# Computer Architecture

IN BA3 - Paolo IENNE

Notes by Ali EL AZDI

September 11, 2024

## Introduction

This document is designed to offer a LaTeX-styled overview of the Computer Architecture course, emphasizing brevity and clarity. Should there be any inaccuracies or areas for improvement, please reach out at ali.elazdi@epfl.ch for corrections. For the latest version, check my GitHub repository.
`https://github.com/elazdi-al/comparch/blob/main/main.pdf`

# Contents

# Chapter 1

# Part I(a) - ISA Reminder, Assembly Language, Compiler - W 1.1

**hum...welcome back**
*In the first part of the course, professor introduced (for motivational purposes) how computer architecture, specifically processors, have become essential to our lives, and how the field is growing exponentially. (didn't think it was essential to mention here...)*

## 1.1 From High Level Languages to Assembly Language

### 1.1.1 High Level Languages

*When talking about programming we usually think of programs that look like this...*

```c
int data = 0x00123456;
int result = 0;
int mask = 1;
int count = 0;
int temp = 0;
int limit = 32;
do {
    temp = data & mask;
    result = result + temp;
    data = data >> 1;
    count = count + 1;
} while (count != limit);
```

| name | value |
|---|---|
| data | 0x00123456 |
| result | 0 |
| mask | 1 |
| count | ... |
| temp | |
| limit | |
| ... | |
| my_float | 3.141529 |
| a_string | Hello world! |

### 1.1.2 Assembly Language

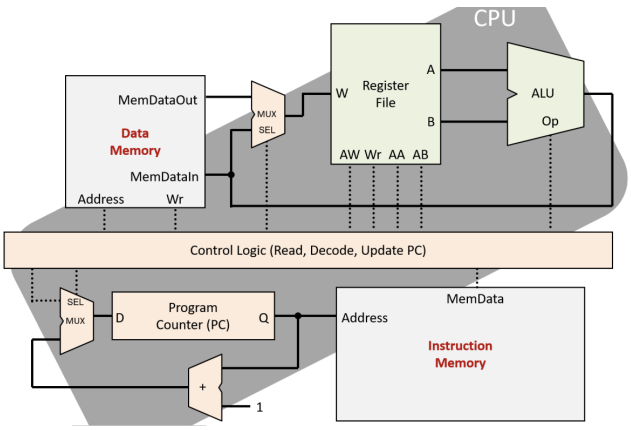
We use this code because it enables us to build a *Finite State Machine*, which isn't feasible with C code. This language provides a more rigid format with a sequence of numbered instructions, an *opcode*, predefined variable names, and the ability to **jump between lines**.

```
    li x1, 0x00123456
    li x2, 0
    li x3, 1
    li x4, 0
    li x5, 0
    li x6, 32
loop: and x5, x1, x3
    add x2, x2, x5
    srli x1, x1, 1
    addi x4, x4, 1
    bne x4, x6, loop
```

## 1.2 Processors

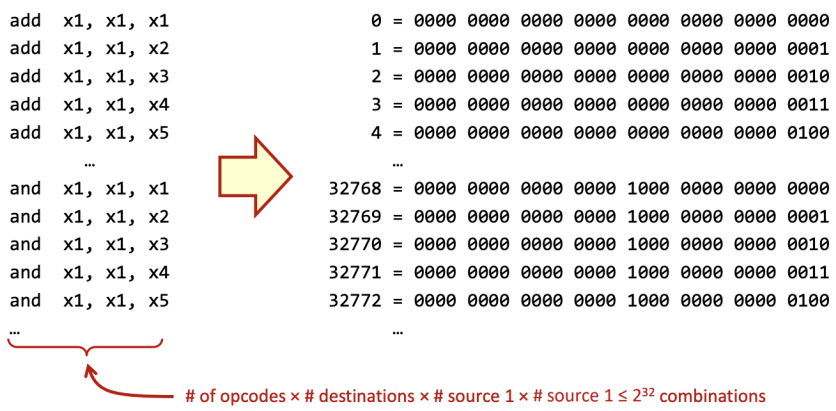
**Remember, a processor can be decomposed into five components:**

- **ALU (Arithmetic and Logic Unit)**: Performs arithmetic and logical operations.
- **Register File**: Stores data temporarily for quick access during processing.
- **Memory**: Holds data and instructions needed by the processor.
- **Control Logic**: Directs the operation of the processor by coordinating the other components.
- **PC (Program Counter)**: Keeps track of the address of the next instruction to be executed.
- **Instruction Memory**: Stores the program instructions that the processor will execute.

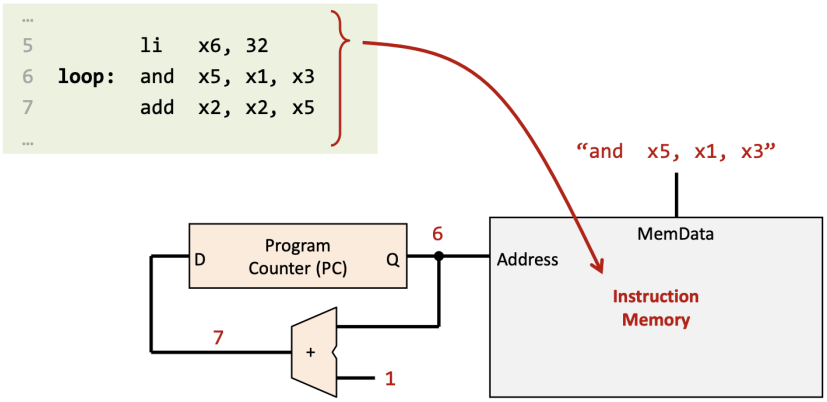



We may distinguish three types of general operations made by the processor:

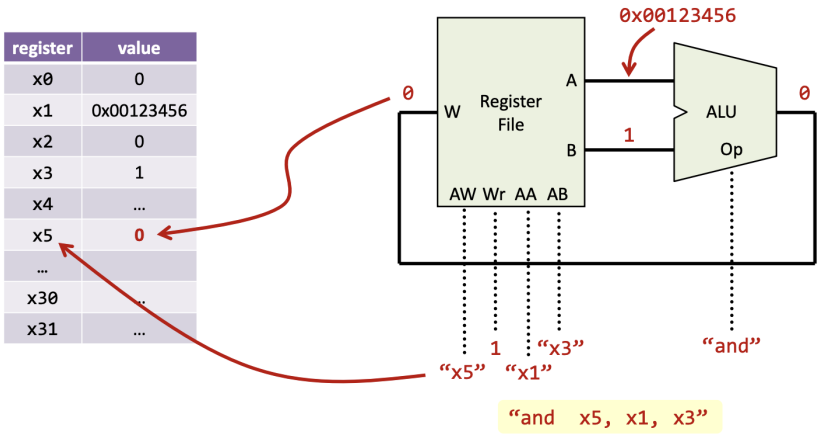
**Encoding**



### Fetching



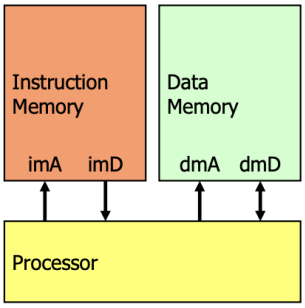

### Executing



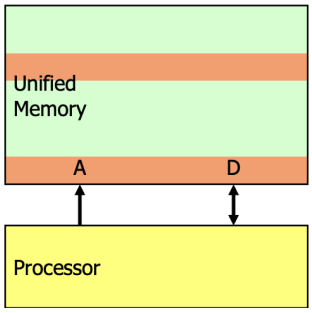

## 1.3 Joint or Disjoint Program and Data Memories

*There are two main types of architectures one called the Harvard Architecture (Where the data and the memory are seperate) and pne called Unified Architecture (where data is shared with the program memory)*

Harvard Architecture




Unified Architecture



## 1.4 The Encoding problem

*We may ask ourselves how we encode assembly written instructions into actual 0s and 1s.*

### 1.4.1 The Stupid Solution

*Now, the professor throws out the "stupid idea"(his words) of just counting all possible instructions, assigning a number to each one, and writing the numbers in binary. The problem with such a method is that the number of instructions could grow exponentially, requiring an unmanageable number of bits to represent each one, leading to inefficiency.*

```
add  x1, x1, x1          0 = 0000 0000 0000 0000 0000 0000 0000 0000
add  x1, x1, x2          1 = 0000 0000 0000 0000 0000 0000 0000 0001
add  x1, x1, x3          2 = 0000 0000 0000 0000 0000 0000 0000 0010
add  x1, x1, x4          3 = 0000 0000 0000 0000 0000 0000 0000 0011
add  x1, x1, x5          4 = 0000 0000 0000 0000 0000 0000 0000 0100
      …                     …
and  x1, x1, x1      32768 = 0000 0000 0000 0000 1000 0000 0000 0000
and  x1, x1, x2      32769 = 0000 0000 0000 0000 1000 0000 0000 0001
and  x1, x1, x3      32770 = 0000 0000 0000 0000 1000 0000 0000 0010
and  x1, x1, x4      32771 = 0000 0000 0000 0000 1000 0000 0000 0011
and  x1, x1, x5      32772 = 0000 0000 0000 0000 1000 0000 0000 0100
…                             …
```

# of opcodes × # destinations × # source 1 × # source 1 ≤ $2^{32}$ combinations

**"stupid solution"**

### 1.4.2 RISC-V Encoding (The Solution)

**Instead, the chosen solution is to use an instruction set encoding where instructions are grouped into classes, each with a fixed format optimizing both memory usage and processing speed by limiting the number of bits required to represent instructions.**

| Instruction | Pseudocode | Type | funct7 | funct3 | opcode |
|---|---|---|---|---|---|
| **Shift** | | | | | |
| sll rd,rs1,rs2 | rd ← rs1 ≪ rs2 | R | 0x00 | 0x1 | 0x33 |
| slli rd,rs1,imm | rd ← rs1 ≪ imm | I | 0x00 | 0x1 | 0x13 |
| srl rd,rs1,rs2 | rd ← rs1 $\gg_u$ rs2 | R | 0x00 | 0x5 | 0x33 |
| srli rd,rs1,imm | rd ← rs1 $\gg_u$ imm | I | 0x00 | 0x5 | 0x13 |
| sra rd,rs1,rs2 | rd ← rs1 $\gg_s$ rs2 | R | 0x20 | 0x5 | 0x33 |
| srai rd,rs1,imm | rd ← rs1 $\gg_s$ imm | I | 0x20 | 0x5 | 0x13 |
| **Arithmetic** | | | | | |
| add rd,rs1,rs2 | rd ← rs1 + rs2 | R | 0x00 | 0x0 | 0x33 |
| addi rd,rs1,imm | rd ← rs1 + sext(imm) | I | | 0x0 | 0x13 |
| sub rd,rs1,rs2 | rd ← rs1 − rs2 | R | 0x20 | 0x0 | 0x33 |
| lui rd,imm | rd ← imm | | | | |
| auipc rd,imm | rd ← pc + | | | | |
| **Logical** | | | | | |
| xor rd,rs1,rs2 | rd ← rs1 | | | | |
| xori rd,rs1,imm | rd ← rs1 | | | | |
| or rd,rs1,rs2 | rd ← rs1 | | | | |
| ori rd,rs1,imm | rd ← rs1 | | | | |
| and rd,rs1,rs2 | rd ← rs1 | | | | |
| andi rd,rs1,imm | rd ← rs1 | | | | |

**Instruction types**

| | 31 … 25 | 24 … 20 | 19 … 15 | 14 … 12 | 11 … 7 | 6 … 0 | |
|---|---|---|---|---|---|---|---|
| **R** | funct7 | rs2 | rs1 | funct3 | rd | opcode | Register-Register |
| **I** | imm[11:0] | | rs1 | funct3 | rd | opcode | Register-Immediate |
| **I** | funct7 | imm[4:0] | rs1 | funct3 | rd | opcode | Register-Immediate Shift |
| **S** | imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | opcode | Store |
| **B** | imm[12—10:5] | rs2 | rs1 | funct3 | imm[4:1—11] | opcode | Branch |
| **U** | imm[31:12] | | | | rd | opcode | Upper Immediate |
| **J** | imm[20—10:1—11—19:12] | | | | rd | opcode | Jump |

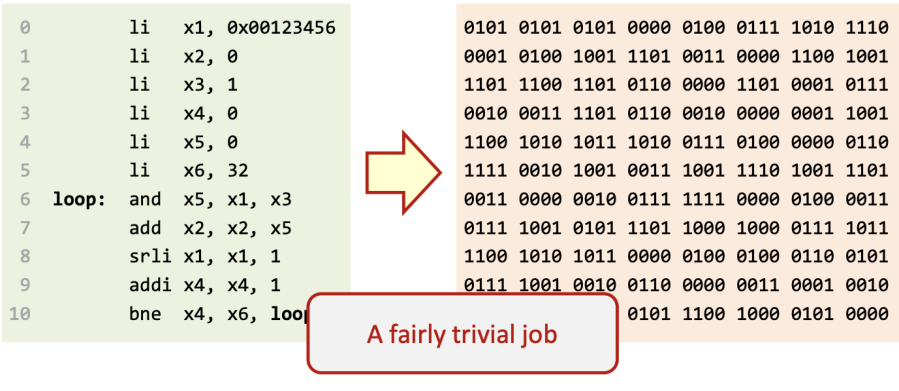
**RISC-V encoding**

### 1.4.3 Automating this process

*Now to automate the processes of decoding assembler code into machine code we use an **Assembler**, and to automate the process of decoding a higher level language to assembler we use a **Compiler**.*

#### Assembler

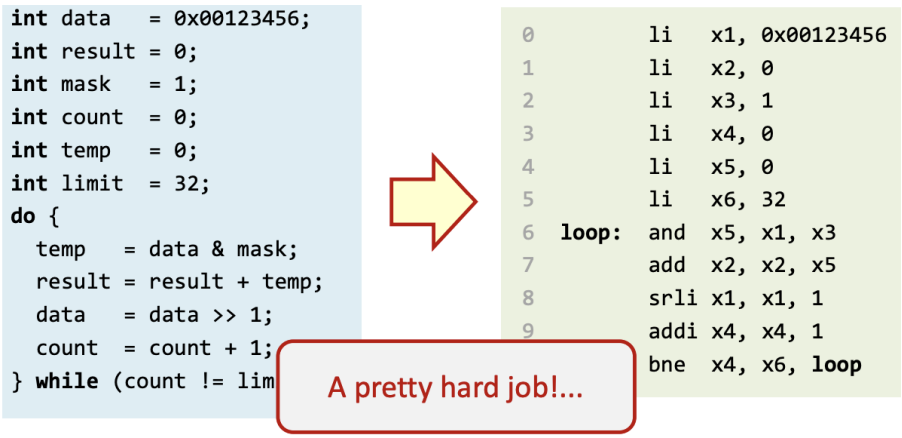
*The program that does this is called an assembler. It takes the assembly code and converts it into machine code.*




**Assembly**

#### Compiler

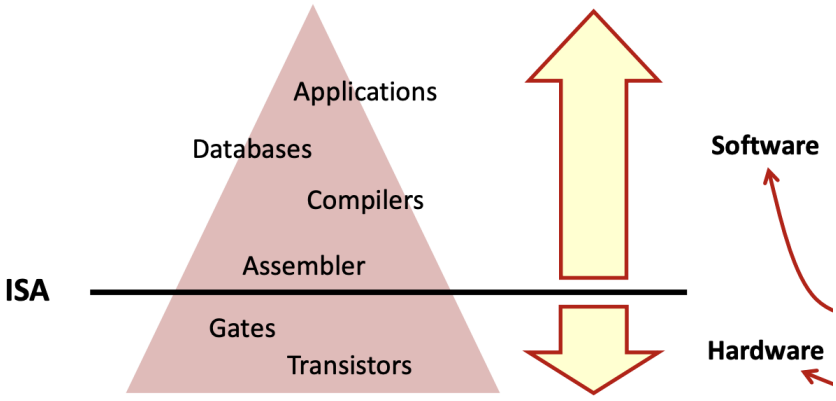
A compiler is a program that translates high-level source code written in languages like C or Java into machine code or an intermediate representation.




**Compilation**

## 1.5 ISA (Instruction Set Architecture)

*The ISA is the interface between the hardware and the software. It defines the instructions that a processor can execute, as well as the format of those instructions.*

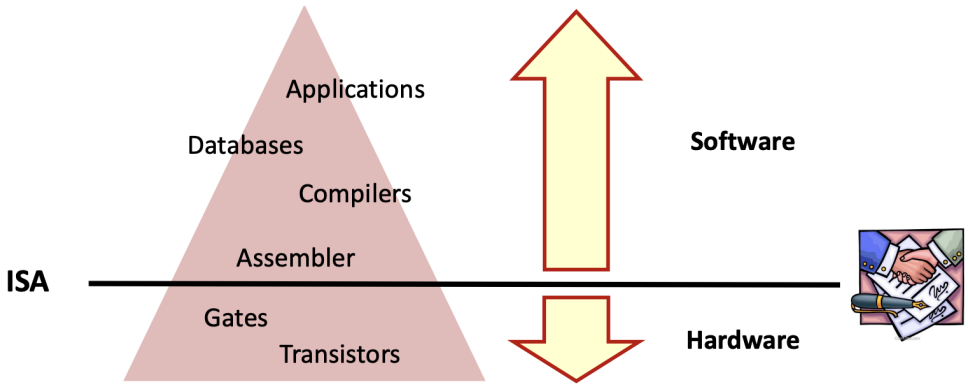

Applications
Databases
Compilers
Assembler
ISA
Gates
Transistors
Software
Hardware

# Chapter 2

# Part I(b) - ISA, Functions, and Stack - W 1.2

## 2.1 The Contract between HW and SW

*The Contract between hardware and software is **ISA** (Instruction Set Architecture), it defiens the rules the hardware and software follow to work together and commuunicate correctly;*




## 2.2 Arithmetic and Logic Instructions in RISCV

*Bellow some examples of RISCV instructions:*
**Two Operands Instructions**

```
sll  x5, x5, x9
add  x6, x5, x7
xor  x6, x6, x8
slt  x8, x6, x7
```

*Shift x5 left by x9 positions → x5*
*Add x5 and x7 → x6*
*Logic XOR bitwise x6 and x8 → x6*
*Set x8 to 1 if x6 is lower than x7, otherwise to 0*

**Arithmetic Instructions**

```
slli x5, x5, 3
addi x6, x5, 72
xori x6, x6, -1
slti x8, x6, 321
```

*Shift x5 left of 3 positions → x5*
*Add 72 to x5 → x6*
*Logic XOR bitwise x6 and 0xFFFFFFFF → x6*
*Set x8 to 1 if x6 is lower than 321, to 0 otherwise*

**Here, you may ask yourself, why are all immediates (constants) writtent on a maximum of 12bits?**

### 2.2.1 Constants must be encoded on 12 bits

*As you may see here, all instructions encode immediates on 12 bits.*

| | 31 — 25 | 24 — 20 | 19 — 15 | 14 — 12 | 11 — 7 | 6 — 0 | |
|---|---|---|---|---|---|---|---|
| **R** | funct7 | rs2 | rs1 | funct3 | rd | opcode | Register-Register |
| **I** | imm[11:0] | | rs1 | funct3 | rd | opcode | Register-Immediate |
| **I** | funct7 | imm[4:0] | rs1 | funct3 | rd | opcode | Register-Immediate Shift |
| **S** | imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | opcode | Store |
| **B** | imm[12—10:5] | rs2 | rs1 | funct3 | imm[4:1—11] | opcode | Branch |
| **U** | imm[31:12] | | | | rd | opcode | Upper Immediate |
| **J** | imm[20—10:1—11—19:12] | | | | rd | opcode | Jump |

### 2.2.2 Assembler Directives

*Assembler directives help write cleaner and more readable code. The code snippets on the left and right below are equivalent.*

```
lui    x5, 0x12345
addiu  x5, x5, 0x678
xor    x6, x6, x5
```

```
.equ   something, 0x12345678

lui    x5, %hi(something)
addiu  x5, x5, %lo(something)
xor    x6, x6, x5
```

The left-hand side code snippet shows an assembly sequence where a 32-bit constant value (`0x12345678`) is loaded into a register (`x5`). Since immediate values are 16-bit limited, this requires splitting the 32-bit value into two instructions:

- The first instruction, `lui`, loads the upper 16 bits (`0x12345`) into the register `x5`.
- The second instruction, `addiu`, adds the lower 16 bits (`0x678`) to `x5`, completing the full 32-bit value in the register.

*This approach, while functional, can become cumbersome when dealing with multiple constants, making the code less readable and harder to maintain.*
The right-hand side shows the same functionality but makes use of assembler directives, specifically the `.equ` directive to define a label (`something`) for the constant `0x12345678`. Using the `%hi()` and `%lo()` pseudo-instructions, the assembler automatically splits the constant into its upper and lower parts:

- The `%hi(something)` loads the upper 16 bits into `x5`.
- The `%lo(something)` adds the lower 16 bits to `x5`.

This method enhances code clarity and maintainability, especially when working with multiple constants, by using human-readable labels instead of raw numeric values. The assembler handles the details of splitting the 32-bit constant into its upper and lower parts.

| Directive | Effect |
|---|---|
| `.text` | Store subsequent instructions at next available address in *text* segment |
| `.data` | Store subsequent items at next available address in *data* segment |
| `.asciiz` | Store string followed by null-terminator in `.data` segment |
| `.byte` | Store listed values as 8-bit bytes |
| `.word` | Store listed values as 32-bit words |
| `.equ` | Define constants |

### 2.2.3 The x0 Register

*The x0 register is hardwired to 0 and cannot be changed. Any attempt to write into x0 will have no effect.*

`Why is this useful?`

One common application is in introducing wait delays during program execution. By leveraging the fixed nature of `x0`, it simplifies certain instructions that require an immediate zero value.

## 2.3 PseudoInstructions

*PseudoInstructions simplify commands involving the x0 register by creating easier-to-use alternatives.*

| Pseudoinstruction | Base Instruction(s) | Meaning |
|---|---|---|
| `nop` | `addi x0, x0, 0` | No operation |
| `li rd, immediate` | Myriad sequences | Load immediate |
| `mv rd, rs` | Myriad sequences | Copy register |
| `not rd, rs` | `xori rd, rs, -1` | One's complement |
| `neg rd, rs` | `sub rd, x0, rs` | Two's complement |
| `seqz rd, rs` | `sltiu rd, rs, 1` | Set if = zero |
| `snez rd, rs` | `sltu rd, x0, rs` | Set if ≠ zero |
| `sltz rd, rs` | `slt rd, rs, x0` | Set if ¡ zero |
| `sgtz rd, rs` | `slt rd, x0, rs` | Set if ¿ zero |

The term *myriad sequences* refers to a series of instructions that together achieve the functionality of a single pseudoinstruction, such as using `lui` and `addi` to implement `li rd, immediate`.
**According to the professor li should be called `mvi` (as move immediate).**

### 2.3.1 Control flow instructions

*Control flow instructions are used to change the order of execution of instructions are a kind of pseudo-instructions.*

```
    li x1, 0x00123456
    li x2, 0
    li x3, 1
    li x4, 0
    li x5, 0
    li x6, 32
loop: and x5, x1, x3
    add x2, x2, x5
    srli x1, x1, 1
    addi x4, x4, 1
    bne x4, x6, loop
```

### 2.3.2 If-Then-Else

```
if (x5 == 72) {
    x6 = x6 + 1;
    } else {
    x6 = x6 - 1;
}
```

```
.text
    li x7, 72
    beq x5, x7, then_clause
else_clause:
    addi x6, x6, -1
    j end_if
then_clause:
    addi x6, x6, 1
end_if:
```

*As seen here, beqi does not exist in RISCV, instead we use* `beq` *and* `li` *to achieve the same result.*

### 2.3.3 Jumps and Branches

A common but not universal distinction exists between *jumps* and *branches*. In RISC-V (inherited from MIPS and used by SPARC, Alpha, etc.), jumps refer to unconditional control transfer instructions, while branches refer to conditional control transfer instructions. However, not all architectures follow this convention. For instance, in x86, all control transfer instructions are considered jumps, such as `JMP`, `JZ`, `JC`, and `JNO`.

### 2.3.4 Comparaisions

*The processor implements only $<$ and $>$, and the assembler "creates" $\leq$ and $\geq$.*

| Pseudoinstruction | Base Instruction(s) | Meaning |
|---|---|---|
| `beqz rs, offset` | `beq rs, x0, offset` | Branch if $=$ zero |
| `bnez rs, offset` | `bne rs, x0, offset` | Branch if $\neq$ zero |
| `blez rs, offset` | `bge x0, rs, offset` | Branch if $\leq$ zero |
| `bgez rs, offset` | `bge rs, x0, offset` | Branch if $\geq$ zero |
| `bltz rs, offset` | `blt rs, x0, offset` | Branch if $<$ zero |
| `bgtz rs, offset` | `blt x0, rs, offset` | Branch if $>$ zero |
| `bgt rs, rt, offset` | `blt rt, rs, offset` | Branch if $>$ |
| `ble rs, rt, offset` | `bge rt, rs, offset` | Branch if $\leq$ |
| `bgtu rs, rt, offset` | `bltu rt, rs, offset` | Branch if $>$, unsigned |
| `bleu rs, rt, offset` | `bgeu rt, rs, offset` | Branch if $\leq$, unsigned |

### 2.3.5 Do-While

*Do-while loops look like this (we obviously use control flow instructions here).*

```
do {
    x5 = x5 >> 1;
    x6 = x6 + 1;
} while (x5 != 0);
```

```
.text
loop:
    srli x5, x5, 1
    addi x6, x6, 1
    bnez x5, loop
```

## 2.4 Functions

*In higher-level programming languages, functions (routines, subroutines, procedures, methods, etc.) are used to encapsulate code and make it reusable.*

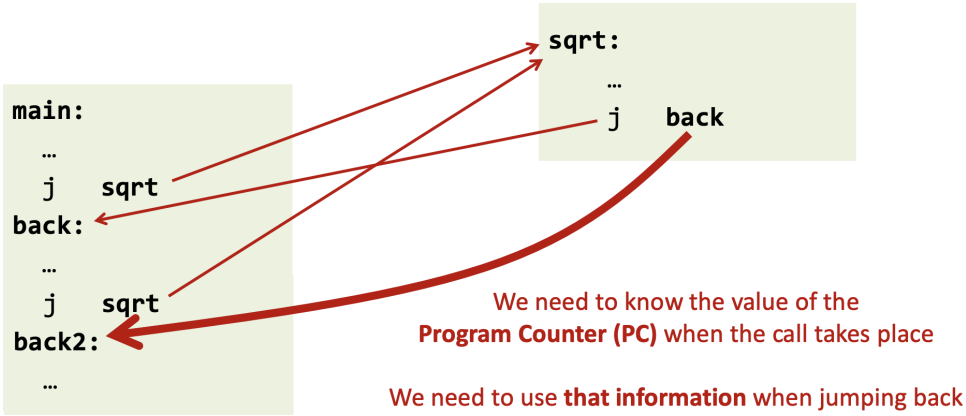
**Calling a function involves these steps:**

1. Place arguments where the called function can access them.
2. Jump to the function.
3. Acquire storage resources the function needs.
4. Perform the desired task of the function.
5. Communicate the result value back to the calling program.
6. Release any local storage resources.
7. Return control to the calling program.

### 2.4.1 Jump to the Function/Retun control to the calling program

#### The too simple not working approach

A simple (not working) approach for creating functions would be to do this:




*With this approach the function doesn't know where to return to after being called (back2 or back)*
**For the next part, remember, the Program Counter is distinct from general-purpose registers. It is dedicated to managing the flow of instruction execution, while general registers are used for data manipulation.**

#### The Good Approach

*The right approach involves using the Jump and Link instruction* `jal`*, here loading PC + 4 (remember 4 bytes per Instruction) into x1 as a way to come back from the function.*

```
main:
    ...
    jal x1, sqrt
    ...
    ...
    jal x1, sqrt
```

```
sqrt:
    ...
    ...
    jr x1
```

*Both times x1 was used to store the return adress, and there is a reason for that (Register Conventions Sections).*

### 2.4.2 Jump Instructions

*There are only two core real jump instructions in RISCV,* `jal` *(jump and link) and* `jalr` *(jump and link register), the rest are pseudo instructions using them.*

| Pseudoinstr. | Base Instruction(s) | Meaning |
|---|---|---|
| `j offset` | `jal x0, offset` | Jump |
| `jal offset` | `jal x1, offset` | Jump and link |
| `jr rs` | `jalr x0, 0(rs)` | Jump register |
| `jalr rs` | `jalr x1, 0(rs)` | Jump and link register |
| `ret` | `jalr x0, 0(x1)` | Return from subroutine |

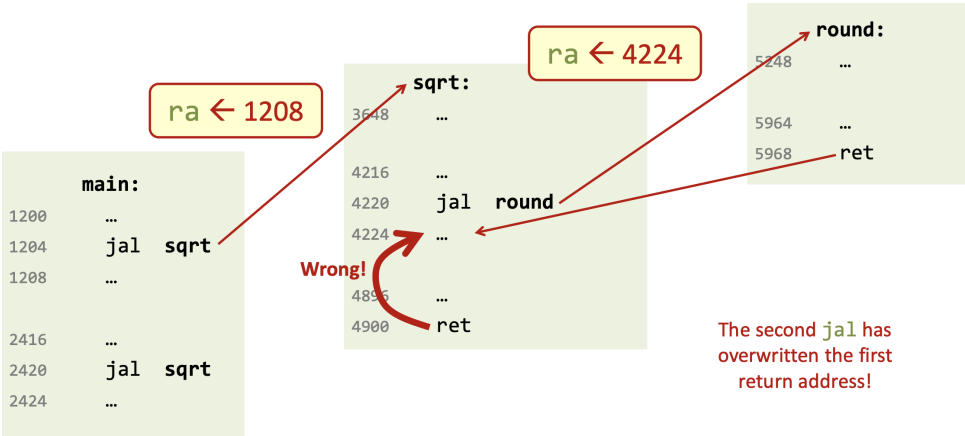
### 2.4.3 Register Conventions

*Register conventions are rules that dictate how registers are used in a program, here are the ones we've seen for now*

| Register | Mnemonic | Description |
|---|---|---|
| x0 | zero | Hard-wired zero |
| x1 | ra | Return Address |

### 2.4.4 Back to the good (not so good) approach

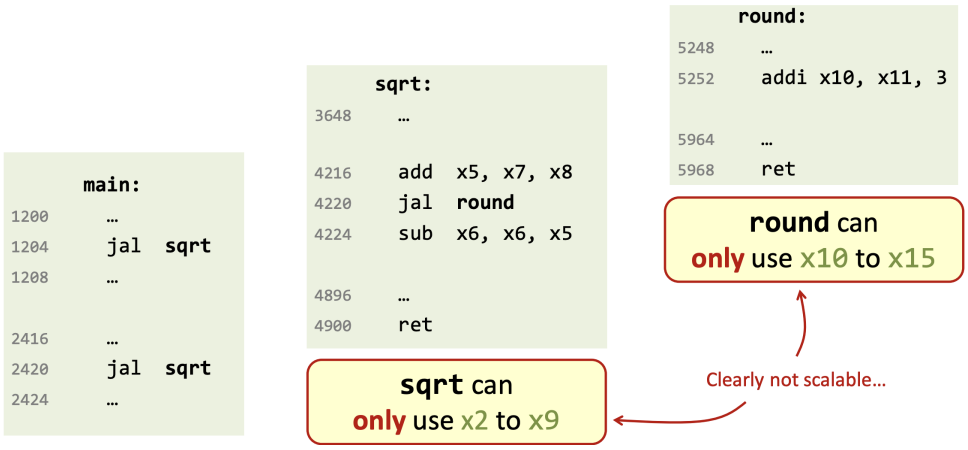
*There's still a problem with the previous approach, say for example you want to call a function from another function.*




*Here the allocated space for the return address is overwritten by the second function call, and the first function can't return to the right place.*

### 2.4.5 One simple solution (still not good)

*One solution would be to say that a range of registers are used for certain functions and that they can't be used by other functions.*




**The problem here is that it's still not very scalable.**

### 2.4.6 Acquire storage resources the function needs (still not it)

One simple solution to our problem would be to allocate memory for the function at in the data section of the program.

```
.data
sqrt_save_ra: .word 0
sqrt_save_x5: .word 0
```

```
.text
sqrt:
...
add x5, x7, x8
sw ra, sqrt_save_ra
sw x5, sqrt_save_x5
jal round
lw ra, sqrt_save_ra
lw x5, sqrt_save_x5
sub x6, x6, x5
...
ret
```

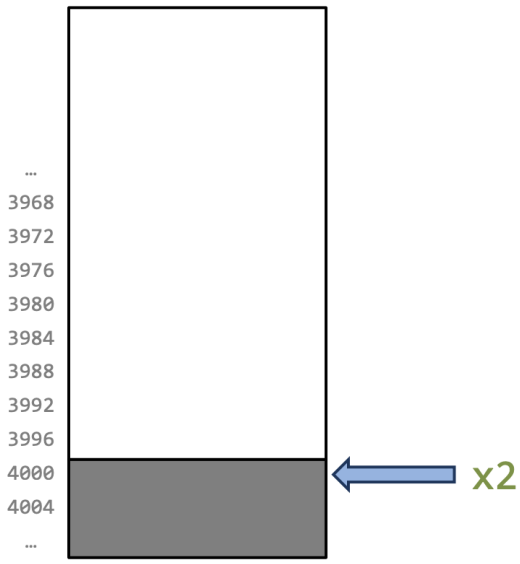
**Problem: Recursive Functions**

*The problem here is that the return address is overwritten by the recursive call.*

```
.data
    find_child_save_ra: .word 0
.text
    find_child:
    ...
    sw ra, find_child_save_ra
    jal find_child
    lw ra, find_child_save_ra
    ...
    ret
```

### 2.4.7 The Stack

*The Solution to our Problem is this, the Stack.*
**The Stack is a region of memory that grows and shrinks as needed.**
We may use a register (e.g `x2`) to point to the first used word after the end of the used region.



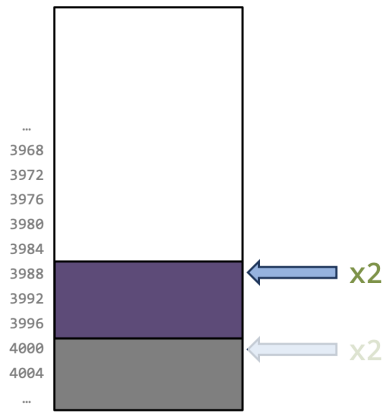

**Dynamic Memory Allocation**

The Stack, contrary to the Data Section, is dynamic and can be used to allocate memory when needed. This means that during program execution, variables or temporary data can be stored in the stack, which grows or shrinks depending on the operations performed.

The `stack pointer`, typically register x2, is used to manage the allocation and deallocation of memory.

*In this instruction, for example, we allocate 12 bytes in the stack. We achieve this by decrementing the stack pointer (x2) by 12. This ensures that the new memory space is available for temporary storage.*

```
addi x2, x2, -12
```

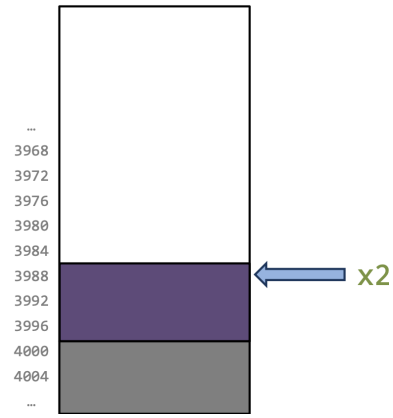



### Retrieving Data from the Stack

Once memory has been allocated on the stack, we can store or retrieve data from it. In this case, we are retrieving data that was previously saved in the stack. The lw (load word) instruction is used to load the values stored at different offsets in the stack.

*In this case, we retrieve three different values from the stack using the lw instruction, which loads a 4-byte value into the specified registers (ra, x5, and x6). The offsets (0, 4, and 8) refer to different positions in the 12 bytes we allocated earlier.*

```
lw ra, 0(x2)
lw x5, 4(x2)
lw x6, 8(x2)
```

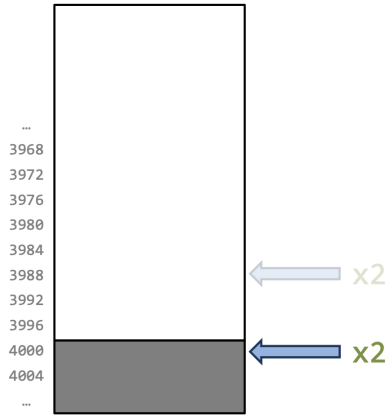



### Memory Deallocation

After the data has been used or is no longer needed, it is good practice to deallocate the memory to ensure proper management of the stack. We deallocate memory by adjusting the stack pointer (x2) back to its original position.

*In this instruction, we restore the stack to its previous state by adding 12 back to the stack pointer (x2).*
*This effectively "frees" the 12 bytes of memory we had allocated earlier.*

```
addi x2, x2, 12
```



### The Stack Pointer

*The Stack Pointer is a register that points to the top of the stack, by convention it corresponds to the x2 register*

| Register | ABI Name | Description | Preserved across call? |
|---|---|---|---|
| x2 | sp | Stack pointer | Yes |

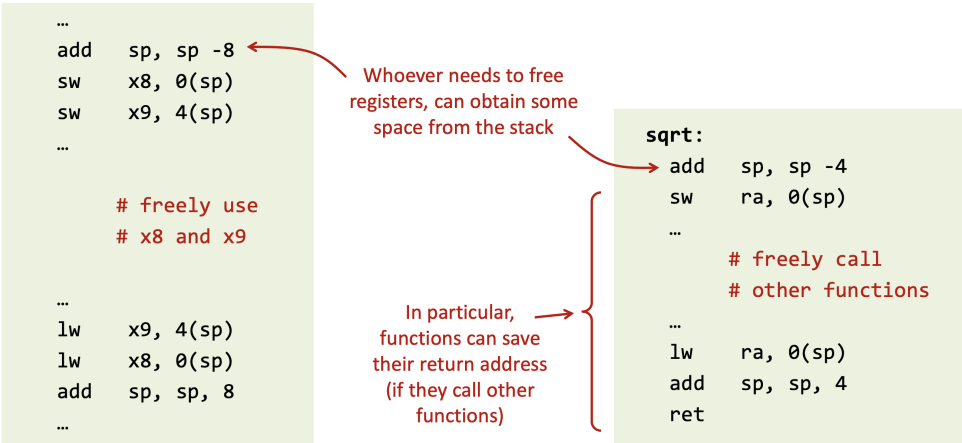
*Other architectures have special instructions to place stuff on the stack (push) and to retrieve it (pop)*

PUSH AX

```
add sp, sp, -4
sw x5, 0(sp)
```

## 2.4.8 Spilling Registers to Memory

*Spilling registers to memory involves saving register values to the stack when more registers are needed or to prevent overwriting important data, allowing the registers to be reused. This technique is also used in function calls to save the return address, ensuring the program can correctly return control after the function finishes.*

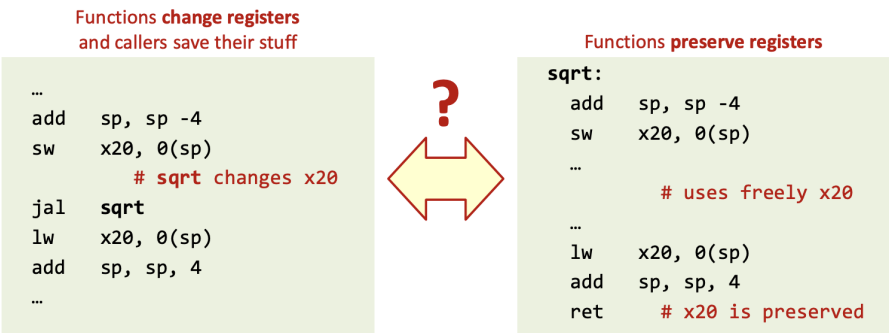



## 2.4.9 Register across functions

In assembly programming, handling registers across functions can be managed in two main ways: either functions **change registers** and expect the caller to save their values, or functions **preserve registers** and ensure that the register values remain the same across function calls.

- On the left, the function `sqrt` changes the value of register `x20`, requiring the caller to save and restore its value.
- On the right, the function `sqrt` preserves the value of `x20`, ensuring that the caller does not need to manage the saving and restoring.

This distinction is important, but it does not cause issues as long as there is agreement on how registers are handled.
*In case it's still not clear, we're looking at the `sw` instruction*




### 2.4.10 Preserving Registers

In RISC-V, register preservation is managed through a combination of callee-saved and caller-saved registers. Callee-saved registers (such as `s0`, `s1`, and `s2-11`) are preserved by the called function, ensuring that their values remain unchanged after the function call.
Caller-saved registers (such as `t0`, `t1-2`, and `t3-6`) are temporary and do not need to be preserved by the called function, meaning the caller must save them if their values are important.

| Register | ABI Name | Description | Preserved across call? |
|---|---|---|---|
| x0 | zero | Hard-wired zero | — |
| x1 | ra | Return address | No |
| x2 | sp | Stack pointer | Yes |
| x5 | t0 | Temporary/alternate link register | No |
| x6–7 | t1–2 | Temporaries | No |
| x8 | s0/fp | Saved register/frame pointer | Yes |
| x9 | s1 | Saved register | Yes |
| x18–27 | s2–11 | Saved registers | Yes |
| x28–31 | t3–6 | Temporaries | No |

## 2.5 Passing Arguments in RISC-V

In RISC-V, there are two main ways to pass arguments to functions:

### 2.5.1 Option 1: Using Registers

- Specific registers are used to pass arguments and return results.

- This can be done in a straightforward way, where each function uses different registers (e.g., passing an argument in `x5` and returning the result in `x6`).

- A more structured approach is to follow a convention where arguments are passed in registers `x10` to `x17`, with results returned in `x10`.

- The limitation: if there are more arguments than available registers (e.g., more than 8 arguments), this approach is insufficient.

### 2.5.2 Option 2: Using the Stack

- When registers are not enough, extra arguments are placed on the stack.

- The stack offers a universal solution because it has no practical limit on size.

- However, using the stack is more complex and requires additional work compared to using registers.

### 2.5.3 The RISC-V Approach

- RISC-V uses a combination of both methods.

- Registers `x10` to `x17` are used to pass arguments, with `x10` and `x11` also handling return values.

- If more arguments are needed beyond what these registers can handle, they are passed via the stack.

| Register | ABI Name | Description | Preserved across call? |
|---|---|---|---|
| x10–11 | a0–1 | Function arguments/return values | No |
| x12–17 | a2–7 | Function arguments | No |

*Register reserved for arguments and return values in RISC-V.*

## 2.6 Summary of RISC-V Register Conventions

Not covered in CS-200

| Register | ABI Name | Description | Preserved across call? |
|---|---|---|---|
| x0 | zero | Hard-wired zero | — |
| x1 | ra | Return address | No |
| x2 | sp | Stack pointer | Yes |
| x3 | gp | Global pointer | — |
| x4 | tp | Thread pointer | — |
| x5 | t0 | Temporary/~~alternate link register~~ | No |
| x6–7 | t1–2 | Temporaries | No |
| x8 | s0/fp | Saved register/~~frame pointer~~ | Yes |
| x9 | s1 | Saved register | Yes |
| x10–11 | a0–1 | Function arguments/return values | No |
| x12–17 | a2–7 | Function arguments | No |
| x18–27 | s2–11 | Saved registers | Yes |
| x28–31 | t3–6 | Temporaries | No |