# Computer Architecture
# (Assembly Language Overview)

विद्याधनं सर्वधन प्रधानम्

**Subhasis Bhattacharjee**

Computer Science and Engineering,
Indian Institute of Technology, Jammu

October 15, 2023

# Outline I

# Overview of Assembly Language

# Assembly Language

## What is an Assembly Language?

The term **assembly language** refers to a family of low-level programming languages that are specific to an ISA. They have a generic structure that consists of a sequence of assembly statements.

# Assembly Language - Basic Characteristics

- A low level programming language that uses simple statements.
- Each statement corresponds to **typically** just one machine instruction.
- These languages are specific to the ISA (Instruction Set Architecture).
- **Typically**, each assembly statement has two parts:
  1. Instruction Code that is a mnemonic for a basic machine instruction,
  2. A list of operands.

# Need for Learning Assembly Language

- Understanding processor and memory function
- Complete control over a system's resources
- Understanding processor and memory function
- Direct access to hardware
- Software developers' perspective
  - Write highly efficient code.
  - Suitable for the core parts of games, and mission critical software.
  - Write code for operating systems and device drivers.
  - Use features of the machine that are not supported by standard programming languages.

# Hardware Designers Perspective

- Learning the assembly language is the same as learning the intricacies of the instruction set.
- For Hardware designers: what to build?

# Assembly Language is Transparent

- This is largely since it has a small number of operations.
- So, this is very helpful for algorithm analysis
- Less worry about semantics and flow of control.
- Easier for debugging, as it is less complex.
- Less overhead as compared to high-level languages.

# Assemblers

## Assembler

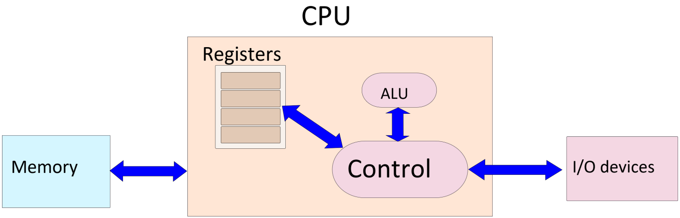
An Assembler is program that converts programs written in Assembly Language (i.e., low level languages) to machine code (0s and 1s).
Examples: **nasm**, **tasm**, and **masm** for x86 ISAs

## Using Assembler in UNIX system

- On a linux system try :
- gcc -S <filename.c>
- filename.s is its assembly representation
- Then type: gcc filename.s (will generate a binary: a.out)

# A Simple Computer Model

# Machine Model – Von Neumann Machine



# View of Registers

## Registers → named storage locations

- in ARM: r0, r1, $\cdots$ r15
- in x86: eax, ebx, ecx, edx, esi, edi

## Machine specific registers (MSR)

- Examples: Control the machine such as the speed of fans, power control settings
- Read the on-chip temperature.

## Registers with special functions

- stack pointer
- program counter
- return address

# View of Memory

- Memory
  - One large array of bytes
  - Each location has an address
  - The address of the first location is 0, and increases by 1 for each subsequent location
- The program is stored in a part of the memory
- The program counter contains the address of the current instruction

# Storage of Data in Memory

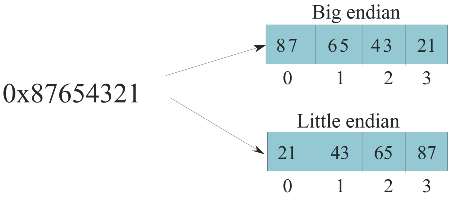
## Data Types

- char (1 byte)
- short (2 bytes)
- int (4 bytes)
- long int (8 bytes)

## multibyte variables stored in memory

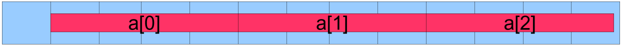
- Example: How is a 4 byte integer stored?
  - Save the 4 bytes in consecutive locations
- Next question: which order? Little endian or Big endian.
  - Little endian representation (used in ARM and x86) $\rightarrow$ The LSB is stored in the lowest location
  - Big endian representation (Sun Sparc, IBM PPC) $\rightarrow$ The MSB is stored in the lowest location

# Arrangements of Bytes - Little Endian vs Big Endian




Note the order of the storage of bytes

# Storage of Arrays in Memory




- Single dimensional arrays.
- Consider an array of integers: a[100]
- Each integer is stored in either a little endian or big endian format
- 2 dimensional arrays :
  - int a[100][100]
  - float b[100][100]
  - Two methods: row major and column major

# Row Major vs Column Major

## Row Major (C, Python)

- Store the first row as an 1D array
- Then store the second row, and so on.

## Column Major (Fortran, Matlab)

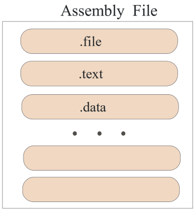
- Store the first column as an 1D array
- Then store the second column, and so on

## Multidimensional arrays

Store the entire array as a sequence of 1D arrays

# Assembly Language Syntax

# Assembly Language Program Structure




- Divided into different sections
- Each section contains some data, or assembly instructions

# Meaning of Different Sections

- .file
- name of the source file
- .text
- contains the list of instructions
- .data
- data used by the program in terms of read only variables, and constants

# Structure of Instruction


- instruction: textual identifier of a machine instruction
- operand: Usually can be any or mix of these
  - constant (also known as an immediate)
  - register
  - memory location

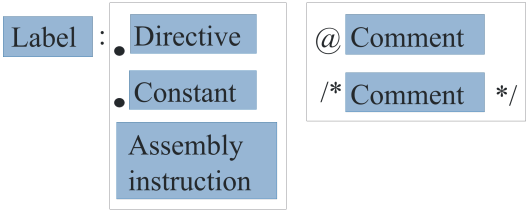
# Examples of Instructions

- subtract the contents of r2 from the contents of r1, and save the result in r3
- multiply the contents of r2 with the contents of r1, and save the results in r3

```
sub r3, r1, r2
mul r3, r1, r2
```

**We will see a lot more examples later on.**

# Generic Statement Structure in Assembly Language Program




- label → identifier of a statement
- directive → tells the assembler to do something like declare a function
- constant → declares a constant
- assembly statement → contains the assembly instruction, and operands
- comment → textual annotations ignored by the assembler

# Types of Instructions

### Data Processing Instructions

add, subtract, multiply, divide, compare, logical or, logical and

### Data Transfer Instructions

transfer values between registers, and memory locations

### Branch instructions

branch to a given label

### Special instructions

interact with peripheral devices, and other programs, set machine specific parameters

# Survey of Instruction Sets

| ISA | Type | Year | Vendor | Bits | Endianness | Registers |
|---|---|---|---|---|---|---|
| VAX | CISC | 1977 | DEC | 32 | little | 16 |
| SPARC | RISC | 1986 | Sun | 32 | big | 32 |
| | RISC | 1993 | Sun | 64 | bi | 32 |
| PowerPC | RISC | 1992 | Apple,IBM,Motorola | 32 | bi | 32 |
| | RISC | 2002 | Apple,IBM | 64 | bi | 32 |
| PA-RISC | RISC | 1986 | HP | 32 | big | 32 |
| | RISC | 1996 | HP | 64 | big | 32 |
| m68000 | CISC | 1979 | Motorola | 16 | big | 16 |
| | CISC | 1979 | Motorola | 32 | big | 16 |
| MIPS | RISC | 1981 | MIPS | 32 | bi | 32 |
| | RISC | 1999 | MIPS | 64 | bi | 32 |
| Alpha | RISC | 1992 | DEC | 64 | bi | 32 |
| x86 | CISC | 1978 | Intel,AMD | 16 | little | 8 |
| | CISC | 1985 | Intel,AMD | 32 | little | 8 |
| | CISC | 2003 | Intel,AMD | 64 | little | 16 |
| ARM | RISC | 1985 | ARM | 32 | bi(little default) | 16 |
| | RISC | 2011 | ARM | 64 | bi(little default) | 31 |

# SimpleRISC ISA

# SimpleRisc - Hypothetical Design only

- Simple RISC ISA
- Contains only 21 instructions
- We will design an assembly language for SimpleRisc
- Design a simple binary encoding,
- Hardware Design & Implementation
- Performance Improvement by redesigning Hardware

# Registers & Fundamental Model

## SimpleRisc has 16 registers

- Numbered: r0 $\cdots$ r15
- r14 is also referred to as the stack pointer (sp)
- r15 is also referred to as the return address register (ra)

## View of Memory

- Von Neumann model
- One large array of bytes

## Special flags register

- Special flags register $\rightarrow$ contains the result of the last comparison
- flags.E $= 1$ (equality), flags.GT $= 1$ (greater than)

# SimpleRISC: Moving data

# Instructions for Moving Data

## Find the Requirements

- Where can we have our data?
- Register, memory, immediate literal value.
- What kind of data movements are required?

## Variations on Data Movement

- Moving data between registers. [ **mov** instruction]
- Moving an immediate value into a register. [ **mov** instruction]
- Moving data between register and memory. [ **ld** and **st** instructions - will study later on]

# **mov** instruction

## **mov** instruction - snapshot

| **Instruction Snapshot** | **Explanation** |
|---|---|
| mov *dest*, *src* | *dest* $\leftarrow$ *src* |
| mov r1, r2 | r1 $\leftarrow$ r2 |
| mov r1, 3 | r1 $\leftarrow$ 3 |

- Transfer the contents of one register to another
- Or, transfer the contents of an immediate to a register
- The value of the immediate is embedded in the instruction
  - SimpleRisc has 16 bit immediates
  - Range -215 to 215 - 1

# SimpleRISC: Arithmetic Instructions

# Arithmetic Instructions

- SimpleRisc has 6 arithmetic instructions
- add, sub, mul, div, mod, cmp

| **Instruction Snapshot** | **Explanation** |
|---|---|
| add r1, r2, r3 | r1 $\leftarrow$ r2 + r3 |
| add r1, r2, 10 | r1 $\leftarrow$ r2 + 10 |
| sub r1, r2, r3 | r1 $\leftarrow$ r2 – r3 |
| mul r1, r2, r3 | r1 $\leftarrow$ r2 * r3 |
| div r1, r2, r3 | r1 $\leftarrow$ r2/r3 (quotient) |
| mod r1, r2, r3 | r1 $\leftarrow$ r2 mod r3 (remainder) |
| cmp r1, r2 | set flags |

# Arithmetic Instructions - Example 1

## Convert the following code to assembly

```
a = 3
b = 5
c = a + b
d = c - 5
```

## Solution:

Assign the variables to registers

- a $\leftarrow$ r0, b $\leftarrow$ r1, c $\leftarrow$ r2, d $\leftarrow$ r3

```
mov r0, 3
mov r1, 5
add r2, r0, r1
sub r3, r2, 5
```

# Arithmetic Instructions - Example 2

## Convert the following code to assembly

```
a = 3
b = 5
c = a * b
d = c mod 5
```

## Solution:

- Assign the variables to registers
- a $\leftarrow$ r0, b $\leftarrow$ r1, c $\leftarrow$ r2, d $\leftarrow$ r3

```
mov r0, 3
mov r1, 5
mul r2, r0, r1
mod r3, r2, 5
```

# Compare Instruction - Example 1

## Convert the following code to assembly & get the value of the flags

```
a = 3
b = 5
compare a and b
```

## Solution:

- Assign the variables to registers
- a $\leftarrow$ r0, b $\leftarrow$ r1

```
mov r0, 3
mov r1, 5
cmp r0, r1
```

- flags.E = 0, flags.GT = 0

# Compare Instruction - Example 2

## Convert the following code to assembly & get the value of the flags

```
a = 5
b = 3
compare a and b
```

## Solution:

- Assign the variables to registers
- a $\leftarrow$ r0, b $\leftarrow$ r1

```
mov r0, 5
mov r1, 3
cmp r0, r1
```

- flags.E = 0, flags.GT = 1

# Compare Instruction - Example 3

## Convert the following code to assembly & get the value of the flags

```
a = 5
b = 5
compare a and b
```

## Solution:

- Assign the variables to registers
- a $\leftarrow$ r0, b $\leftarrow$ r1

```
mov r0, 5
mov r1, 5
cmp r0, r1
```

- flags.E = 1, flags.GT = 0

# More Involved Arithmetic Computations - Example

## Write Assembly Code for the following.

Compute: 31 / 29 - 50, and save the result in r4.

## Solution:

- Assign the variables to registers
- Let us keep r4 free for the result.
- 31 $\leftarrow$ r1, 29 $\leftarrow$ r2, r3 $\leftarrow$ 31 / 29, r4 $\leftarrow$ (31 / 29 - 50)

```
mov r1, 31
mov r2, 29
div r3, r1, r2
sub r4, r3, 50
```

# SimpleRISC: Logical Instructions

# Logical Instructions

| Instruction Snapshot | Explanation | Textual Explanation |
|---|---|---|
| and r1, r2, r3 | r1 $\leftarrow$ r2 & r3 | bitwise AND |
| or r1, r2, r3 | r1 $\leftarrow$ r2 $\mid$ r3 | bitwise OR |
| not r1, r2 | r1 $\leftarrow$ ˜r2 | logical complement |

The second argument can either be a register or an immediate value.

# Logical Instructions - Example

## Compute (a | b)

Solution:

- Assign the variables to registers.
- a $\leftarrow$ r0, b $\leftarrow$ r1, r2 to store the result.

```
or r2, r0, r1
```

# SimpleRISC: Shift Instructions

# Logical & Arithmetic Shift Concept

## Logical shift left (lsl) – We use $\ll$ operator

- $0010 \ll 2$ is equal to $1000$ [Note: **0 is entering from right**]
- $(\ll n)$ is the same as multiplying by $2^n$

## logical shift right (lsr) – We use $\ggg$ operator

- $1000 \ggg 2 = 0010$ [Note: **0 is entering from left**]
- $(\ggg n)$ is the same as dividing the unsigned representation by $2^n$

## Arithmetic shift right (asr) – We use $\gg$ operator

- $0010 \gg 1 = 0001$ [Note: **0 is entering from left**]
- $1000 \gg 2 = 1110$ [Note: **1 is entering from left**]
- Input is assumed to be a **signed integer**.
- $(\gg n)$ same as dividing a signed number by $2^n$

# Shift Instructions

| Instruction Snapshot | Explanation |
|---|---|
| lsl r3, r1, r2 | r3 $\leftarrow$ r1 $\ll$ r2 (shift left) |
| lsl r3, r1, 4 | r3 $\leftarrow$ r1 $\ll$ 4 (shift left) |
| lsr r3, r1, r2 | r3 $\leftarrow$ r1 $\gg>$ r2 (shift right logical) |
| lsr r3, r1, 4 | r3 $\leftarrow$ r1 $\gg>$ 4 (shift right logical) |
| asr r3, r1, r2 | r3 $\leftarrow$ r1 $\gg$ r2 (arithmetic shift right) |
| asr r3, r1, 4 | r3 $\leftarrow$ r1 $\gg$ r2 (arithmetic shift right) |

# Shift Instructions - Example 1

## Compute 101 * 6 with shift operators

- We need to compute 101 * (4 + 2)
- = 101 * 4 + 101 * 2, or, $101 * 2^2 + 101 * 2^1$
  - Let a = $101 * 2^2 \rightarrow 101 \ll 2$
  - Let b = $101 * 2^1 \rightarrow 101 \ll 1$
  - Let c = a + b

```
mov r0, 101
lsl r1, r0, 1
lsl r2, r0, 2
add r3, r1, r2
```

# Shift Instructions - Example 2

## Compute 102 * 7.5 with shift operators

- We need to compute 102 * (8 - 0.5)
- = 102 * 8 - 102 * 0.5, or, 101 * $2^3$ + 102 * $2^{-1}$
  - Let a = 102 * $2^3 \rightarrow$ 101 $\ll$ 3
  - Let b = 102 * $2^1 \rightarrow$ 101 $\ggg$ 1
  - Let c = a - b

```
mov r0, 102
lsl r1, r0, 3
lsr r2, r0, 1
sub r3, r1, r2
```

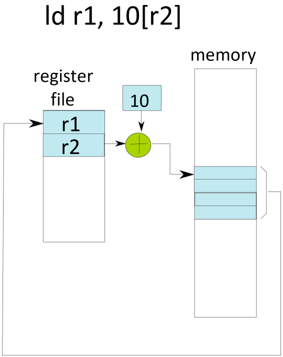
# SimpleRISC: Load & Store Instructions

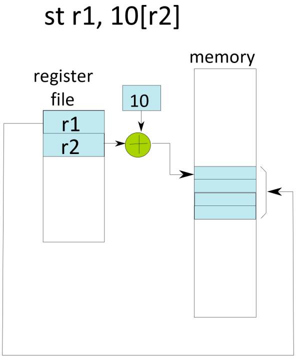
# Memory Access - Load & Store Instructions

| Instruction Snapshot | Explanation | Meaning |
|---|---|---|
| ld r1, 10[r2] | r1 $\leftarrow$ [r2 +10] | Load value from memory |
| st r1, 10[r2] | [r2+10] $\leftarrow$ r1 | Store value into memory |

- 2 address format, base-offset addressing
- Fetch the contents of r2, add the offset (10), and then perform the memory access

# Load-Store




Note the direction of arrow between register and memory.

# Load/Store - Example 1

## Translate

```
int arr[10];
arr[3] = 5;
arr[4] = 8;
arr[5] = arr[4] + arr[3];
```

```
/* assume base of array saved in r0 */
mov r1, 5
st r1, 12[r0]
mov r2, 8
st r2, 16[r0]
add r3, r1, r2
st r3, 20[r0]
```

# SimpleRISC: Branch Instructions

# Branch Instructions

## Unconditional branch instruction

| Instruction Snapshot | Meaning |
|---|---|
| b .foo | branch to .foo |

## Example branch

```
add r1, r2, r3
b .foo
...
...
.foo:
add r3, r1, r4
```

# Conditional Branch Instructions

The flags are only set by cmp instructions.

| Snapshot | Explanation | Textual Explanation |
| --- | --- | --- |
| beq .foo | branch to .foo if flags.E = 1 | branch if equal |
| bgt .foo | branch to .foo if flags.GT = 1 | branch if greater than |

# Conditional Branch Instructions - Example 1

## Example - Conditional Branch

If r1 > r2, then save 4 in r3, else save 5 in r3.

```
cmp r1, r2
bgt .gtlabel
mov r3, 5
...
...
.gtlabel: mov r3, 4
```

# Conditional Branch Instructions - Example 2

## Example - Conditional Branch & C Code

Compute the factorial of the variable num.

```c
int prod = 1;
int idx;
for(idx = num; idx > 1; idx --) {
  prod = prod * idx
}
```

## Solution: Conditional Branch & Assembly Code

```
mov r1, 1 /* prod = 1 */
mov r2, r0 /* idx = num */
.loop:
        mul r1, r1, r2 /* prod = prod * idx */
        sub r2, r2, 1 /* idx = idx - 1 */
        cmp r2, 1 /* compare (idx, 1) */
        bgt .loop /* if (idx > 1) goto .loop*/
```

# SimpleRISC: Modifiers in Instructions

# Modifiers - immediate operand

- Following modifiers to an instruction that has an immediate operand
- Modifier :
  - default: mov $\rightarrow$ treat the 16 bit immediate as a signed number (automatic sign extension)
  - (u) : movu $\rightarrow$ treat the 16 bit immediate as an unsigned number
  - (h) : movh $\rightarrow$ left shift the 16 bit immediate by 16 positions

# Mechanism of Computing with Modifiers

- The processor internally converts a 16 bit immediate to a 32 bit number
- It uses this 32 bit number for all the computations
- Valid only for arithmetic/logical insts
- We can control the generation of this 32 bit number
  - sign extension (default)
  - treat the 16 bit number as unsigned (u suffix)
  - load the 16 bit number in the upper bytes (h suffix)

# **mov** Instruction Modifier - Example 1

default : mov r1, 0xAB 12

| | | | |
|---|---|---|---|
| sign bit | | AB | 12 |

unsigned : movu r1, 0xAB 12

| | | | |
|---|---|---|---|
| 00 | 00 | AB | 12 |

high: movh r1, 0xAB 12

| | | | |
|---|---|---|---|
| AB | 12 | 00 | 00 |

# **mov** Instruction Modifier - Example 2

Move : 0x FF FF A3 2B in r0

```
mov r0, 0xA32B
```

Move : 0x 00 00 A3 2B in r0

```
movu r0, 0xA32B
```

Move : 0x A3 2B 00 00 in r0

```
movh r0, 0xA32B
```

# **mov** Instruction Modifier - Example 3

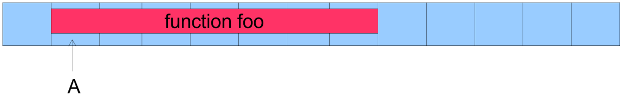
Set r0 ← 0x 12 AB A9 2D

```
movh r0, 0x 12 AB
addu  r0, 0x A9 2D
```

# Functions and Stacks

# Implementing Functions

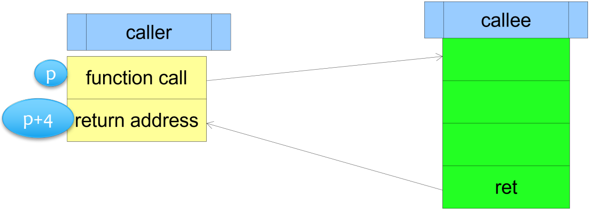
- Functions are blocks of assembly instructions that can be repeatedly invoked to perform a certain action
- Every function has a starting address in memory
- Eg., foo has a starting address A




- To call a function, we need to set :
  - pc $\leftarrow$ A
- We also need to store the location of the pc that we need to come to after the function returns (Called as **return address**)
- We can call any function, execute its instructions, and then return to the saved return address

# Notion of the Return Address

- PC of the call instruction $\rightarrow$ p
- PC of the return address $\rightarrow$ p + 4
  - because, every instruction takes 4 bytes)



# Passing Arguments / Return Value - using register

```
.foo:
    add r2, r0, r1
    ret
.main:
    mov r0, 3
    mov r1, 5
    call .foo
    add r3, r2, 10
```

# Problems of Using Registers for Storing

## Space Problem

- We have a limited number of registers
- We cannot pass more than 16 arguments
- Solution: Use memory also

## Overwrite Problem

- What if a function calls itself ? (recursive call)
- The callee can overwrite the registers of the caller
- Solution: Spilling

# Register Spilling

## Register Spilling into Memory

- Save the set of registers its needs (including the PC)
- Call the function
- And, then restore the set of registers after the function returns.

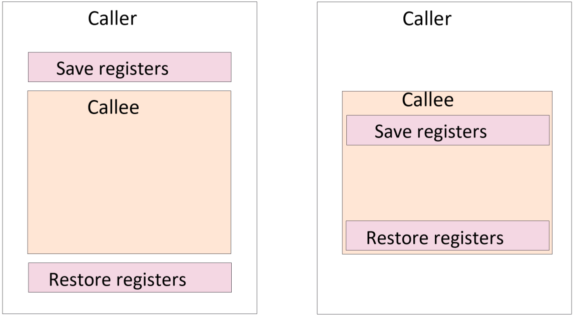
## Caller saved scheme

The caller the registers before calling the function, and later restores them after the function returns.

## callee saved scheme

The callee the registers before calling the function, and later restores them after the function returns.

# Register Spilling - Caller & Callee




(a) Caller saved

(b) Callee saved

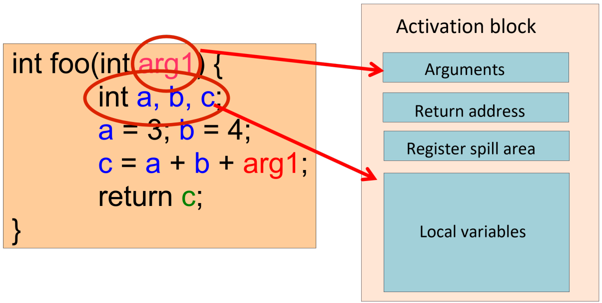
# Where to Store in Memory

- Using memory, and spilling solves both the space problem and overwrite problem
- However, there needs to be :
  - a strict agreement between the caller and the callee regarding the set of memory locations that need to be used
  - Secondly, after a function has finished execution, all the space that it uses needs to be reclaimed

# Activation Block

## Activation block:

The memory map of a function arguments, register spill area, local vars.



# Using Activation Blocks

## Caller saved scheme:

- Before calling a function:
  - Allocate the activation block of the callee
  - Spill the registers
  - Write the arguments to the activation block of the callee
- Call the function
- In the called function:
  - Read the arguments and transfer to registers (if required)
  - Save the return address if the called function can call other functions
  - Allocate space for local variables
  - Execute the function
- Once the function ends:
  - Restore the value of the return address register (if required)
  - Write the return values to registers, or activation block of the caller
  - Destroy the activation block of the callee

# Activation Blocks Organization on Stack

# Organising Activation Blocks

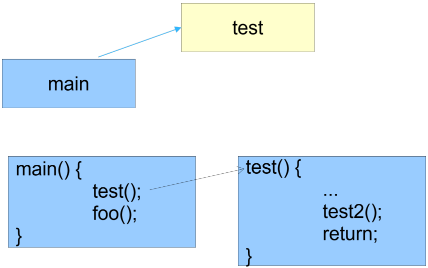
- All the information of an executing function is stored in its activation block
- These blocks need to be dynamically created and destroyed – millions of times
- What is the correct way of managing them, and ensuring their fast creation and deletion ?
- Is there a pattern ?foo

# Function Call - Example - 1 - Page 1

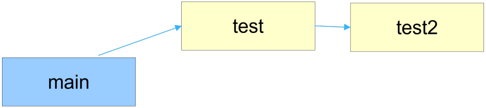
main

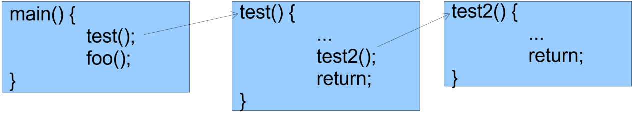
```
main() {
        test();
        foo();
}
```

# Function Call - Example - 1 - Page 2

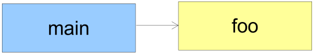


# Function Call - Example - 1 - Page 3



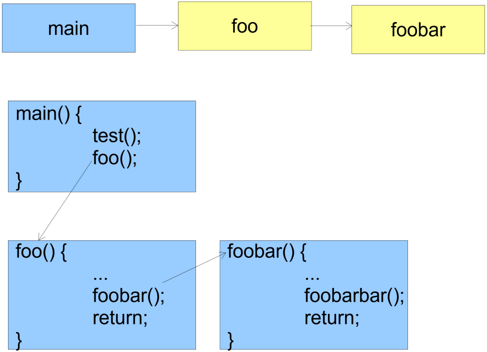
# Function Call - Example - 1 - Page 4



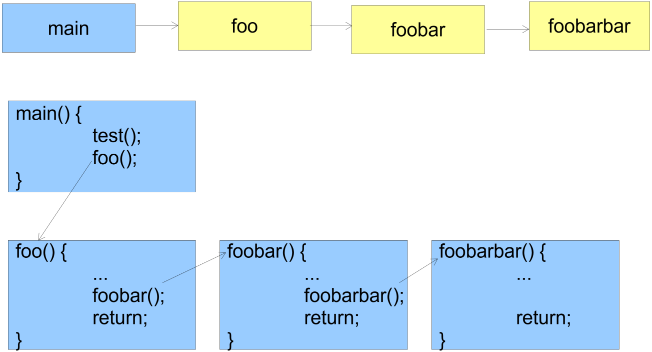

```
main() {
        test();
        foo();
}
```

```
foo() {
        ...
        foobar();
        return;
}
```

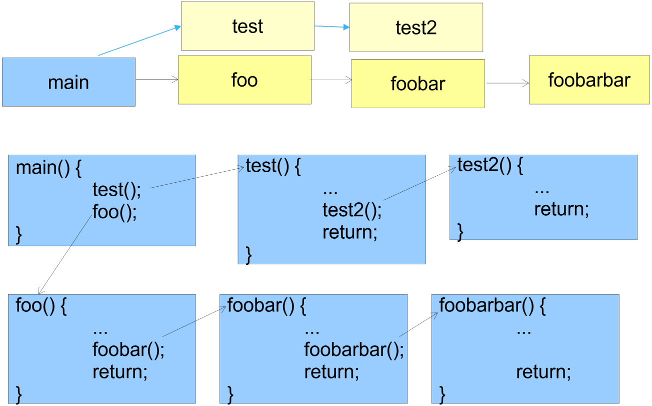
# Function Call - Example - 1 - Page 5



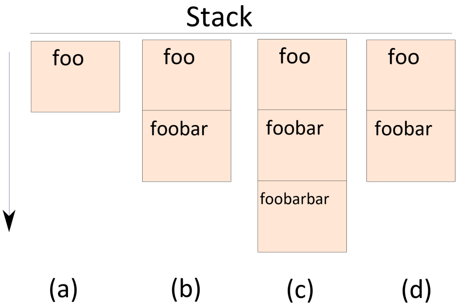
# Function Call - Example - 1 - Page 6



# Function Call - Example - 1 - Page 7



# Function Call - Example - 1 - Stack Grow & Sink




- Last in First Out
- Use a stack to store activation blocks

# Working with the Stack

- Allocate a part of the memory to save the stack
- Traditionally stacks are downward growing.
  - The first activation block starts at the highest address
  - Subsequent activation blocks are allocated lower addresses
- The stack pointer register (sp (R14)) points to the beginning of an activation block
- Allocating an activation block :
  - sp $\leftarrow$ sp - <constant>
- De-allocating an activation block :
  - sp $\leftarrow$ sp + <constant>

# Issues Solved using Stack

- Space problem: Pass as many parameters as required in the activation block
- Overwrite problem: Solved by activation blocks
- Management of activation blocks: Solved by the notion of the stack

# Function Calling Instructions

# call and ret instructions

## Return address register

ra (or r15) $\rightarrow$ return address register
In ARM, this register is called **lr (link register)**

| **Instr.** | **Explanation** | **Textual Explanation** |
|---|---|---|
| call .foo | ra $\leftarrow$ PC + 4; PC $\leftarrow$ address(.foo) | call instruction: Puts pc + 4 in ra, and jumps to the function |
| ret | PC $\leftarrow$ ra | ret instruction: Puts value of ra into pc |

# Recursive Factorial Program - Example 1

```
int factorial(int num) {
  if (num <= 1) return 1;
  return num * factorial(num - 1);
}
void main() {
  int result = factorial(10);
}
```

# Factorial in SimpleRisc - Example 1

```
.factorial:
        cmp r0, 1 /* compare (1,num) */
        beq .return
        bgt .continue
        b .return
.continue:
        sub sp, sp, 8 /* create space on the stack */
        st r0, [sp] /* push r0 on the stack */
        st ra, 4[sp] /* push the return address register */
        sub r0, r0, 1 /* num = num - 1 */
        call .factorial /* result will be in r1 */
        ld r0, [sp] /* pop r0 from the stack */
        ld ra, 4[sp] /* restore the return address */
        mul r1, r0, r1 /* factorial(n) = n * factorial(n-1) */
        add sp, sp, 8 /* delete the activation block */
        ret
.return: mov r1, 1
        ret
.main:  mov r0, 10
        call .factorial
```

# nop instruction

- nop $\rightarrow$ does nothing
- Example: nop

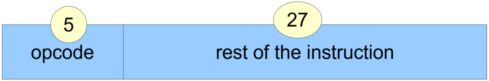
# SimpleRISC: Instruction Encoding

# Encoding of Opcode in Instructions

- Encode the SimpleRisc ISA using 32 bits.
- We have 21 instructions. Let us allot each instruction an unique code (opcode)

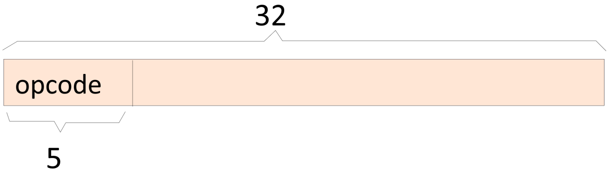
| Instruction | Code | Instruction | Code | Instruction | Code |
|---|---|---|---|---|---|
| add | 00000 | not | 01000 | beq | 10000 |
| sub | 00001 | mov | 01001 | bgt | 10001 |
| mul | 00010 | lsl | 01010 | b | 10010 |
| div | 00011 | lsr | 01011 | call | 10011 |
| mod | 00100 | asr | 01100 | ret | 10100 |
| cmp | 00101 | nop | 01101 | | |
| and | 00110 | ld | 01110 | | |
| or | 00111 | st | 01111 | | |

# Basic Instruction Format



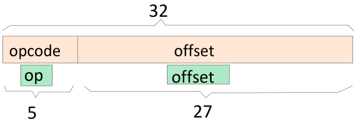

| Inst. | Code | Format | Inst. | Code | Format |
|---|---|---|---|---|---|
| add | 00000 | add rd,rs1, (rs2/imm) | lsl | 01010 | lsl rd, rs1, (rs2/imm) |
| sub | 00001 | sub rd,rs1, (rs2/imm) | lsr | 01011 | lsr rd, rs1, (rs2/imm) |
| mul | 00010 | mul rd, rs1, (rs2/imm) | asr | 01100 | asr rd, rs1, (rs2/imm) |
| div | 00011 | div rd, rs1, (rs2/imm) | nop | 01101 | nop |
| mod | 00100 | mod rd, rs1, (rs2/imm) | ld | 01110 | ld rd.imm[rs1] |
| cmp | 00101 | cmp rs1, (rs2/imm) | st | 01111 | st rd. imm[rs1] |
| and | 00110 | and rd, rs1, (rs2/imm) | beq | 10000 | beq offset |
| or | 00111 | or rd,rs1, (rs2/imm) | bgt | 10001 | bgt offset |
| not | 01000 | not rd, (rs2/imm) | b | 10010 | b offset |
| mov | 01001 | mov rd, (rs2/imm) | call | 10011 | call offset |
| ret | 10100 | ret | | | |

# 0-Address Instructions



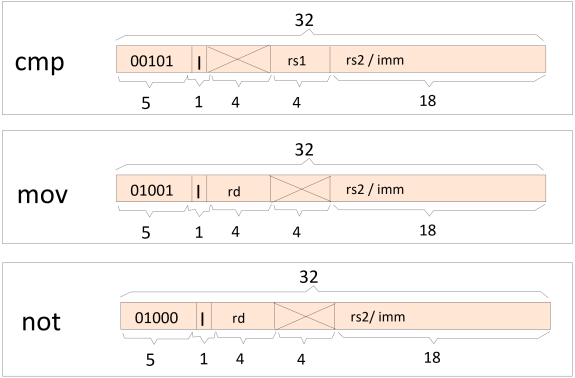

- nop and ret instructions

# 1-Address Instructions




- Instructions – call, b, beq, bgt
- Use the branch format
- Fields :
  - 5 bit opcode
  - 27 bit offset (PC relative addressing)
  - Since the offset points to a 4 byte word address
  - The actual address computed is : PC + offset * 4

# 2-Address Instructions

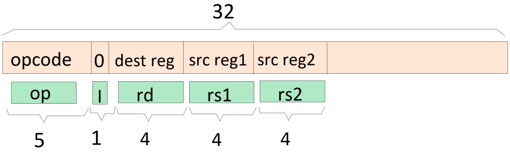



- cmp, not, and mov
- Use the 3 address: immediate or register formats
- Do not use one of the fields

# 3-Address Instructions

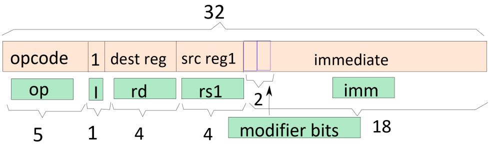
- Instructions – add, sub, mul, div, mod, and, or, lsl, lsr, asr
- Generic 3 address instruction
  - <opcode> rd, rs1, <rs2/imm>
- I bit to specify if the second operand is an immediate or a register.
  - $I = 0 \rightarrow$ second operand is a register
  - $I = 1 \rightarrow$ second operand is an immediate
- Since we have 16 registers, we need 4 bits to specify a register

# 3-Address Instructions - Register Format




- opcode $\rightarrow$ type of the instruction
- I bit $\rightarrow$ 0 (second operand is a register)
- dest reg $\rightarrow$ rd
- source register 1 $\rightarrow$ rs1
- source register 2 $\rightarrow$ rs2

# 3-Address Instructions - Immediate Format




- opcode $\rightarrow$ type of the instruction
- I bit $\rightarrow$ 1 (second operand is an immediate)
- dest reg $\rightarrow$ rd
- source register 1 $\rightarrow$ rs1
- Immediate $\rightarrow$ imm
- modifier bits $\rightarrow$ 00 (default), 01 (u), 10 (h)

# Instruction Format - Load & Store

## Load

ld rd, imm[rs1]

32 bits:

| 01110 | 1 | rd | rs1 | imm |
|---|---|---|---|---|
| 5 | 1 | 4 | 4 | 18 |

## Store

st rd, imm[rs1]

32 bits:

| 01111 | 1 | rd | rs1 | imm |
|---|---|---|---|---|
| 5 | 1 | 4 | 4 | 18 |

# Load and Store Instructions

## Load

- ld rd, imm[rs1]
- rs1 $\rightarrow$ base register
- Use the immediate format.

## Store Instruction

- st reg1, imm[reg2]
- has two register sources, no register destination, 1 immediate
  - Let us make an exception and use the immediate format.
  - We use the rd field to save one of the source registers
  - st rd, imm[rs1]

# Summary of Instruction Formats

| Format | Definition | | | | |
|---|---|---|---|---|---|
| *branch* | *op* (28-32) | *offset* (1-27) | | | |
| *register* | *op* (28-32) | I (27) | *rd* (23-26) | *rs* 1(19-22) | *rs* 2(15-18) |
| *immediate* | *op* (28-32) | I (27) | *rd* (23-26) | *rs* 1(19-22) | *imm* (1-18) |
| *op* → opcode, *offset* → branch offset, *I* → immediate bit, *rd* → destination register | | | | | |
| *rs*1 → source register 1, *rs*2 → source register 2, *imm* → immediate operand | | | | | |

- branch format → nop, ret, call, b, beq, bgt
- register format → ALU instructions
- immediate format → ALU, ld/st instructions