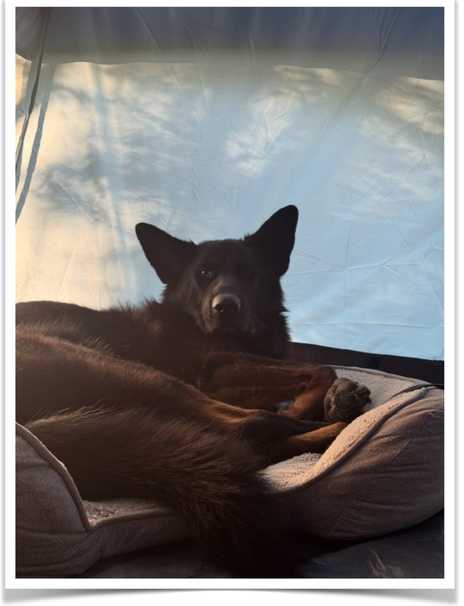
Cornell Bowers CIS
Computer Science



CS3410: Computer Systems and Organization

LEC11: RISC-V Data Transfer + Control Flow

Professor Giulia Guidi
Wednesday, October 1, 2025

Credits: Bala, Bracy, Garcia, Guidi, Kao, Sampson, Sirer, Weatherspoon

CC BY NC SA

Plan for Today

- Review: RISC-V Data Transfer — Vol. I
- RISC-V Data Transfer — Vol. II
- RISC-V Control Flow or Decision Making

PSA: Prelim & Prelim Survey [due **Friday 10/3**]
Please go to **OH** or post on **Ed**

CC BY NC SA

Review of RISC-V data transfer, so far

CC BY NC SA

RISC-V Assembly

High Level Language Program (e.g. C)

Compiler

Assembly Language Program (e.g., RISC-V)

Assembler

Machine Language Program (e.g., RISC-V)

Hardware

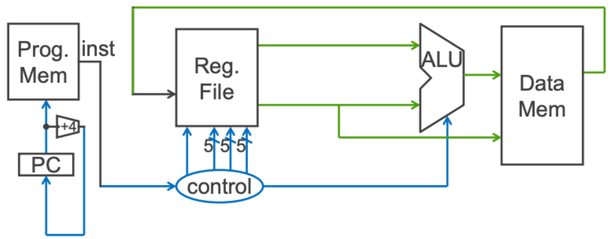
Hardware Architecture Description (e.g., block diagram)

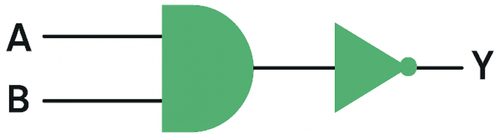
Logic Circuit Description (e.g., circuit schematic diagram)

```
temp = v[k];
v[k] = v[k+1];
```

```
lw x13, 0(x12) # temp = v[k]
lw x14, 4(x12) # x14 = v[k+1]
sw x14, 0(x12) # v[k] = v[k+1]
```

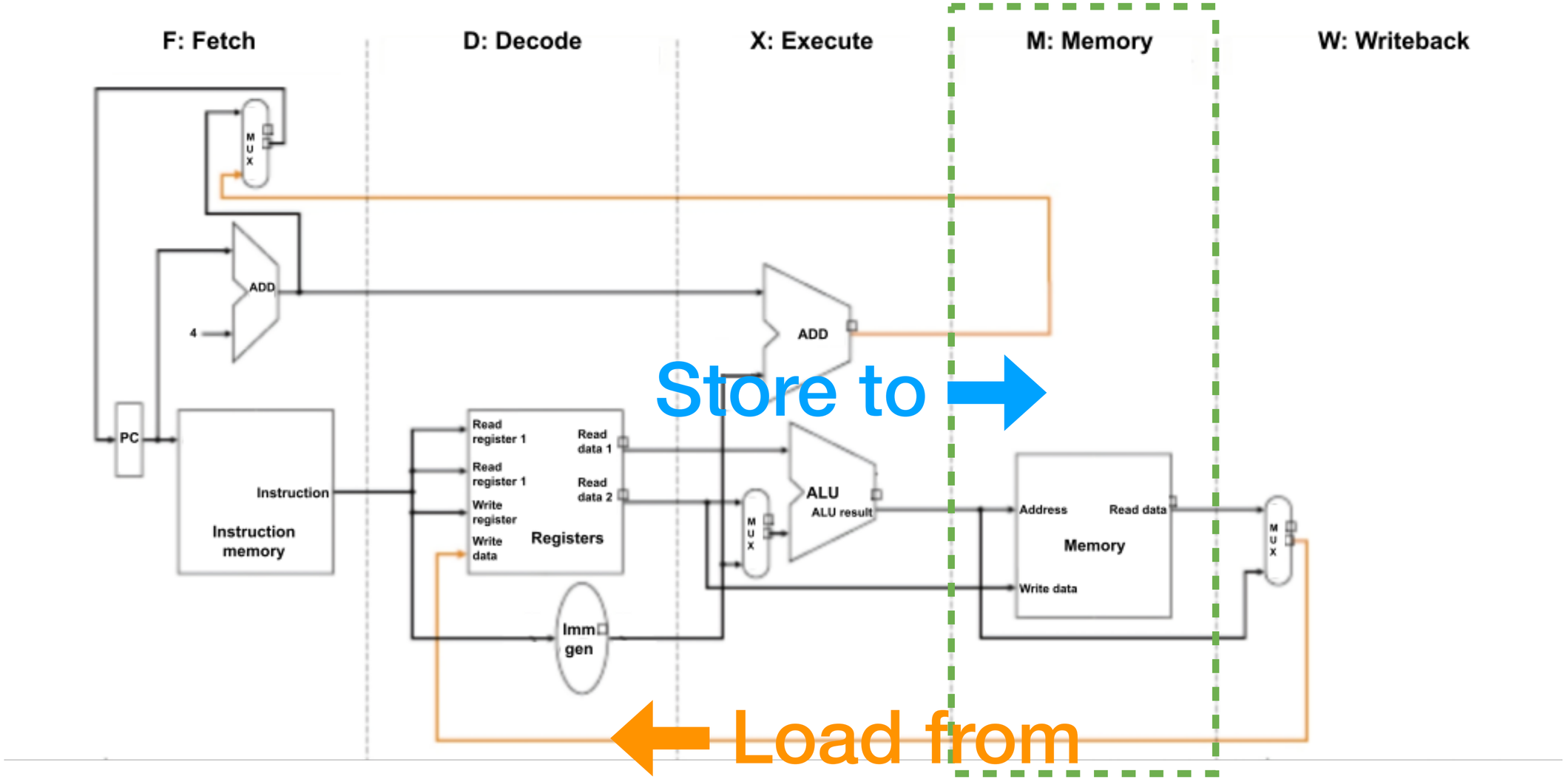
```
0000 0000 0000 0110 0010 0110 1000 0011
0000 0000 0100 0110 0010 0111 0000 0011
0000 0000 1110 0110 0010 0000 0010 0011
```





CC BY NC SA

Data Transfer: Load from and Store to Memory



CC BY NC SA

Load from Memory to Register

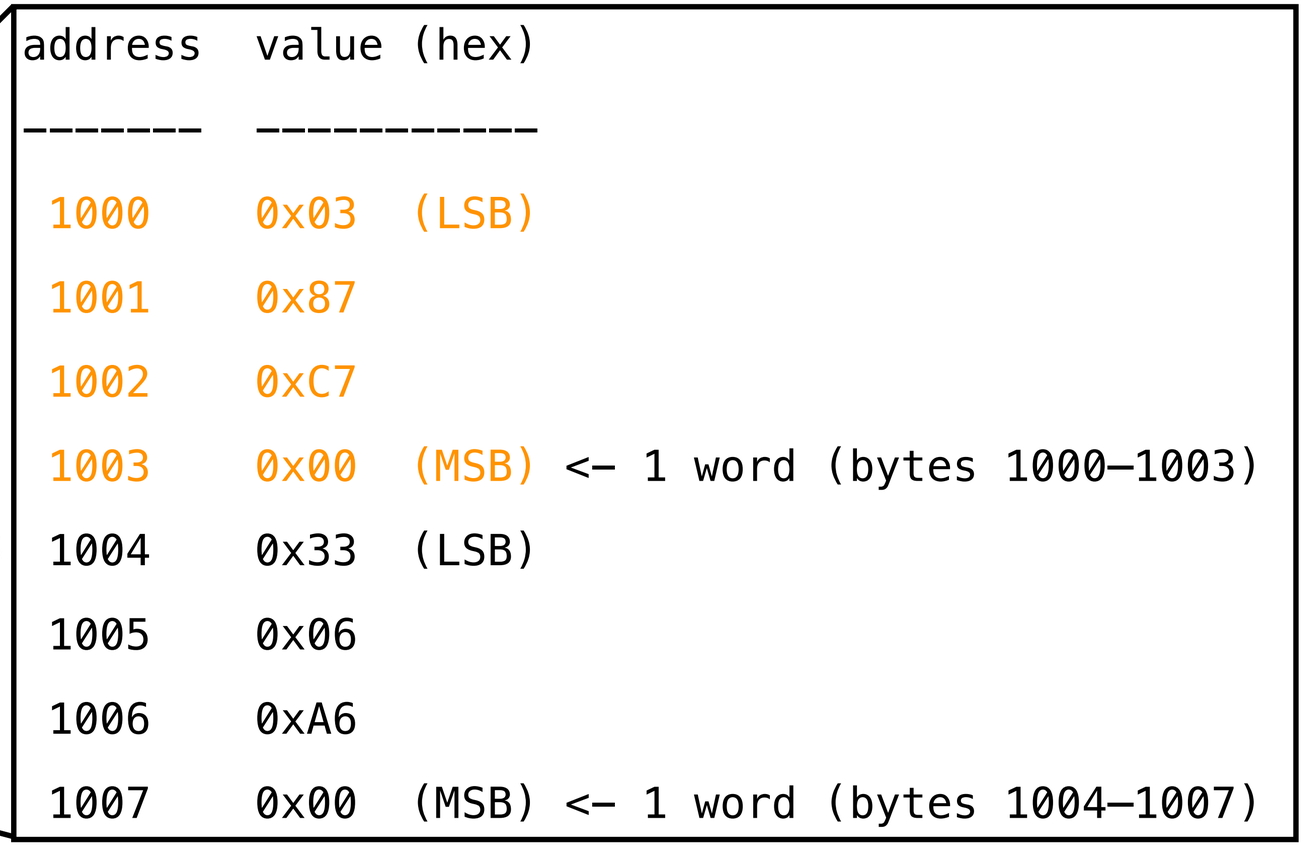
C code:

```
int A[100];
g = h + A[3];
```

1 word = 4 bytes

Using “load word” (`lw`) in RISC-V:

```
lw  x10, 12(x15)  # Reg x10 gets
add x11, x12, x10 # g = h + A[3]
```



→ `lw` updates `x10` in the register file

CC BY NC SA

Load from Memory to Register

In this example, we assumed we knew `x15` was the base address of `A[0]`

Code (or compiler) must have **loaded the base address** into `x15`

1 word = 4 bytes

Using “load word” (`lw`) in RISC-V:

```
la x15, A           # x15 = &A[0]  (address of first element)
lw  x10, 12(x15)    # Reg x10 gets A[3]
add x11, x12, x10   # g = h + A[3]
```

`x15:` address in memory (pointer to A[0])

`12:` offset in bytes but we load one **word** at a time

→ `lw` updates `x10` in the register file

CC BY NC SA

Load from Memory to Register

In this example, we assumed we knew `x15` was the base address of `A[0]`

Code (or compiler) must have **loaded the base address** into `x15`

1 word = 4 bytes

Using “load word” (`lw`) in RISC-V:

```
la x15, A           # x15 = &A[0]
lw  x10, 12(x15)    # Reg x10 gets A[3]
add x11, x12, x10 # g = h + A[3]
```

```
la rd, imm → lui  rd, imm        # put 20-bit imm into the top 20 bit of rd
             addi rd, rs1, imm   # put 12-bit imm into the low 12 bit of rd
```


CC BY NC SA

Store from Register to Memory

C code:

```
int A[100];
A[10] = h + A[3];
```

1 word = 4 bytes

Using “store word” (sw) in RISC-V:

```
lw  x10, 12(x15)  # Temp reg x10 gets A[3]
add x11, x12, x10 # Temp reg x11 gets h + A[3]
sw  x11, 40(x15)  # A[10] = h + A[3]
```

x15 + **12**

Offset must be a **multiple of 4**

x15 + **40**

➡ Store to (Data flow)

CC BY NC SA

CPU 5 Stages in RISC-V

Instruction	FETCH	DECODE	EXECUTE	MEMORY	WRITEBACK
`add rd, rs1, rs2`	✓	✓	✓		✓

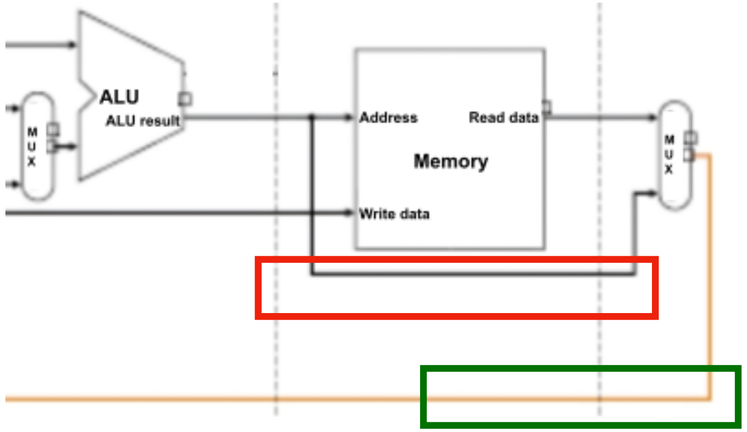
The instruction is fetched from the instruction memory

The registers `rs1` and `rs2` read, control signals set

ALU adds `rs1` + `rs2`

Don't need it, **no** memory access

ALU result written back to the destination register `rd`



CC BY NC SA

Poll

Q: Please select the CPU stage **not** involved in the operation: `sw x10, 36(x5)`

PollEv.com /gguidi

Or send **gguidi** to **22333**

CC BY NC SA

CPU 5 Stages in RISC-V

Instruction	FETCH	DECODE	EXECUTE	MEMORY	WRITEBACK
`sw rs2, offse(rs1)`	✓	✓	✓	✓	

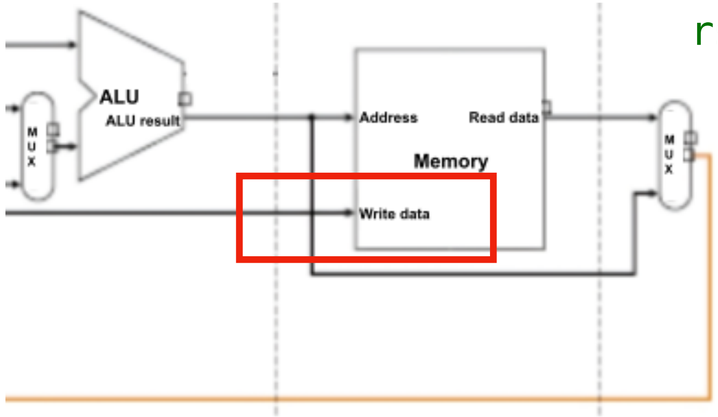
The instruction is fetched from the instruction memory

Decode instruction, read register `rs2`, `rs1`, extract the **offset**

ALU computes effective **address** = `rs1 base + offset`

Data memory **accessed**, `rs2` value **stored**

There's nothing to write back into registers



BY NC SA

CPU 5 Stages in RISC-V

Instruction	FETCH	DECODE	EXECUTE	MEMORY	WRITEBACK
`lw rd, offset(rs1)`	✓	✓	✓	✓	✓

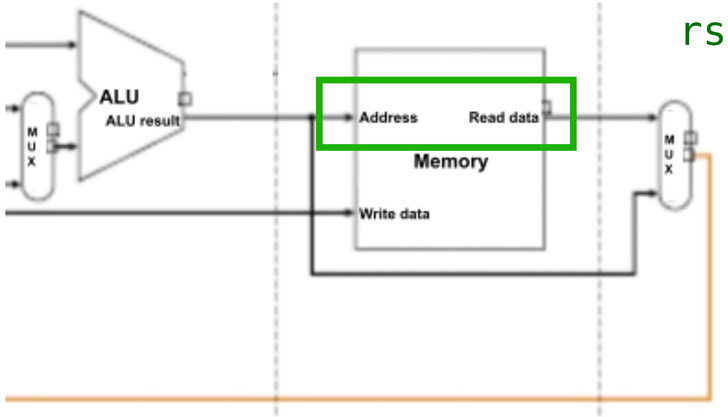
The instruction is fetched from the instruction memory

Decode instruction, read register `rs1`, extract the **offset (imm)**

ALU computes effective **address** = `rs1 base + offset`

Data memory **accessed**, value loaded

The loaded value is written back into destination register `rd`



CC BY NC SA

Ok, back to data transfer — Vol. II

CC BY NC SA

Loading and Storing Bytes

"load word" "store word" "load byte"

In addition to `lw` and `sw`, RISC-V has `lb` and `sb` "store byte"

CC BY NC SA

Loading and Storing Bytes

"load word" "store word" "load byte"

In addition to `lw` and `sw`, RISC-V has `lb` and `sb` "store byte"

Uses same format as `lw` and `sw`:

- E.g., `lb x10, 3(x11)` pointer to memory

  offset in bytes (doesn't have to be a multiple of 4 because I'm only loading 1 byte, i.e., one line in memory)

1. Compute the effective address = content of `x11` + `3`
   - Let us assume `3(x11)` contains the value `4` (decimal) and we use **16-bit** register
   - `4` (decimal) = `0000 0000 0000 0100` (16-bit binary)

CC BY NC SA

Loading and Storing Bytes

"load word" "store word" "load byte"

In addition to `lw` and `sw`, RISC-V has `lb` and `sb` "store byte"

Uses same format as `lw` and `sw`:

- E.g., `lb x10, 3(x11)` pointer to memory

  offset in bytes (doesn't have to be a multiple of 4 because I'm only loading 1 byte, i.e., one line in memory)

1. Compute the effective address = content of `x11` + `3`
   - Let us assume `3(x11)` contains the value `4` (decimal) and we use **16-bit** register
   - `4` (decimal) = `0000 0000 0000 0100` (16-bit binary)
2. Then, **load 1 byte** from memory at that address
   - The loaded **byte** is `0000 0100` (8-bit binary)

CC BY NC SA

Loading and Storing Bytes

"load word" "store word" "load byte"

In addition to `lw` and `sw`, RISC-V has `lb` and `sb` "store byte"

Uses same format as `lw` and `sw`:

- E.g., `lb x10, 3(x11)` pointer to memory

  offset in bytes (doesn't have to be a multiple of 4 because I'm only loading 1 byte, i.e., one line in memory)

2. Then, **load 1 byte** from memory at that address
   - The loaded **byte** is `0000 0100` (8-bit binary)
3. Finally, **sign-extend** the byte to 16 bits (we assumed `x10` is a **16-bit** register)
   - The final value in `x10` is `0000 0000 0000 0100` (16-bit binary) = `4` (decimal)

CC BY NC SA

Loading and Storing Bytes

"load word" "store word" "load byte"

In addition to `lw` and `sw`, RISC-V has `lb` and `sb` "store byte"

Uses same format as `lw` and `sw`:

- E.g., `lb x10, 3(x11)` pointer to memory
  offset in bytes (doesn't have to be multiple of 4)

1. Compute the effective address = content of `x11` + `3`
   - Ok **but what if** 3(x11) contains the value `3410` (decimal) and we use **16-bit** register
   - `3410` (decimal) = `0000 1101 0101 0010` (16-bit binary)
2. Then, **load 1 byte** from memory at that address
   - The loaded **byte** is `0101 0010` (8-bit binary)

CC BY NC SA

Loading and Storing Bytes

"load word" "store word" "load byte"

In addition to `lw` and `sw`, RISC-V has `lb` and `sb` "store byte"

Uses same format as `lw` and `sw`:

- E.g., `lb x10, 3(x11)` pointer to memory
  offset in bytes (doesn't have to be multiple of 4)

2. Then, **load 1 byte** from memory at that address
   - The loaded **byte** is `0101 0010` (8-bit binary)
3. Finally, **sign-extend** the byte to 16 bits (we assumed `x10` is a **16-bit** register)
   - The final value in `x10` is `0000 0000 0101 0010` (16-bit binary)
   - Ops! `0000 0000 0101 0010` (16-bit binary) = `82` (decimal) != `3410` (decimal)

CC BY NC SA

`lbu`

"load word" "store word" "load byte"

In addition to `lw` and `sw`, RISC-V has `lb` and `sb` "store byte"

Uses same format as `lw` and `sw`:

- E.g., `lb x10, 3(x11)` pointer to memory
  offset in bytes (doesn't have to be multiple of 4)

`lbu` = unsigned load byte It doesn't need to preserve the sign

CC BY NC SA

`lbu`

"load word" "store word" "load byte"

In addition to `lw` and `sw`, RISC-V has `lb` and `sb` "store byte"

Uses same format as `lw` and `sw`:

- E.g., `lb x10, 3(x11)` pointer to memory
  offset in bytes (doesn't have to be multiple of 4)

`lbu` = unsigned load byte — It doesn't need to preserve the sign: **zero extension**

But no `sbu`, why? — It doesn't matter! You're just writing the low 8 bits of a register directly to memory, so **no** extension

CC BY NC SA

Register versus Memory

10^9 tape/optical robot	**Andromeda**	2,000 years
10^6 disk	**Pluto**	2 years
100 memory	**New York City**	4 hrs
2 “something great coming up”	**this campus**	15 min
1 register	**my head**	1 min
[ns]		

CC BY NC SA

addi

The following two instructions:

```
lw  x10, 12(x15)  # temp reg x10 gets A[3]
add x12, x12, x10 # reg x12 = reg x12 + A[3]
```

Replace `addi`:

```
addi x12, value   # put value in A[3]
```

This involves going to New York City (load from memory)

The add immediate is so common that is deserves its own instruction

CC BY NC SA

Poll

```
addi x11, x0, 0x49C
sw   x11, 0(x5)
lb   x12, 0(x5)
```

Reminder: 1 word = 4 bytes

Reminder: 1 hex digit = 4-bit pattern

Let us assume 32-bit register

PollEv.com /gguidi

Or send **gguidi** to **22333**

CC BY NC SA

`sw` and `lb`

```
addi x11, x0, 0x49C
sw   x11, 0(x5)
lb   x12, 0(x5)
```

`addi` = add immidiate

`x0` is always 0!

So this just loads `0x49C` (hex) into `x11`

Reminder: 1 word = 4 bytes

Reminder: 1 hex digit = 4-bit pattern

Let us assume 32-bit register

CC BY NC SA

`sw` and `lb`

```
addi x11, x0, 0x49C
sw   x11, 0(x5)
lb   x12, 0(x5)
```

Reminder: 1 word = 4 bytes

Reminder: 1 hex digit = 4-bit pattern

Let us assume 32-bit register

So this just loads `0x49C` (hex) into `x11`

`sw` = store word (32-bit)

It takes the value in `x11` and writes 1 word to memory starting at address in `x5`

In memory at `x5`, the 32-bit value `0x0000049C` is stored

→ `sw` does **not** update any register in the register file; only memory is updated

CC BY NC SA

`sw` and `lb`

```
addi x11, x0, 0x49C
sw   x11, 0(x5)
lb   x12, 0(x5)
```

Reminder: 1 word = 4 bytes

Reminder: 1 hex digit = 4-bit pattern

Let us assume 32-bit register

So this just loads `0x49C` (hex) into `x11`

`sw` = store word (32-bit)

It takes the value in x11 and writes 1 word to memory starting at address in x5

In memory at x5, the 32-bit value `0x0000049C` is stored

memory address (little endian)	value
x5 + 0	0x9C
x5 + 1	0x04
x5 + 2	0x00
x5 + 3	0x00

CC BY NC SA

`sw` and `lb`

```
addi x11, x0, 0x49C
sw   x11, 0(x5)
lb   x12, 0(x5)
```

So this just loads `0x0000049C` (hex) into `x11`

`x5` stores `0x0000049C`

`lb` = load byte (8-bit), sign-extend to 32-bit

This loads the **first byte** from memory at x5

`x5 + 0 = 0x9C`

Reminder: 1 word = 4 bytes

Reminder: 1 hex digit = 4-bit pattern

Let us assume 32-bit register

→ `lb` updates `x12` in the register file

memory address (little endian)	value
x5 + 0	0x9C
x5 + 1	0x04
x5 + 2	0x00
x5 + 3	0x00

CC BY NC SA

`sw` and `lb`

```
addi x11, x0, 0x49C
sw   x11, 0(x5)
lb   x12, 0(x5)
```

So this just loads `0x0000049C` (hex) into `x11`

`x5` stores `0x0000049C`

`lb` = load byte (8-bit), sign-extend to 32-bit

This loads the **first byte** from memory at x5

`x5 + 0 = 0x9C`

$0x9C_{16} = 10011100_2 = -100_{10}$

MSB is 1 → negative!

Reminder: 1 word = 4 bytes

Reminder: 1 hex digit = 4-bit pattern

Let us assume 32-bit register

CC BY NC SA

sw and lb

```
addi x11, x0, 0x49C
sw   x11, 0(x5)
lb   x12, 0(x5)
```

So this just loads 0x0000049C (hex) into x11

x5 stores 0x0000049C

lb = load byte (8-bit), sign-extend to 32-bit

This loads the **first byte** from memory at x5

x5 + 0 = 0x9C

$0x9C_{16} = 10011100_2 = -100_{10}$

Reminder: 1 word = 4 bytes

Reminder: 1 hex digit = 4-bit pattern

Let us assume 32-bit register

Recall

Ok, what about hex? E.g., -1 = FFFF

If MSB = 0-7 ➡ then append 0s

Otherwise append Fs

x12 stores -100_{10} = $0xFFFFFF9C_{16}$

CC BY NC SA

Data Transfer Conclusion

- The memory is **byte-addressable**, but `lw` and `sw` access one **word** at a time
- A pointer (used by `lw` and `sw`) is just a memory address, we can add to it or subtract from it (using offset)
- Partial memory hierarchy (register versus DRAM)
- Bit sign-extension and zero-extension
- Review of endianess

CC BY NC SA

Ok, let’s move on to RISC-V control flow

CC BY NC SA

Computer Decisions Making

I.e., based on computation, do something different

- In programming languages, *if*-statement

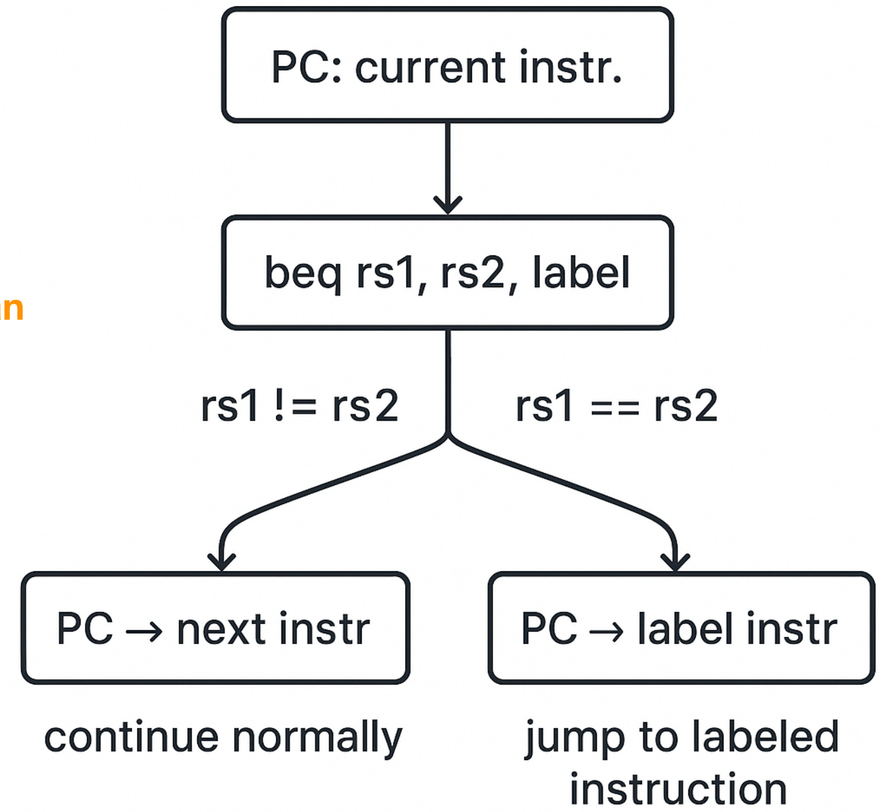
RISC-V *if*-statement instruction:

```
beq rs1, rs2, label
```

→ It marks the **address of an instruction**

"branch if equal"

→ If `rs1` == `rs2`, then go to the instruction at `label`



CC BY NC SA

Computer Decisions Making

I.e., based on computation, do something different

- In programming languages, *if*-statement

RISC-V *if*-statement instruction:

```
beq rs1, rs2, label
```

It marks the **address of an instruction**

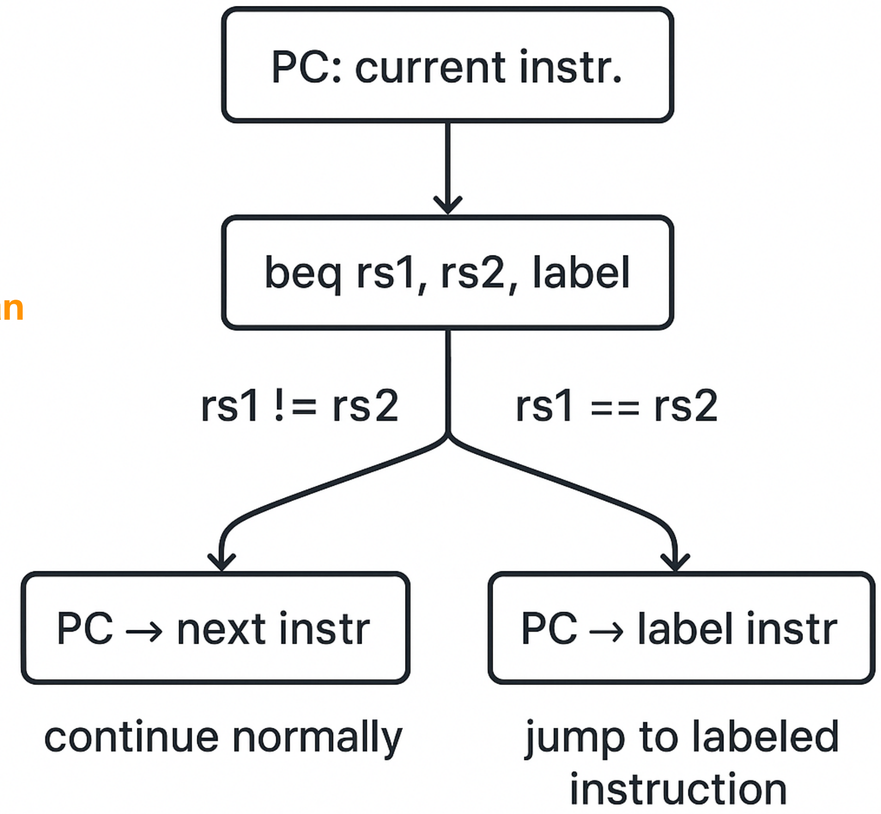
"branch if equal"

→ If `rs1` == `rs2`, then go to the instruction at `label`

```
bne rs1, rs2, label
```

"branch if **not** equal"

→ If `rs1` != `rs2`, then go to the instruction at `label`



CC BY NC SA

Types of Branches

- **Branch:** change of control flow
- **Conditional branch:** change control flow depending on outcome of comparison

`beq, bne`	
`blt, bge`	"branch if **less than**", "branch if **greater or equal than**"
`bltu, bgeu`	"branch if **less than**" unsigned, "branch if **greater than or equal to**" unsigned

The result of `x1 < x2` changes if we use `blt` (true) vs `bltu` (false)

`x1` = $1111_2 \rightarrow 15_{10}$ if unsigned and -1_{10} if signed

`x2` = $0000_2 \rightarrow 0_{10}$

CC BY NC SA

Types of Branches

- **Branch:** change of control flow
- **Conditional branch:** change control flow depending on outcome of comparison

`beq, bne`	
`blt, bge`	“branch if **less than**”, “branch if **greater or equal than**”
`bltu, bgeu`	“branch if **less than**” unsigned, “branch if **greater than or equal to**” unsigned

→ `beq, bne, blt, etc.` do **not** update any register in the register file

→ Branches only affect **program counter** (PC)

CC BY NC SA

Types of Branches

- **Branch:** change of control flow
- **Conditional branch:** change control flow depending on outcome of comparison

`beq, bne`	
`blt, bge`	“branch if **less than**”, “branch if **greater or equal than**”
`bltu, bgeu`	“branch if **less than**” unsigned, “branch if **greater than or equal to**” unsigned

- **Unconditional branch:** always branch

`jump (j)` `j label` == `beq x0, x0, label`

CC BY NC SA

beq

`beq rs1, rs2, label` → It marks the **address of an instruction**

```
address  instruction
-------  -----------
 1000    beq  x10, x12, mahler
 1004    sub  x5, x5, x1
 1008    srli x5, x5, 0x01
 1012    add x1, x2, x3 (label) mahler
```

→ Points to **1012**

1 line is 1 instruction, 1 line is 1 word, 1 line “includes” 4 bytes

The assembler computes the branch `offset`:

`offset` = `label_address` − `PC` =

`0x1012` − `0x1000` = 12

…and encodes it in the instruction itself as an `imm`

In machine code, the instruction no longer says `mahler` — it has a fixed signed `offset` (12)

If the **branch is taken**, we **skip** the instruction at `0x1004` and the instruction at `0x1008` and jump directly to `mahler`, i.e., `0x1012`

CC BY NC SA

beq Execution Flow

Instruction	FETCH	DECODE	EXECUTE	MEMORY	WRITEBACK
`beq x10, x12, mahler`					

```
address  instruction
-------  -----------
 1000    beq  x10, x12, mahler
 1004    sub  x5, x5, x1
 1008    srli x5, x5, 0x01
 1012    add x1, x2, x3 (label) mahler
```


CC BY NC SA

beq Execution Flow

Instruction	FETCH	DECODE	EXECUTE	MEMORY	WRITEBACK
`beq x10, x12, mahler`	✓				

Fetch `beq` at `0x1000`; compute PC + 4 = `0x1004` → "next instruction" if branch **not** taken

```
address   instruction
-------   -----------
 1000     beq  x10, x12, mahler
 1004     sub  x5, x5, x1
 1008     srli x5, x5, 0x01
 1012     add x1, x2, x3 (label) mahler
```


CC BY NC SA

beq Execution Flow

Instruction	FETCH	DECODE	EXECUTE	MEMORY	WRITEBACK
`beq x10, x12, mahler`	✓	✓			

Decode `beq`, read `x10` and `x12`; sign-extend branch `offset` = 12

= `label_address` − `PC` =

`0x1012` − `0x1000` = 12

```
address  instruction
-------  -----------
 1000    beq  x10, x12, mahler
 1004    sub  x5, x5, x1
 1008    srli x5, x5, 0x01
 1012    add x1, x2, x3 (label) mahler
```

In machine code, the instruction no longer says `mahler`
It has a fixed signed `offset` (12)

CC BY NC SA

beq Execution Flow

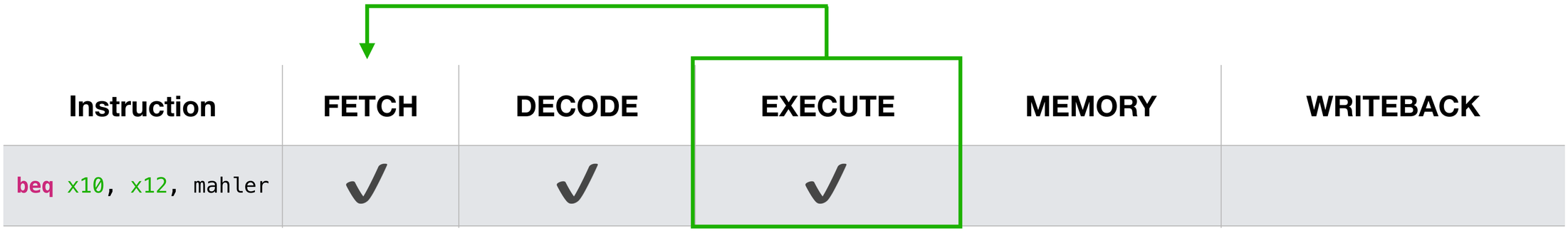
Instruction	FETCH	DECODE	EXECUTE	MEMORY	WRITEBACK
`beq x10, x12, mahler`	✓	✓	✓		

Compare `x10` == `x12` and compute `branch_target` = `PC` + `offset` = `0x1000 + 12 = 0x1012`

```
address  instruction
-------  -----------
 1000    beq  x10, x12, mahler
 1004    sub  x5, x5, x1
 1008    srli x5, x5, 0x01
 1012    add x1, x2, x3 (label) mahler
```


CC BY NC SA

beq Execution Flow



Instruction	FETCH	DECODE	EXECUTE	MEMORY	WRITEBACK
`beq x10, x12, mahler`	✓	✓	✓		

If `x10` == `x12`, we update the PC and jump directly to `mahler`

If `x10` ≠ `x12`, we go to the next instruction, i.e., `0x1004`

```
address  instruction
-------  -----------
 1000    beq  x10, x12, mahler
 1004    sub  x5, x5, x1
 1008    srli x5, x5, 0x01
 1012    add x1, x2, x3 (label) mahler
```


CC BY NC SA

beq Execution Flow

Instruction	FETCH	DECODE	EXECUTE	MEMORY	WRITEBACK
`beq x10, x12, mahler`	✓	✓	✓		

`beq` doesn't read nor write to data memory

```
address  instruction
-------  -----------
 1000    beq  x10, x12, mahler
 1004    sub  x5, x5, x1
 1008    srli x5, x5, 0x01
 1012    add x1, x2, x3 (label) mahler
```


CC BY NC SA

beq Execution Flow

Instruction	FETCH	DECODE	EXECUTE	MEMORY	WRITEBACK
`beq x10, x12, mahler`	✓	✓	✓		

`beq` doesn't write any register; only PC update occurs if branch taken

```
address  instruction
-------  -----------
 1000     beq  x10, x12, mahler
 1004     sub  x5, x5, x1
 1008     srli x5, x5, 0x01
 1012     add x1, x2, x3 (label) mahler
```


CC BY NC SA

Example *if* Statement

Let us assume the translations below, **compile** *if* block

```
f → x10, g → x11, h → x12, i → x13, j → x14
```

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

```
bne x13, x14, Exit
add x10, x11, x12
Exit: % terminate
```


CC BY NC SA

Example *if-else* Statement

Let us assume the translations below, **compile** *if-else* block

```
f → x10, g → x11, h → x12, i → x13, j → x14
```

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

```
bne x13, x14, Else
add x10, x11, x12
j Exit
Else: sub x10, x11, x12
Exit: % terminate
```


CC BY NC SA

Loops in C and Assembly

3 types of **loops**:

- `while`
- `do … while`
- `for`

Each can be rewritten as either of the other two!

```c
int main() {
    int i = 0;
    while (i < 10) {
        printf("%d\n", i);
        i++;
    }
    return 0;
}
```

```c
int main() {
    for (int i = 0; i < 10; i++) {
        printf("%d\n", i);
    }
    return 0;
}
```

```c
int main() {
    int i = 0;
    do {
        printf("%d\n", i);
        i++;
    } while (i < 10);
    return 0;
}
```


CC BY NC SA

Loops in C and Assembly

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


CC BY NC SA

Loops in C and Assembly

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

```
add  x9,  x8, x0   # x9  = &A[0]
```


CC BY NC SA

Loops in C and Assembly

```c
int A[20];
// fill A with data
int sum = 0;
for (int i = 0; i < 20; i++)
    sum += A[i];
```

```
add  x9,  x8, x0    # x9  = &A[0]
add  x10, x0, x0    # sum = 0
```


CC BY NC SA

Loops in C and Assembly

```c
int A[20];
// fill A with data
int sum = 0;
for (int i = 0; i < 20; i++)
    sum += A[i];
```

```
add  x9,  x8, x0   # x9  = &A[0]
add  x10, x0, x0   # sum = 0
add  x11, x0, x0   # i   = 0
```


CC BY NC SA

Loops in C and Assembly

```c
int A[20];
// fill A with data
int sum = 0;
for (int i = 0; i < 20; i++)
    sum += A[i];
```

```
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
```


CC BY NC SA

Loops in C and Assembly

```c
int A[20];
// fill A with data
int sum = 0;
for (int i = 0; i < 20; i++)
    sum += A[i];
```

```
    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:
```


CC BY NC SA

Loops in C and Assembly

```c
int A[20];
// fill A with data
int sum = 0;
for (int i = 0; i < 20; i++)
    sum += A[i];
```

```
    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
```

You can only compare branches using registers—register to register, **not** e.g., branch less than immediate

CC BY NC SA

Loops in C and Assembly

```c
int A[20];
// fill A with data
int sum = 0;
for (int i = 0; i < 20; i++)
    sum += A[i];
```

```
    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
```

Ok, why `bge` and not `blt`?

If that test fails, you never enter the for loop—branches usually point to the **exit**, not the body, and can **save** 1 unconditional jump

CC BY NC SA

Loops in C and Assembly

```c
int A[20];
// fill A with data
int sum = 0;
for (int i = 0; i < 20; i++)
    sum += A[i];
```

```
    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]
```


CC BY NC SA

Loops in C and Assembly

```c
int A[20];
// fill A with data
int sum = 0;
for (int i = 0; i < 20; i++)
    sum += A[i];
```

```
    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 += x12
```


CC BY NC SA

Loops in C and Assembly

```c
int A[20];
// fill A with data
int sum = 0;
for (int i = 0; i < 20; i++)
    sum += A[i];
```

```
    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 += x12
    addi x9,  x9,  4    # &A[i+1]
```


CC BY NC SA

Loops in C and Assembly

```c
int A[20];
// fill A with data
int sum = 0;
for (int i = 0; i < 20; i++)
    sum += A[i];
```

```
    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 += x12
    addi x9,  x9,  4    # &A[i+1]
    addi x11, x11, 1    # i++
```


CC BY NC SA

Loops in C and Assembly

```c
int A[20];
// fill A with data
int sum = 0;
for (int i = 0; i < 20; i++)
    sum += A[i];
```

```asm
    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 += x12
    addi x9,  x9,  4    # &A[i+1]
    addi x11, x11, 1    # i++
    j Loop
```


CC BY NC SA

Loops in C and Assembly

```c
int A[20];
// fill A with data
int sum = 0;
for (int i = 0; i < 20; i++)
    sum += A[i];
```

```
    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 += x12
    addi x9,  x9,  4    # &A[i+1]
    addi x11, x11, 1    # i++
    j Loop
Done:
```

1 **conditional** jump and 1 **unconditional** jump

CC BY NC SA