Computer Architecture

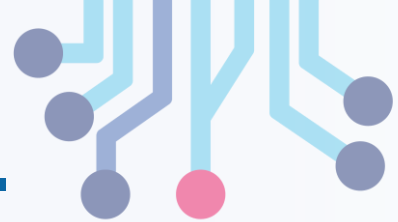
RISC-V – Program Control Instructions

Lecture 4

Agenda

- C Control Flow and goto
- Reducing C with goto
- RISC-V Control Flow

RISC-V Guiding Philosophy



- Goal of assembly:
  - Create a set of instructions such that:
    - Each instruction represents a single computation or "step"
      - Ex. add adds two registers together, addi adds a register and an immediate!

- Every C program can be broken down into instructions
  - Ex. a = b+c+d;
    - a = b+c; → add x5 x6 x7
    - a = a+d; → add x5 x5 x8

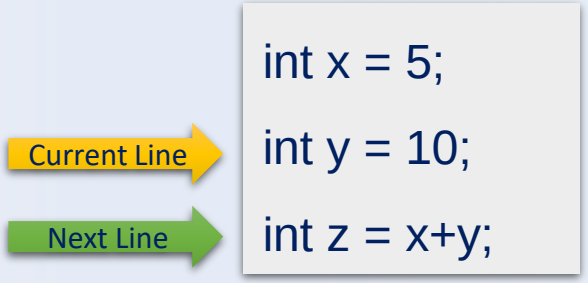
- Each instruction works in isolation without depending on context
- A program's behavior should depend only on memory, registers, and the current line being run
- RISC: There should be as few unique instructions as possible

Agenda

- **C Control Flow and `goto`**
- Reducing C with `goto`
- RISC-V Control Flow

Control Flow in C

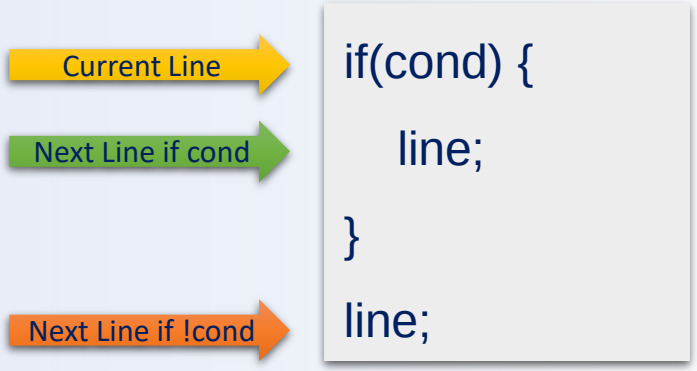
- In C, we run code one line at a time
- Most of the time, when we run a line of code, the next line that we will run is the line immediately afterwards



- A few lines make it so that the next line isn't the line immediately afterwards, but somewhere else
  - i.e., we "jump" to another line of code

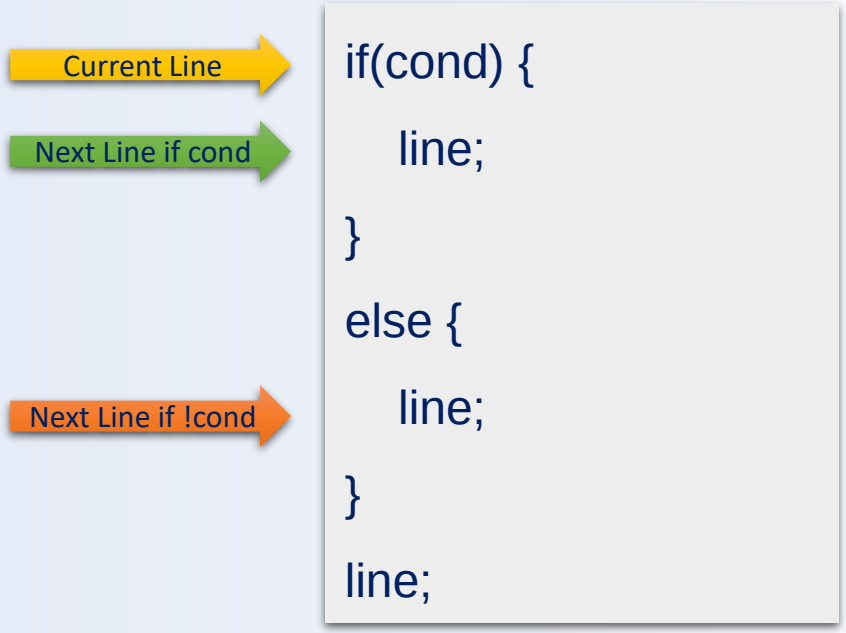
Control Flow in C

- Lines in C that affect the program flow:
  - If Statements



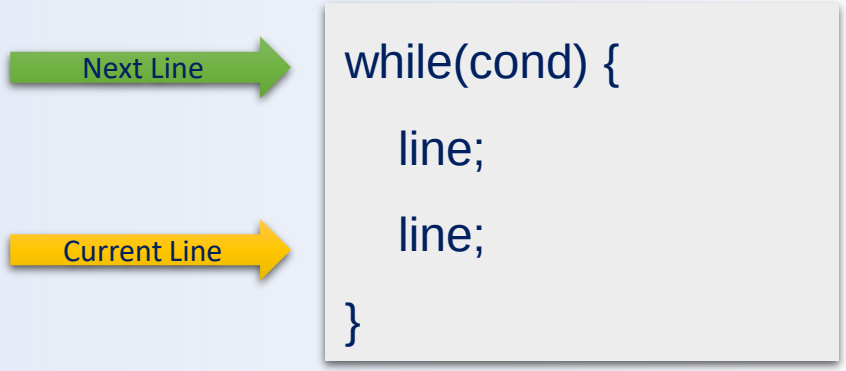
Control Flow in C

- Lines in C that affect the program flow:
  - If Statements
    - If-else statements



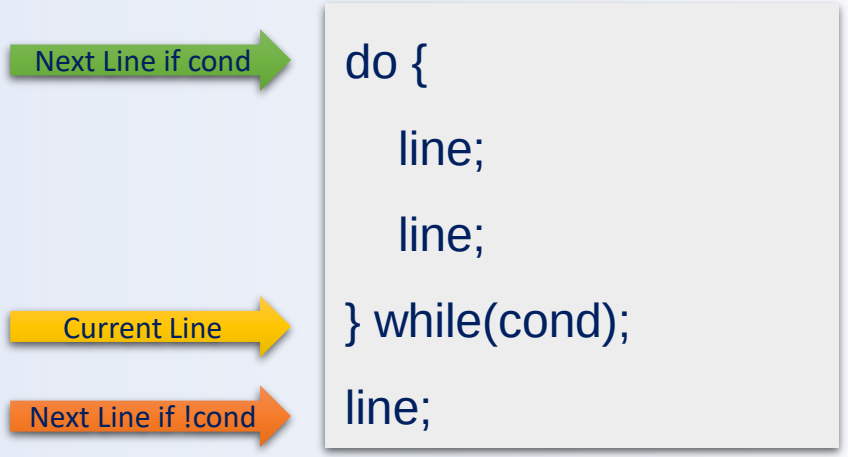
Control Flow in C

- Lines in C that affect the program flow:
  - If Statements
    - If-else statements
  - While Loops



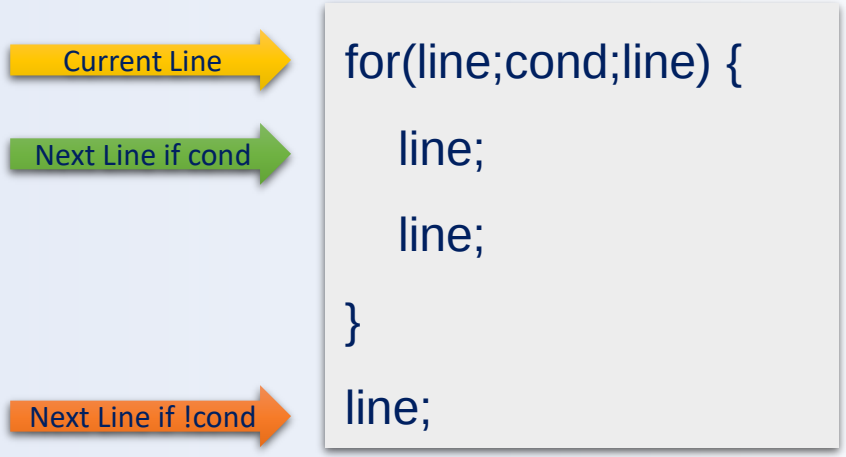
Control Flow in C

- Lines in C that affect the program flow:
  - If Statements
    - If-Else Statements
  - While Loops
    - Do-While Loops



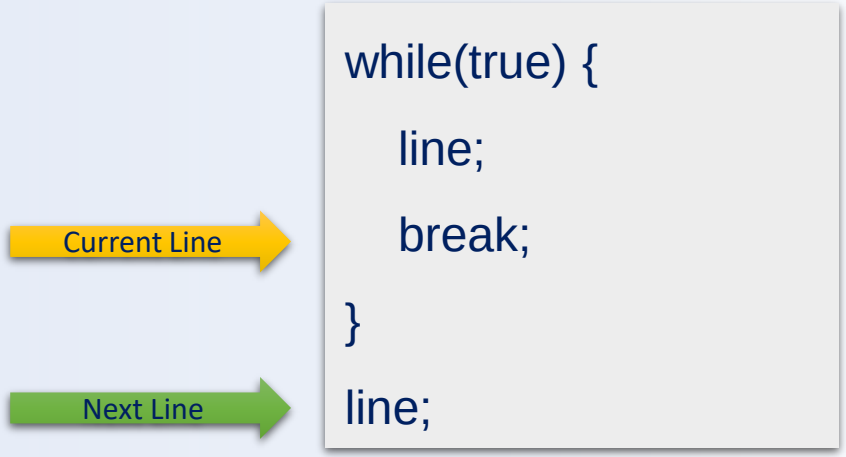
Control Flow in C

- Lines in C that affect the program flow:
  - If Statements
    - If-Else Statements
  - While Loops
    - Do-While Loops
  - For Loops



Control Flow in C

- Lines in C that affect the program flow:
  - If Statements
    - If-Else Statements
  - While Loops
    - Do-While Loops
  - For Loops
  - Break / Continue



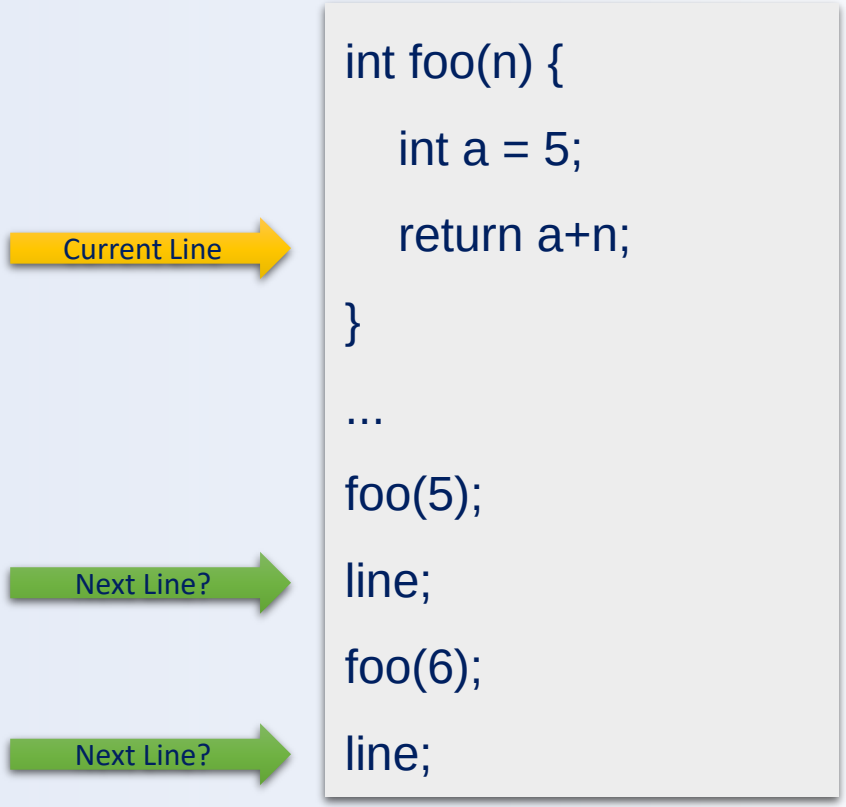
Control Flow in C

- Lines in C that affect the program flow:
  - If Statements
    - If-Else Statements
  - While Loops
    - Do-While Loops
  - For Loops
  - Break / Continue
  - Function Calls


```
int foo(n) {
    int a = 5;
    return a+n;
}
...
line;
foo(5);
line;
```

Control Flow in C

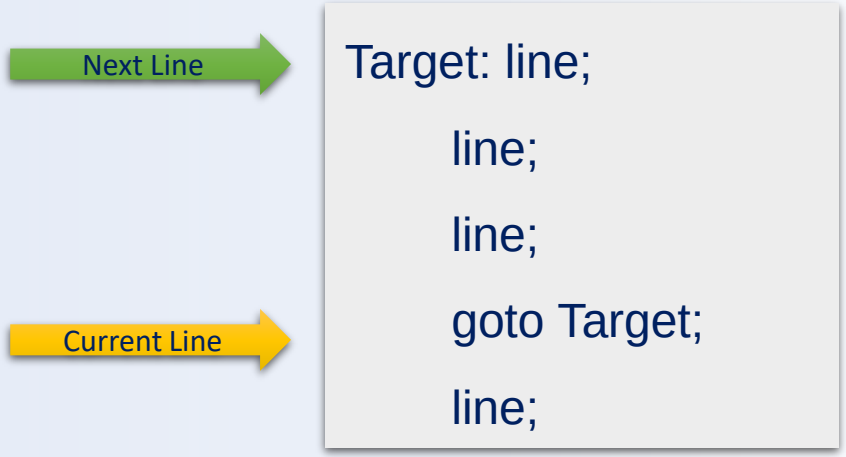
- Lines in C that affect the program flow:
  - If Statements
    - If-Else Statements
  - While Loops
    - Do-While Loops
  - For Loops
  - Break / Continue
  - Function Calls
    - Both call and return!
    - Return line depends on which line called foo.



Control Flow in C

- Lines in C that affect the program flow:
  - goto statement

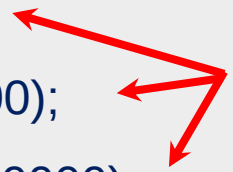
  - A label is an identifier to a particular line of code
    - Doesn't count as a line of code itself; merely "points out" a particular line
  - Each label must have a unique name (like variable names)



  - The goto statement changes the next line to be run to the labelled line
    - The label can be either before or after the goto statement.

`goto` Example: Handling Mallocs

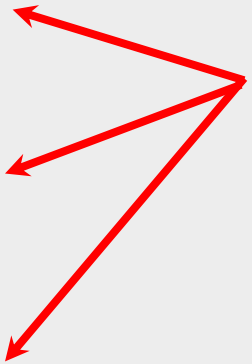
```
int* a = malloc(sizeof(int)*1000);
int* b = malloc(sizeof(int)*1000000);
int* c = malloc(sizeof(int)*1000000000);
FILE* d = fopen(filename);
```



Bad code: malloc can fail (returning NULL), and we should catch that before it causes a segfault!

`goto` Example: Handling Mallocs

```
int* a = malloc(sizeof(int)*1000);
        if(a == NULL) allocation_failed();
int* b = malloc(sizeof(int)*1000000);
        if(b == NULL) allocation_failed();
int* c = malloc(sizeof(int)*1000000000);
        if(c == NULL) allocation_failed();
FILE* d = fopen(filename);
        if(d == NULL) allocation_failed();
```



Bad code: leaks memory since **a** gets allocated but never freed.

`goto` Example: Handling Mallocs

```
int* a = malloc(sizeof(int)*1000);
                if(a == NULL) allocation_failed();

int* b = malloc(sizeof(int)*1000000);
        if(b == NULL) {
            free(a);
            allocation_failed();
        }

int* c = malloc(sizeof(int)*1000000000);
        if(c == NULL) {
            free(b);
            free(a);
            allocation_failed();
        }

FILE* d = fopen(filename);
        if(d == NULL) {
            free(c);
            free(b);
            free(a);
            allocation_failed();
        }
```

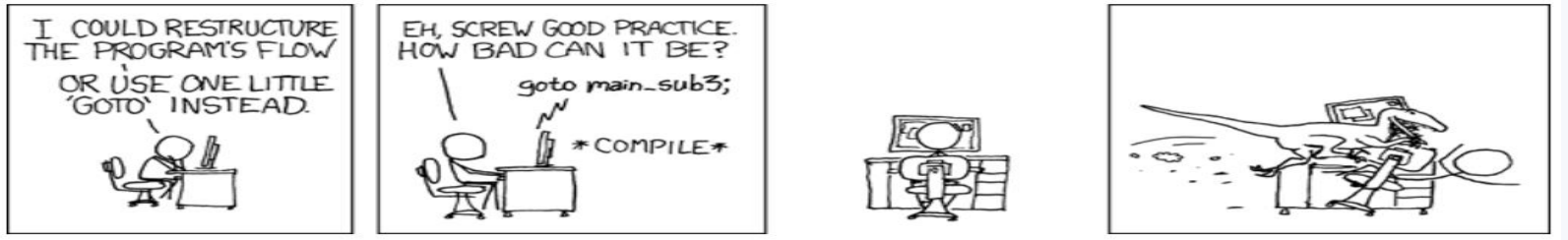
`goto` Example: Handling Mallocs

```
int* a = malloc(sizeof(int)*1000);
            if(a == NULL) goto ErrorA;
int* b = malloc(sizeof(int)*1000000);
            if(b == NULL) goto ErrorB;
int* c = malloc(sizeof(int)*1000000000);
            if(c == NULL) goto ErrorC;
FILE* d = fopen(filename);
      if(d == NULL) {
            free(c);
      ErrorC:   free(b);
      ErrorB:   free(a);
      ErrorA:   allocation_failed();
      }
```

NEVER USE goto!!!!

- Go to has a tendency to create completely illegible code
  - Generally considered bad practice, except in very specific situations
    - Error handling
    - Jumping out of nested loops
      - Even with the above, there are other approaches that don't use go to

- Nevertheless, goto is useful in that we can create any other control flow statements with just goto and conditional goto statements



Agenda

- C Control Flow and goto
- **Reducing C with goto**
- RISC-V Control Flow

Reducing C with goto: Break

```
while(true) {
        line;
        break;
}
line;
```

Reducing C with goto: Break

```
while(true) {
        line;
        goto afterWhile;
}
afterWhile: line;
```

Reducing C with goto: IF

```
if(cond) {
        line;
        line;
}
else {
    line;
    line;
}
line;
```

Reducing C with goto: IF

```
if(cond) goto IfCase;
goto ElseCase;
IfCase:
        line;
        line;
goto AfterIf;
ElseCase:
        line;
        line;
AfterIf: line;
```

Reducing C with goto: If without an Else

```
if(!cond) goto AfterIf;
        line;
        line;
AfterIf: line;
```

Reducing C with goto: Do-While

```
do {
        line;
        line;
} while(cond)
line;
```

Reducing C with goto: Do-While

```
Loop:
        line;
        line;
if(cond) goto Loop;
line;
```

Reducing C with goto: While

```
while(cond) {
        line;
        line;
}
line;
```

Reducing C with goto: While

```
Loop: if(!cond) goto AfterLoop;
        line;
        line;
goto Loop;
AfterLoop: line;
```

Reducing C with goto: For

```
for(startline; cond; incline) {
        line;
        line;
}
line;
```

Reducing C with goto: For implemented as While!

```
startline;
while(cond) {
        line;
        line;
        incline;
}
line;
```

Reducing C with goto: For implemented as While!

```
startline;
Loop: if(cond) goto AfterLoop
        line;
        line;
        incline;
goto Loop
AfterLoop: line;
```

Agenda

- C Control Flow and goto
- Reducing C with goto
- **RISC-V Control Flow**

RISC-V Control Flow Operations

- Like in C, RISC-V allows you to write labels to signify particular lines of code

- RISC-V has instructions for both conditional and unconditional jumps:
  - Branch instructions:
    - General format: bxx rs1 rs2 Label
    - Jumps to the specified Label if the condition is met
    - If the condition is not met, just moves to the next line

  - j Label
    - Jumps to the specified label
  - Technically a pseudoinstruction;

Conditional Branch Instructions

- Format
  - *opcode reg1, reg2, label*
  - Test for relation between reg1 and reg2
    - If true, execute next instruction from target label at (PC+offset) address
    - Branches are PC-relative!
      - Machine language encodes how many bytes away your target instruction is!
  - Example
    - beq x5, x6, label
      - Branch if equal
      - if (x5 == x6) go to label

Conditional Branch Instructions

- List of Branch Instructions
  - beq → branch if equal
  - bne → branch if not equal
  - blt → branch if less than (signed)
  - bge → branch if greater or equal (signed)
  - bltu → branch if less (unsigned)
  - bgeu → branch if greater or equal (unsigned)

- Note that bgt, bgtu, ble, and bleu are pseudoinstructions
  - Can make them by reversing inputs of existing instructions

Unconditional Branch Instructions

- Format
  - *opcode reg1, label*
  - Save return address (address of next instruction) in x1 (usually used for return address) and go to target at (PC+offset)!

  - Examples
    - jal x1, target
      - jump and link
      - x1 = PC+4 (address of next instruction), go to target
      - This instruction is used for calling a procedure!

    - jalr x1, 0(x5)
      - jump and link register
      - x1 = PC+4, go to x5+0!
      - Used for procedure return or indirect procedure call
    - x1 is also named as ra as it stores the return address!

Unconditional Branch Instructions

- call printf is a pseudoinstruction, what is its RISC-V equivalent?
  - jal ra, printf
  - Store PC+4 (address of next instruction) in ra, jump to printf
  - It may also translate to (depending on target range)
    - auipc & jalr

- ret is a pseudoinstruction, what is its RISC-V equivalent?
  - jalr x0, 0(ra)
  - ra already has a return address!

RISC-V Control Flow Operations: Example

```
int a = 0;
for(int i = 0; i < 10; i++) {
    if(i == 7) {
        break;
    }
    a = a + i;
}
a = a + 50;
```

RISC-V Control Flow Operations: Example

```
int a = 0;
for(int i = 0; i < 10; i++) {
        if(i == 7) goto End;
        a = a + i;
}
End: a = a + 50;
```

RISC-V Control Flow Operations: Example

```
int a = 0;
int i = 0;
Loop: if(i >= 10) goto End;
    if(i == 7) goto End;
    a = a + i;
    i = i + 1;
    goto Loop;
End: a = a + 50;
```

RISC-V Control Flow Operations: Example

```
int a = 0;
int i = 0;
Loop:
    int j = 10;
    if(i >= j) goto End;
    j = 7;
    if(i == j) goto End;
    a = a + i;
    i = i + 1;
    goto Loop;
End: a = a + 50;
```

RISC-V Control Flow Operations: Example

```
li x10 0                    # int a = 0;
li x5 0                     # int i = 0;
Loop:
    li x6 10                # int j = 10;
    bge x5 x6 End           # if(i >= j) goto End;
    li x6 7                 # j = 7;
    beq x5 x6 End           # if(i == j) goto End;
    add x10 x10 x5          # a = a + i;
    addi x5 x5 1            # i = i + 1;
    j Loop                  # goto Loop; (pseudo for jal)
End:  addi x10 x10 50       # a = a + 50;
```

Exercise …

- Convert following C-language statement into RISC-V assembly
  - if (i == j) f = g + h; else f = g − h;
    - Assume {x19, x20, x21, x22, x23}={f, g, h, i, j}
    - Use x9 as the temporary register!
    - Hint: Use labels to branch (jump) to a target!
- Solution

```
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            # if 0 == 0, go to Exit
Else: sub x19, x20, x21     # f = g − h (skipped if i = j)
Exit:
```

Another Exercise …

- Convert the following loop to RISC-V Assembly

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

  - Assume {x22, x24}={i, k} (word-sized)
  - x25 = base address of save!
  - Use x10 and x9 as temporary registers!
  - Hint: use shift left for multiplication (to get offset)!

- Solution

```
Loop:
        slli x10, x22, 2        # Temp reg x10 = i * 4
        add x10, x10, x25       # x10 = address of save[i]
        lw x9, 0(x10)           # Temp reg x9 = save[i]
        bne x9, x24, Exit       # go to Exit if save[i] ≠ k
        addi x22, x22, 1        # i = i + 1
        beq x0, x0, Loop        # go to Loop
Exit:
```

Summary: RISC-V so far …

- Arithmetic & Logical
  - add
  - sub
  - and
  - or
  - xor
  - sll
  - srl
  - sra
- Immediate
  - addi
  - andi
  - ori
  - xori
  - slli
  - srli
  - srai
  - lui
  - auipc
- Loads & Stores
  - lw
  - lb
  - lbu
  - lh
  - lhu
  - sw
  - sb
  - sh
- Branches & Jumps
  - beq
  - bne
  - bge
  - blt
  - bgeu
  - bltu
  - jal
  - jalr

Thank You

NCDC
NUST
CHIP
DESIGN
CENTRE