Digital Design with Verilog

RISC-V : Introduction to Computer Architecture

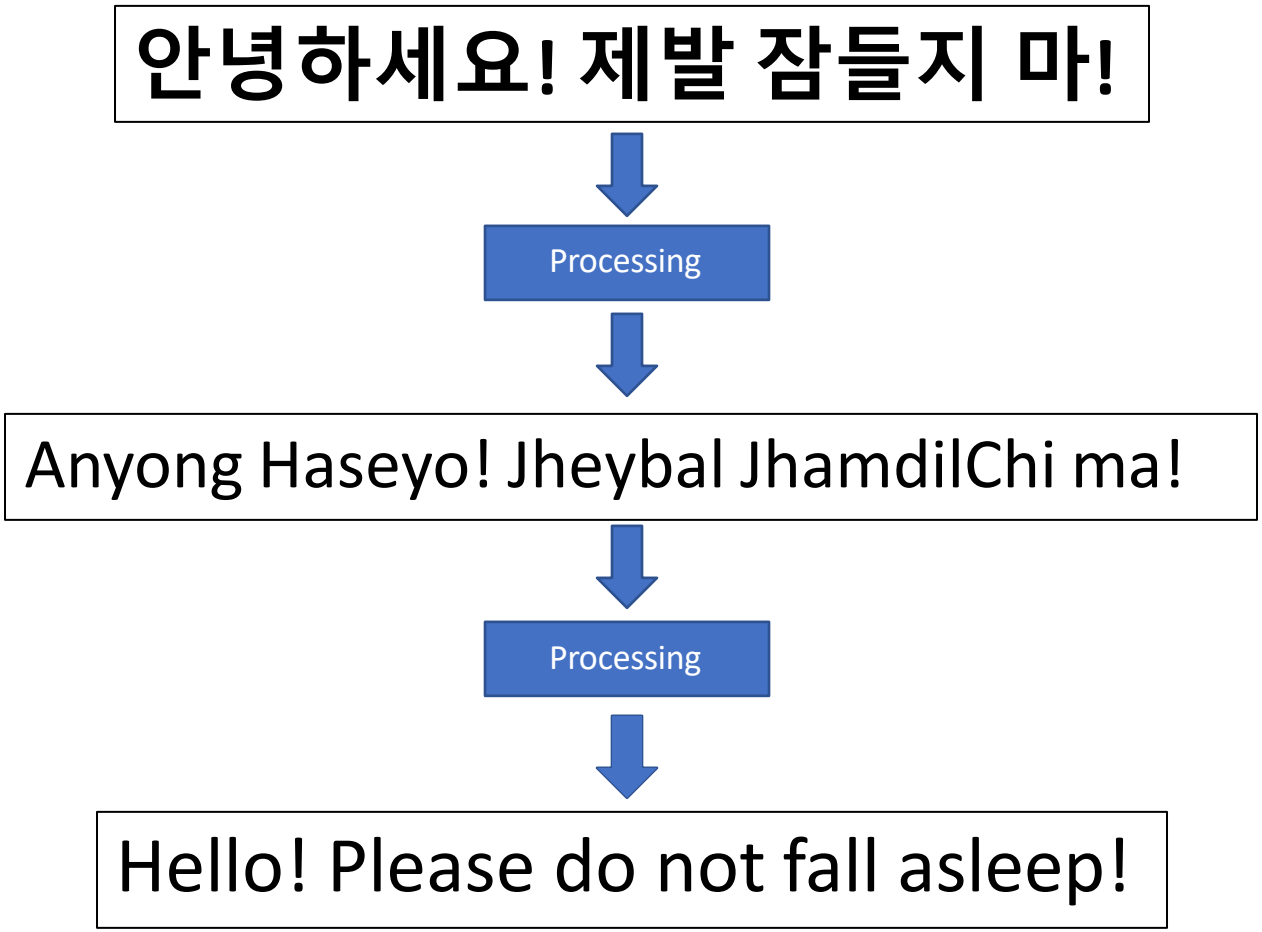
Lecture 26 : RISC-V Part 1

Disclaimer

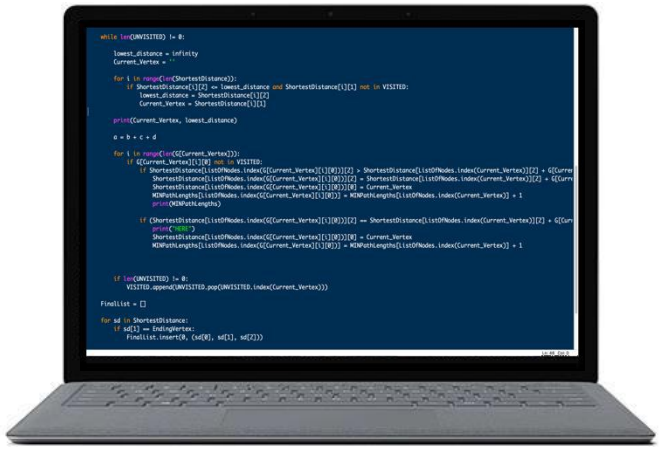
I do not own all the slides which I am going to present. The content is mostly from

- GIAN course titled "**Next-Generation Semiconductors: RISC-V, AI, TL-Verilog"** by Steeve Hoover. https://github.com/silicon-vlsi/gian-course-2024-IITBBS
- Introduction to Computer Architecture by Hasan Baig, https://www.hasanbaig.com
  - Most of the slides are copied/directly from his course content, please visit his site for reference.
- Digital Design and Computer Architecture, RISC-V Edition by Sarah L. Harris and David Money Harris.
- Other online publicly available sources. I tried my best to acknowledge the source wherever possible. I too might have missed some sources too ☺

Introduction



Introduction



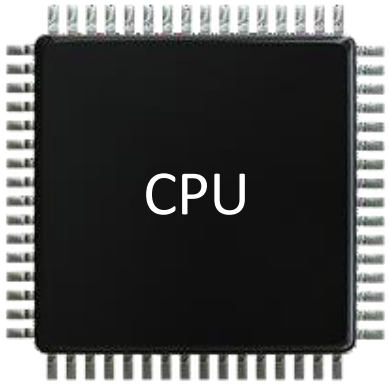
a = b + c + d

Compiler

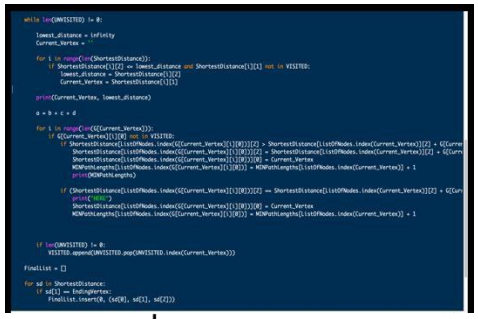
```
add   t0, s1, s2
add   s0, t0, s3
```

Assembler

000000	10001	10010	01000	00000	100000



Introduction



Compiler

```
add   t0, s1, s2
add   s0, t0, s3
```

Assembler

000000	10001	10010	01000	00000	100000

안녕하세요! 제발 잠들지 마!

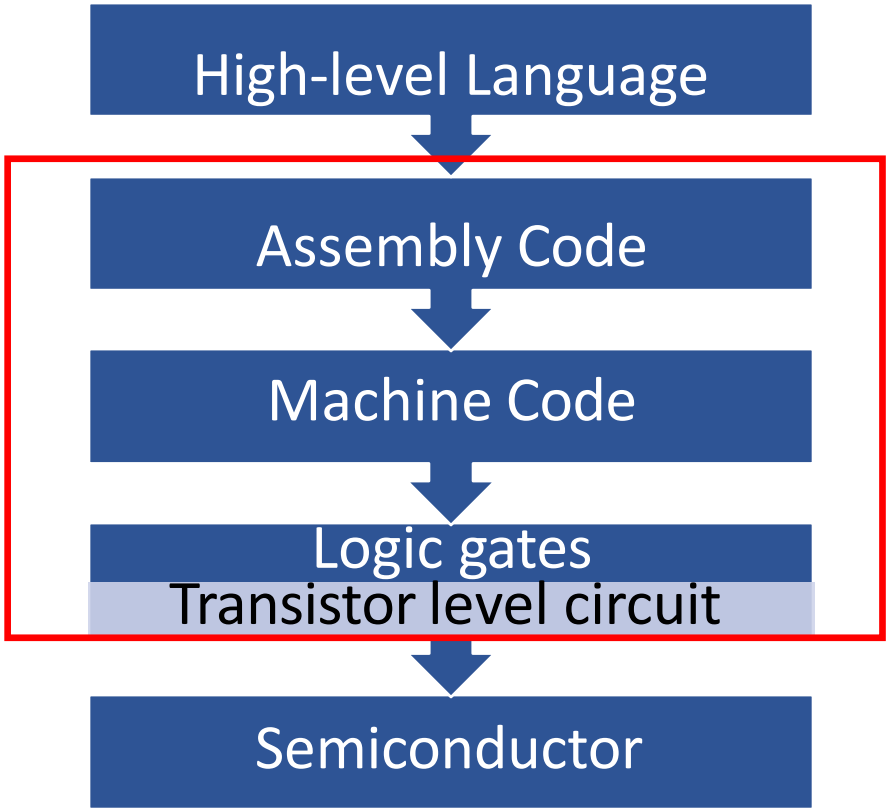
Processing

Anyong Haseyo! Cheybal ChamdilChi ma!

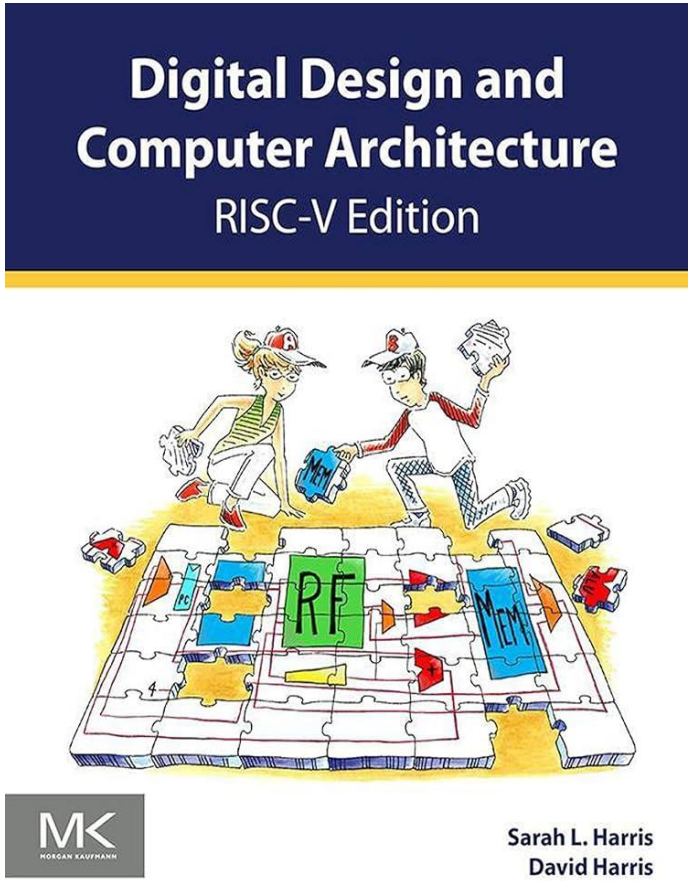
Processing

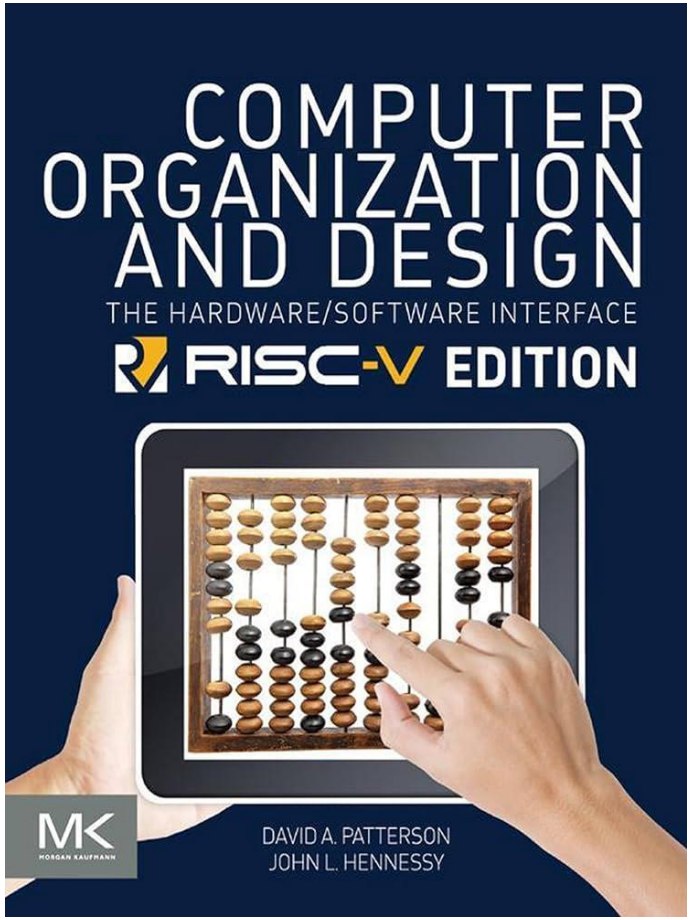
Hello! Please do not fall asleep!

Design Hierarchy



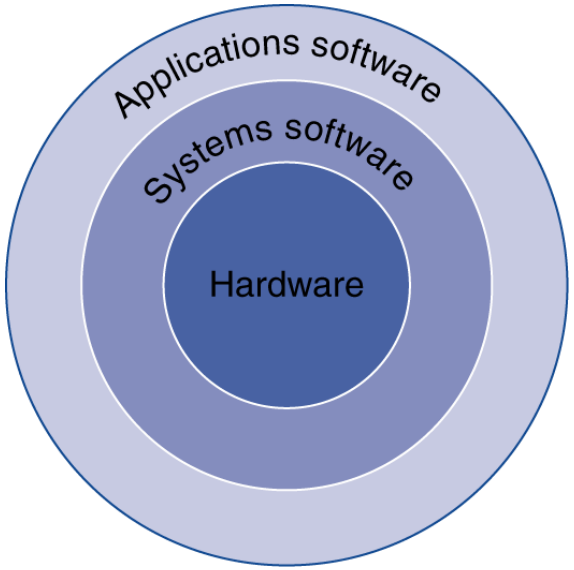
Textbooks


Digital Design and
Computer Architecture
RISC-V Edition
RF
MEM
Sarah L. Harris
David Harris


COMPUTER
ORGANIZATION
AND DESIGN
THE HARDWARE/SOFTWARE INTERFACE
RISC-V EDITION
DAVID A. PATTERSON
JOHN L. HENNESSY

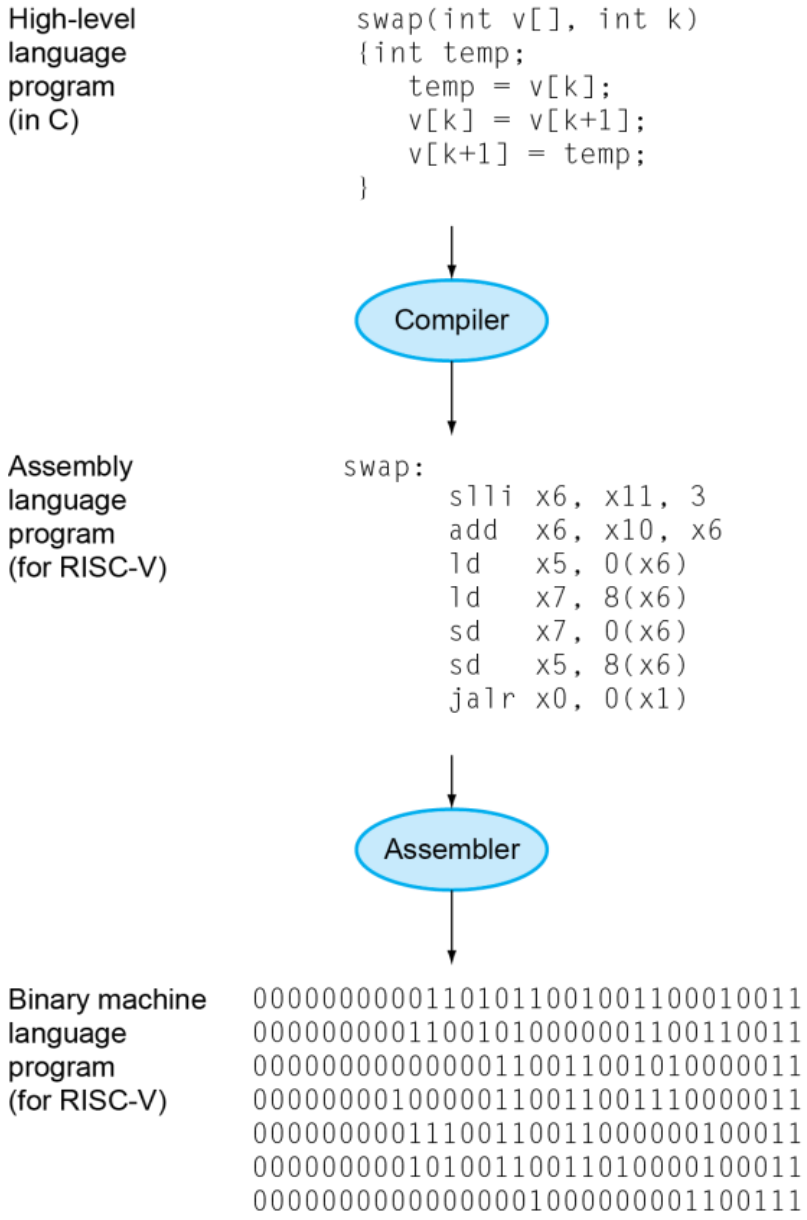
Below Your Program

- Application software
  - Written in high-level language
- System software
  - Compiler: translates HLL code to machine code
  - Operating System: service code
    - Handling input/output
  - Managing memory and storage
  - Scheduling tasks & sharing resources
- Hardware
  - Processor, memory, I/O controllers

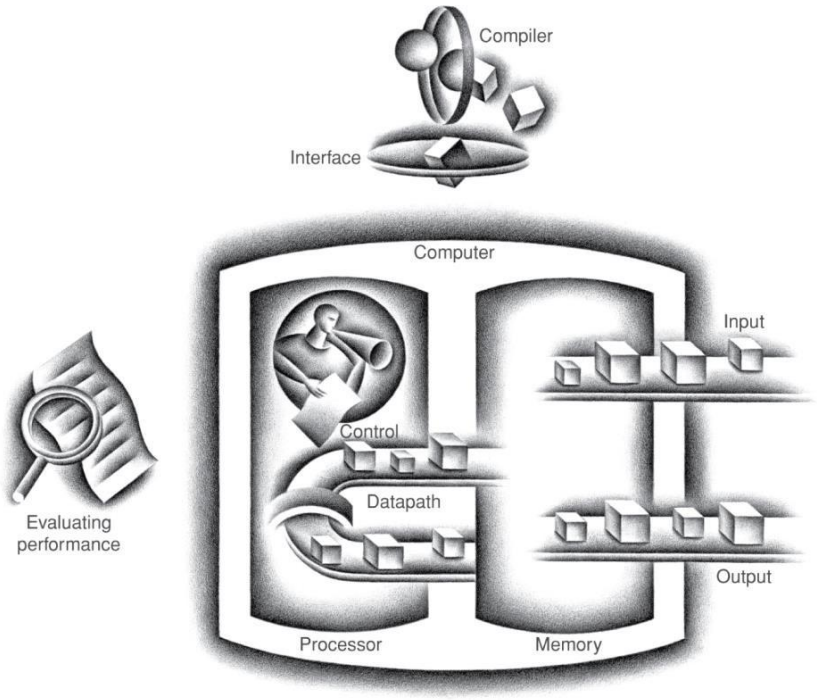


Below Your Program

- High-level language
  - Level of abstraction closer to problem domain
  - Provides for productivity and portability
- Assembly language
  - Textual representation of instructions
- Hardware representation
  - Binary digits (bits)
  - Encoded instructions and data



Components of a Computer



- Same components for all kinds of computer
  - Desktop, server, embedded
- Input/output includes
  - User-interface devices
    - Display, keyboard, mouse
  - Storage devices
    - Hard disk, CD/DVD, flash
  - Network adapters
  - For communicating with other computers

Instruction Set Architecture (ISA)

Instruction Set Architecture (ISA)

- Computer Language ➔ Instructions
- Computer Vocab ➔ Instruction set
- Different computers have different instruction sets
  - But with many aspects in common
- Early computers had very simple instruction sets
  - Simplified implementation
- Many modern computers also have simple instruction sets
- RISC-V was developed at UC Berkeley as open ISA, now managed by RISC-V foundation https://riscv.org/
- Similar ISAs have a large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, etc.

RISC-V Instruction Set: Arithmetic Operations

- Add and subtract, three operands
- Two sources and one destination

```
add a, b, c    // a gets b + c
```

- All arithmetic operations have this form

- For `a = b + c + d + e`

```
add  a, b, c
add  a, a, d
add  a, a, e
```

- **Design Principle 1**: *Simplicity favors regularity*
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

RISC-V Instruction Set: Arithmetic Operations

C code:

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

Compiled RISC-V code:

```
add t0, g, h   // temp t0 = g + h
add t1, i, j   // temp t1 = i + j
sub f, t0, t1  // f = t0 - t1
```

Storing Data in a Computer

- Memory
  - A place where data can be stored
  - Usually bigger in size

- Register file
  - Another place where data can be stored
  - Comparatively very small than main memory
  - A set of general purposes registers
    - Integers, addresses, characters, etc.
    - Programmers know the type

Design Principle 2: Smaller is faster

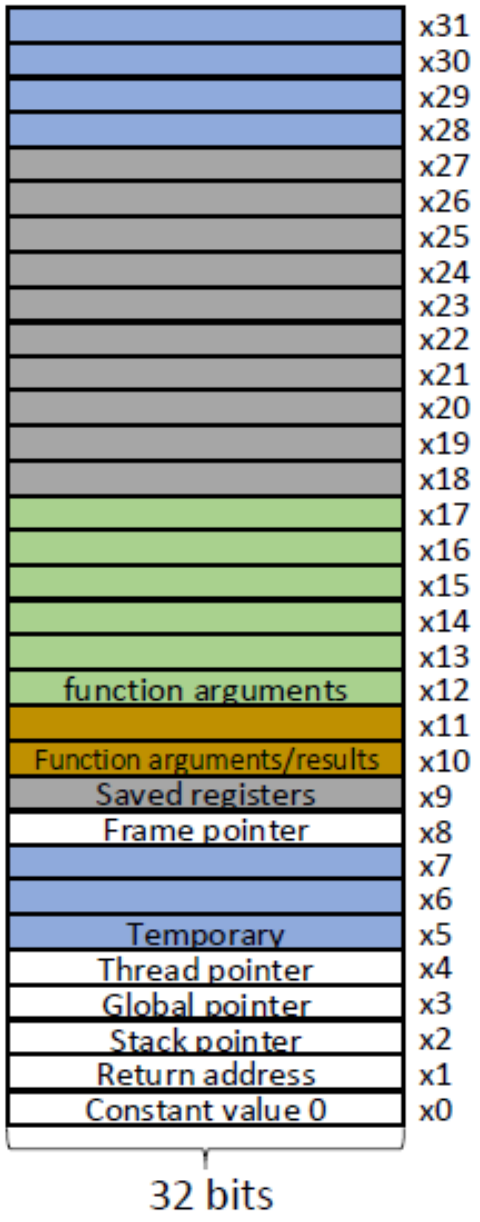
- Accessing register file is much faster than accessing memory

Register Operands

- Arithmetic instructions use register operands
- RISC-V has a 32 × 32-bit register file
  - Used for frequently accessed data
  - 32 general purpose registers - x0 to x31

RISC-V Registers

- x0: the constant value 0
- x1: return address
- x2: stack pointer
- x3: global pointer
- x4: thread pointer
- x5 – x7, x28 – x31: temporaries
- x8: frame pointer
- x9, x18 – x27: saved registers
- x10 – x11: function arguments/results
- x12 – x17: function arguments



Register Operands Example

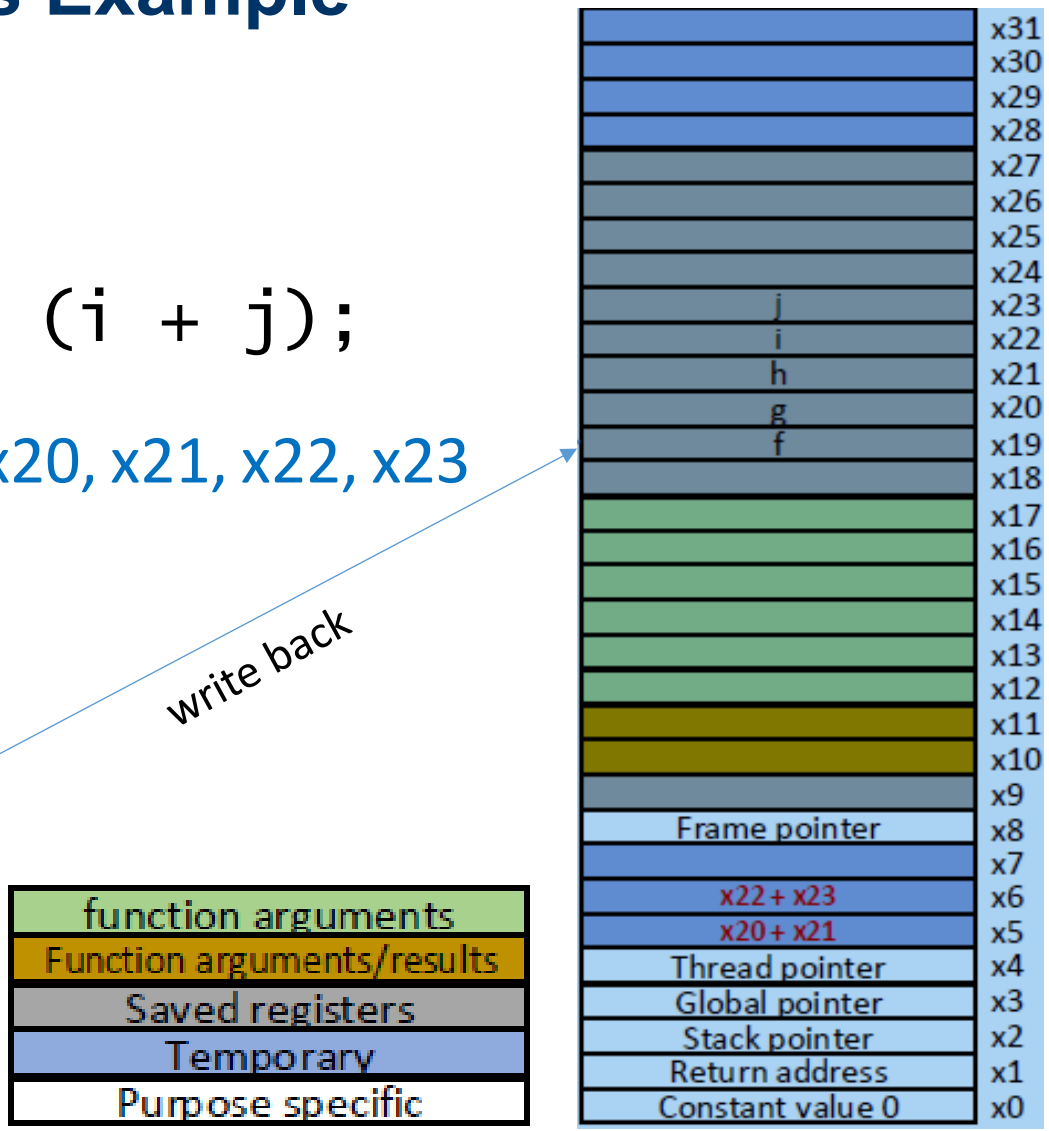
C code:

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

f, g , h, i, j ➔ x19, x20, x21, x22, x23

Compiled RISC-V code:

```
add x5,  x20, x21
add x6,  x22, x23
sub x19,  x5, x6
```



Immediate Operands

- Constant data specified in an instruction
  - a = a + 4

```
addi x22, x22, 4
```

- **Design Principle 3: Make the common case fast**
  - Small constants are common
  - Immediate operand avoids a load instruction

Zero Register

- Register 0 (zero) is the constant 0
  - Cannot be overwritten
  - If you need 0, it is already in x0. No need to use another register

- Useful for common operations
  - Move value between registers

```
add t2, s1, zero
```

  - Load a small constant into a register

```
addi    t3,   x0,    100
addi    t4,   x0,    -20
```

Conditional Operations

- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- beq rs1, rs2, L1 # if (rs1 == rs2) branch to (or go to) instruction labeled L1
- bne rs1, rs2, L2 # if (rs1 != rs2) branch to (or go to) instruction labeled L2
- blt rs1, rs2, L3 # if (rs1 < rs2) branch to (or go to) instruction labeled L3
- bge rs1, rs2, L4 # if (rs1 >= rs2) branch to (or go to) instruction labeled L4

```
# example of a label. Label itself is not an
instruction! L1:     ADD    x1, x2, x3
```

Branches compare two registers! Not with an immediate value

Conditional Operations

- C code:

```
if (i==j)
{
    f = g+h;
}
```

Pseudocode

```
    if (i != j) goto Skip
    f = g + h
Skip:  # this is a label
```

Again, Label is not an instruction
It is the location/address of an instruction

Conditional Operations

- C code:

```
if (i==j)
{
      f = g+h;
}
```

Variable	Register
f	x19
g	x20
h	x21
i	x22
j	x23

- Compiled RISC-V code:

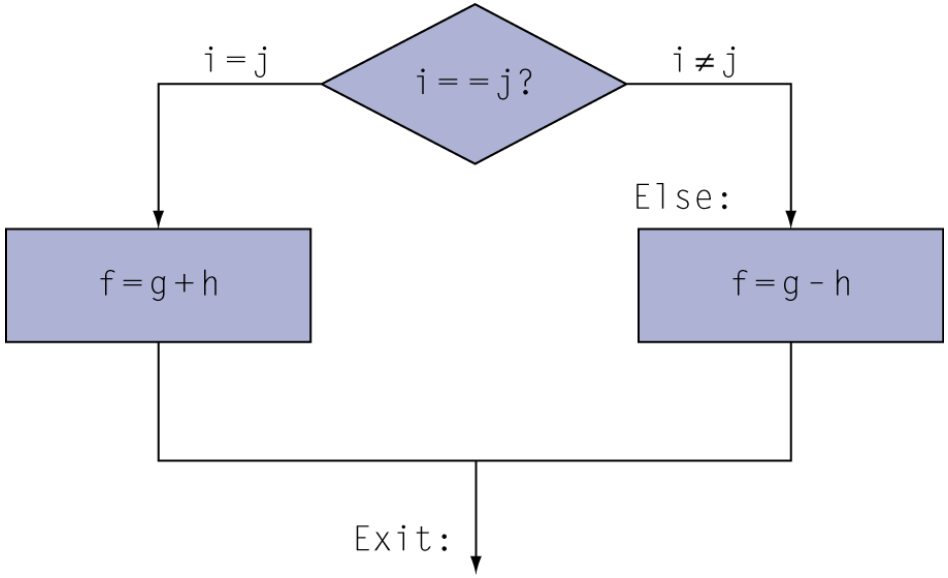
```
        bne   x22, x23,  Skip
        add   x19, x20,  x21

                  // more instructions if needed
skip: …
```

Assembler calculates addresses

Conditional Operations

- C code

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



- Pseudocode

```
        if (i != j) goto Else
        f = g + h
        goto  Exit
Else:   f = g  -  h
Exit:
```

← Do not forget to skip the else branch

Conditional Operations

- C code

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

Variable	Register
f	x19
g	x20
h	x21
i	x22
j	x23

- Compiled RISC-V code

```
        bne  x22,  x23,  Else
        add  x19,  x20,  x21
        beq x0,x0,Exit sub   // unconditional
Else:   x19, x20, x21
Exit:   …
```

Compiling Loop Statements

```
while (cond) {
        Statements
}
```

```
# Method 1
Loop:   if (! cond) goto Exit
        Statements
        goto Loop
Exit:
```

```
# Method 2
        goto Test
Loop:
        Statements
Test:   if (cond) goto Loop
```

Compiling Loop Statements

Implement the following loop with RISC-V instructions.

```
sum = 0;

i = 0;

while (i < 100) {

        sum += i;

        i += 1;

}
```

Variable	Register
i	x18
sum	x19
end	x20

Compiling Loop Statements

```
# Pseudocode
        i = 0
        sum = 0

loop:   if (!(i < 100)) goto exit
        sum += i
        i += 1
        goto loop

exit:
```

```
# Method 1 RISC-V code
        addi    x18, x0, 0
        addi    x19, x0, 0
        addi    x20, x0, 100

loop:   bge     x18, x20, exit
        add     x19, x19, x18
        addi    x18, x18, 1
        beq     x0, x0, loop

exit:
```

Variable	Register
i	x18
sum	x19
Value 100	x20

Compiling Loop Statements

Write the assembly code for the following pseudocode.

```
# Method 2 - Pseudocode
        i = 0
        sum = 0
        goto test
        sum += i
        i += 1
test:   if (i < 100) goto loop
```

Variable	Register
i	x18
sum	x19
Value 100	x20

Thank you