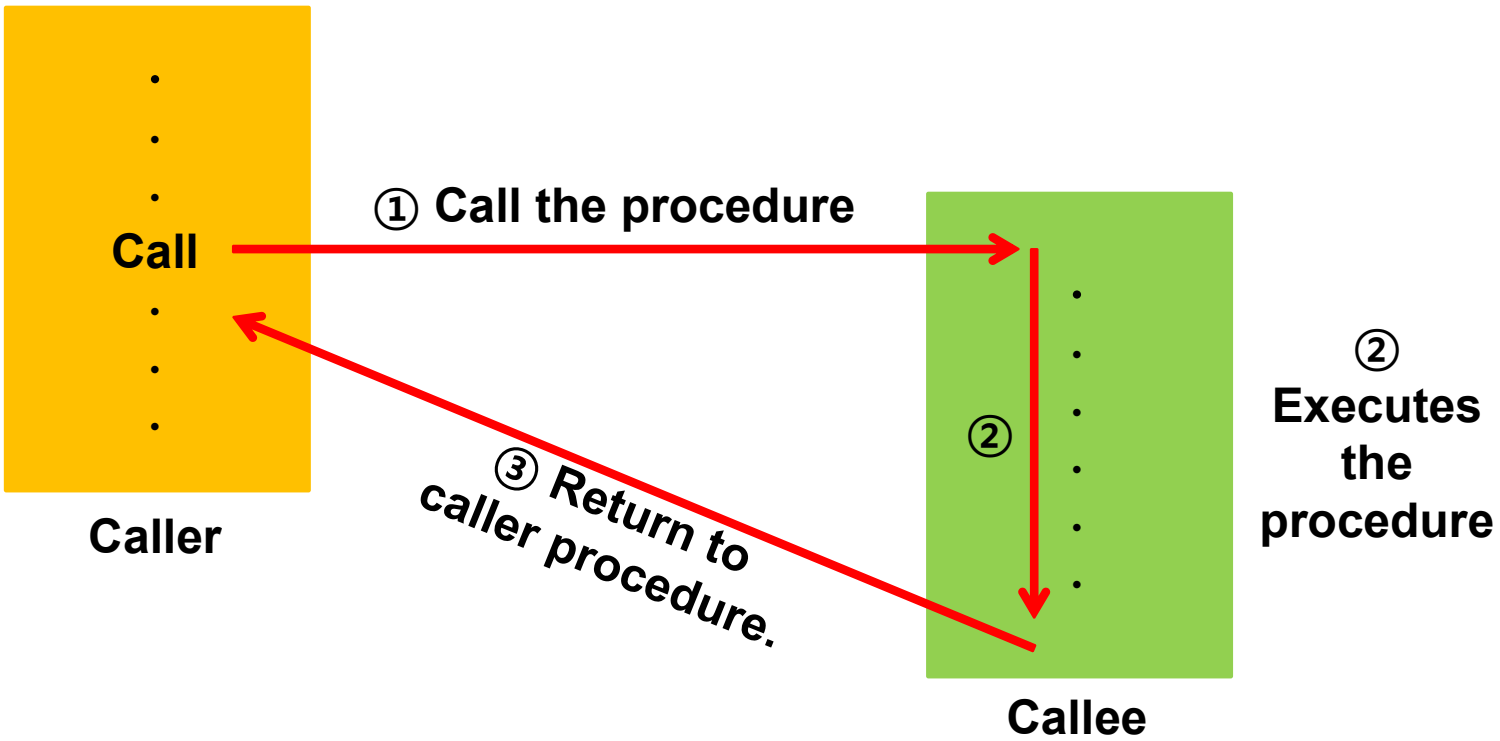
# Supporting Procedures in Computer H/W

## Procedure

- Procedure performs a specific task with the parameters.
- Procedure is similar to “function” in C-language
- Caller: a procedure that calls another procedure and provides the necessary parameter values
- Callee: a procedure that is called by the caller procedure and returns to the caller when it finishes the procedure execution.
- We need instructions to branch to the callee and to return to the caller
- We also need to save data in registers during procedure call
  - Some data may be overwritten by another procedure

# Supporting Procedures in Computer H/W

## The execution of procedures



# Supporting Procedures in Computer H/W

## jal instruction (Uj-type)

- **jump-and-link instruction (jal)** is represented with assembly codes as follows:

**jal rd, Label**

- jal instruction branches to an Label and saves the return address (PC+4) in the rd
- RISC-V computer compiles this label as an immediate value (Procedure address)

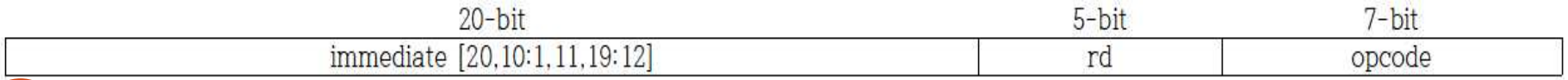
**jal x0, Label** //unconditionally branch to Label

- Since x0 is hard-wired to zero, the effect is to discard the return address

**jal** instruction can also be used to perform an unconditional branch

**jal x1, Label** – What does it mean? (x1 is typically used for storing return address)

## Uj-type instruction format



# Supporting Procedures in Computer H/W

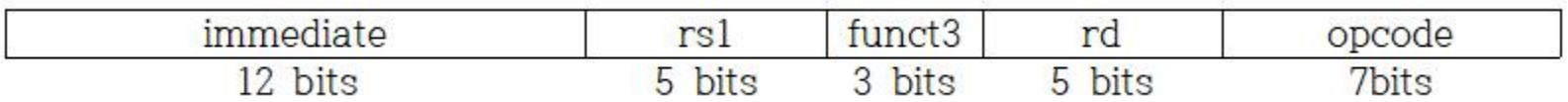
## jalr instruction (I-type) – can be used for return

- **jump-and-link register (jalr)** is usually used like:
  **jalr rd, offset(rs)**

- Unlike jal instruction, jalr instruction uses “base address + offset” as a branch target while the PC+4 is saved to the register rd

**jalr x0, 0(x1)** – What does it mean?

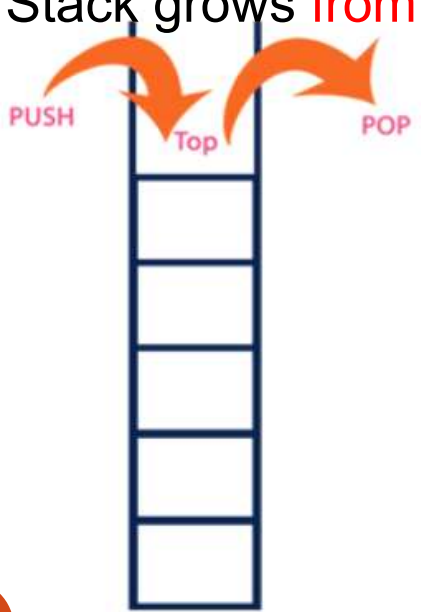
**I-type**



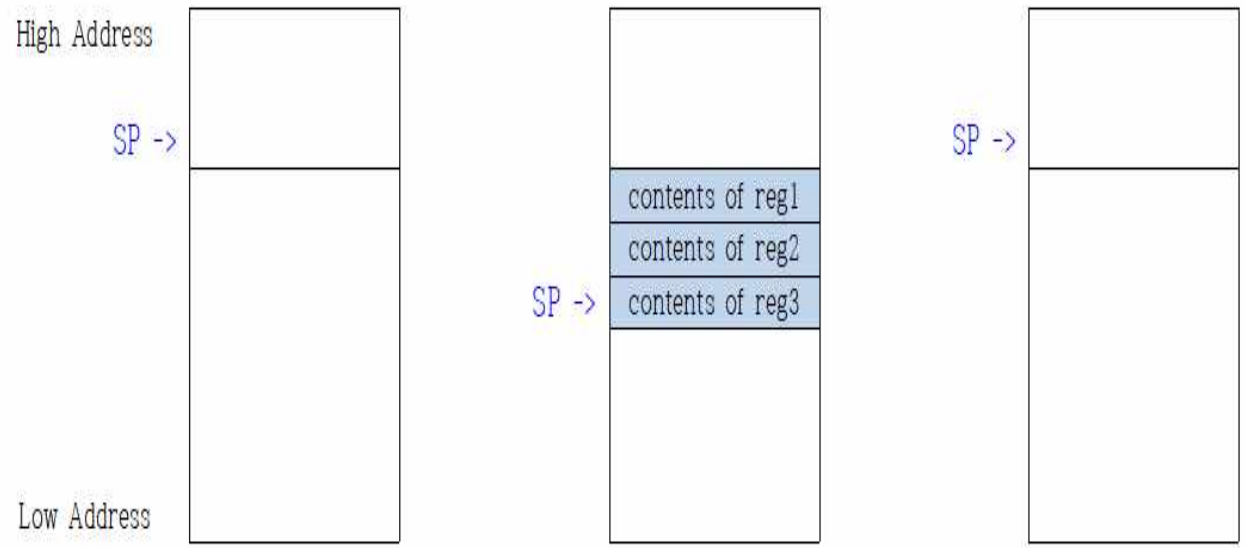
# Supporting Procedures in Computer H/W

**Using More Registers**

- We need to spill registers to memory using stack which is a last-in-first-out queue
- Because of limited size of register, we must use the stack for saving the value of register
- A stack needs a pointer to the most recently allocated address in the stack
- In RISC-V, the stack pointer (sp) is generally saved in register x2
- Stack grows from higher addresses to lower addresses




Push and Pop of stack



# Supporting Procedures in Computer H/W

**Example when using more register**

Q) What is the compiled RISC-V assembly code from following C-procedure:

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

(The parameter g, h, i, and j correspond to the argument registers x10, x11, x12, and x13,
and f corresponds to x20.)

# Supporting Procedures in Computer H/W

**Nested procedure example**

Q) Let's tackle a recursive procedure that calculates factorial:

```
long long int fact (long ling int n)
{
        if (n < 1)          return (1);
        else                return (n * fact (n – 1) );
}
```

What is the RISC-V assembly code?

The parameter variable n corresponds to the argument register x10.

# Supporting Procedures in Computer H/W

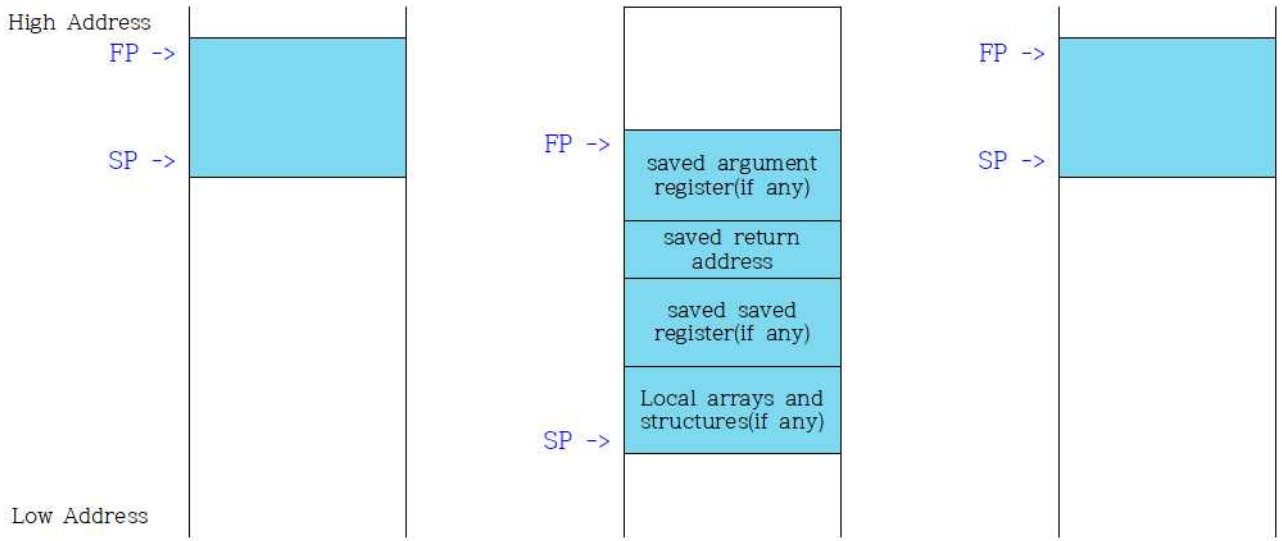
## RISC-V register convention

| Name | Register number | Usage | Preserved on call? |
|---|---|---|---|
| x0 | 0 | The constant value 0 | n.a. |
| x1 (ra) | 1 | Return address (link register) | yes |
| x2 (sp) | 2 | Stack pointer | yes |
| x3 (gp) | 3 | Global pointer | yes |
| x4 (tp) | 4 | Thread pointer | yes |
| x5-x7 | 5–7 | Temporaries | no |
| x8-x9 | 8–9 | Saved | yes |
| x10-x17 | 10–17 | Arguments/results | no |
| x18-x27 | 18–27 | Saved | yes |
| x28-x31 | 28–31 | Temporaries | no |

The figure is from:
https://books.google.co.kr/books?id=H7wxDQAAQBAJ&pg=PA102&lpg=PA102&dq=let's+tackle+a+recursive+procedure+that+calculates+factorial&source=bl&ots=bDQHPutm8Z&sig=EZtR0e520U5rcasOjWIaoKwHLzE&hl=ko&sa=X&ved=2ahUKEwil7-nYxrLdAhWaAYgKHTqSDAgQ6AEwAXoECAkQAQ#v=onepage&q=let's%20tackle%20a%20recursive%20procedure%20that%20calculates%20factorial&f=false

# Supporting Procedures in Computer H/W

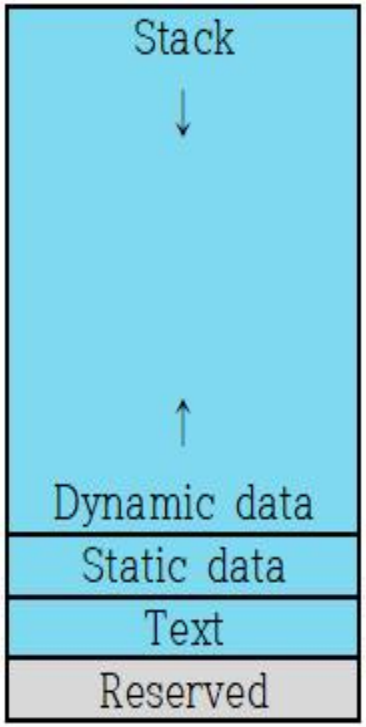
**Allocating space for new data on the stack**




- Procedure frame (or activation record)
  : the segments (memory area) where procedure's saved register and local variables in the stack
- Frame pointer (fp) points to the first doubleword of the frame of a procedure
  Stack pointer (sp) points to the current position of the stack

# Supporting Procedures in Computer H/W

**Allocating space for new data on the heap**




- The stack starts in the high end of the user addresses space and grows down
- The heap starts opposite site of stack data structure and grows up
  - To use heap space, we use dynamic memory allocation
    e.g., malloc() in C language

- This allocation allows the stack and heap to grow toward each other
- If we don't free dynamically allocated data, "memory leak" occurs
  → the operating system may be crashed
- To prevent it, programmer should free space after using the dynamic