

# Assembly Language Programming

# Outline

- **Introduction to Assembly Language**
- Arithmetic and Logic Instructions
- Input / Output Instructions
- Arithmetic Instructions with Immediates
- Branches and Jumps
- Conditional Execution and Loops
- Memory Instructions
- Variables in Assembly Language
- Function Calls

# Assembly Language

- **Assembly language** is a low level programming language that provides direct access to registers and memory.
- Assembly language is dependent on the microprocessor family.
- An assembly language program consists of a sequence of instructions.
- There is a direct correspondence between an assembly language instruction and a machine language instruction (raw 1s and 0s).

# ANNA Architecture

In this class, we will be using the ANNA Architecture. It has these characteristics:

- 16 bit data words (can represent two's complement integers or addresses)
- 16 bit instructions
- ( $2^{16}$ ) 64K words of word-addressable main memory
- 16 different instructions
- 8 general purpose 16-bit registers (numbered from r0 to r7)
- 16 bit ALU
- 16 bit program counter (PC)
- Register r0 always has the value 0
- No hardware support / instructions for floating point values

# Assembly Language Programming Process

Assembly Language Programming Process (in ANNA):

1. Write the program in assembly language (saves to a `.ac` file).
2. Assemble the program using an assembler. The result is a machine code file (`.mc` file).
3. Execute the machine code file in a simulator.

# Registers and Variables

Registers are memory locations inside the CPU.

- Registers can be: `r0`, `r1`, `r2`, `r3`, `r4`, `r5`, `r6`, `r7`.
- Remember that `r0` always has the value 0, even if you try to write to it.

Variables within a program can either be stored in memory or registers.

- In addition, registers need to store temporary values for longer expressions such as `"a = 3 * b + c / d;"`
- Initially, in this unit we are going to assume that all variables are stored in registers. Later, we will discuss how to handle variables stored in memory.

# Program Counter

- The **program counter (PC)** is a special register that keeps track of the address of which instruction is currently executing.
- The machine program is stored in memory starting at address 0.
- The PC cannot be directly accessed by assembly instructions.
- The PC is incremented each cycle.
- Some instructions (branches and jumps) can update the PC. This is needed to implement control decisions (if, switch), loops, and function calls.

# ANNA Simulator

The simulator performs these steps upon startup:

1. Each location in memory is filled with zero.
2. All of the registers are set to zero.
3. The program counter (PC) is set to zero.
4. The program is loaded into memory from a file.



# ANNA Simulator

The simulator is then controlled by the user. The simplest command is to run the program. In this mode, the simulator:

1. Fetches the instruction at PC.
2. Executes the instruction.
3. Updates PC (typically  $PC+1$ ).
4. Repeats steps 1-3 until the program halts.

# Assembly Instructions

- All instructions have an **opcode** that describes the operation. Each of the 16 instructions in ANNA have a unique opcode.
- Most instructions have up to three operands or fields that specify...
  - where to get data that is used by the instruction (registers, memory, input, the instruction itself).
  - where to put data that is produced by the instruction (registers, memory, output).
- The opcodes and the fields are delimited by spaces.
- Each instruction appears on its own line (no semicolons to delimit instructions).

# Comments and Assembler Directives

- Comments are specified by using '#'.
  - Anything after the '#' sign on that line is treated as a comment.
  - Comments can either be placed on the same line after an instruction or as a standalone line.
- In addition to the 16 instructions, the ANNA assembler supports two assembler directives (`.halt` and `.fill`).
  - Assembler directives are commands to the assembler.

# Outline

- Introduction to Assembly Language
- **Arithmetic and Logic Instructions**
- Input / Output Instructions
- Arithmetic Instructions with Immediates
- Branches and Jumps
- Conditional Execution and Loops
- Memory Instructions
- Variables in Assembly Language
- Function Calls

# add

Your first ANNA assembly instruction, add, has the following format:

add *Rd* *Rs<sub>1</sub>* *Rs<sub>2</sub>*

$$R(Rd) \leftarrow R(Rs_1) + R(Rs_2)$$

- add is the opcode.
- *Rd* refers to the destination register: the result of the add is stored in the specified register.
- *Rs<sub>1</sub>* and *Rs<sub>2</sub>* refer to the source registers: they contain the numbers of the registers that contain the numbers to be added together.

Examples:

add r3 r2 r7

$$R(r3) = R(r2) + R(r7)$$

add r5 r5 r5

$$R(r5) = R(r5) + R(r5)$$

# Additional Notes about add

- This instruction only accesses registers, it does not access memory.
- Overflow is not detected for this or any other instruction.
  - We will assume that overflow never occurs.

# Other Arithmetic and Logic Instructions

Other arithmetic and logic instructions:

sub $Rd\ Rs_1\ Rs_2$	subtract	$R(Rd) \leftarrow R(Rs_1) - R(Rs_2)$
and $Rd\ Rs_1\ Rs_2$	bitwise and	$R(Rd) \leftarrow R(Rs_1) \& R(Rs_2)$
or $Rd\ Rs_1\ Rs_2$	bitwise or	$R(Rd) \leftarrow R(Rs_1)   R(Rs_2)$
not $Rd\ Rs_1$	bitwise not	$R(Rd) \leftarrow \sim R(Rs_1)$

# Fetch and Execute Loop

Fetch and execute loop for arithmetic and logic instructions:

1. Fetch the instruction at the offset in memory indicated by the PC.
2. Set  $PC \leftarrow PC + 1$ .
3. Get the value of the source register(s).
4. Perform the specified operation.
5. Place the result into the destination register.



# Class Problem

What is the state of the register file after this instruction sequence?

```
and r1 r2 r3  
or  r2 r4 r3  
add r6 r2 r7  
sub r3 r3 r5
```

	Before	After
<b>r1</b>	0xbead	
<b>r2</b>	0xface	
<b>r3</b>	0xaced	
<b>r4</b>	0x42a5	
<b>r5</b>	0x0005	
<b>r6</b>	0x4567	
<b>r7</b>	0xfffa	

# Conversion to Machine Language

- Remember that instructions in ANNA are 16 bits.
- The arithmetic and logic instructions all use the same general format, called the R-type format, as shown:

15	12	11	9	8	6	5	3	2	0
Opcode		Rd		Rs <sub>1</sub>		Rs <sub>2</sub>		Unused	

- The term R-type refers to instructions that use only register operands.

# Conversion to Machine Language

- The opcode identifies the instruction and is in bits 15-12 for all instructions.
  - Sixteen instructions → four bits of opcode
- The opcode bits for the arithmetic and logic instructions are shown in the table.
- The register fields indicate the register number used in that operand.
- The `not` instruction only uses two of the three fields: the  $Rs_2$  field is ignored (can be any value).

<i>Opcode mnemonic</i>	<i>Opcode bits</i>
<code>add</code>	0000
<code>sub</code>	0001
<code>and</code>	0010
<code>or</code>	0011
<code>not</code>	0100

<i>Register Number</i>	<i>Register bits</i>
<code>r0</code>	000
<code>r1</code>	001
<code>r2</code>	010
<code>r3</code>	011
<code>r4</code>	100
<code>r5</code>	101
<code>r6</code>	110
<code>r7</code>	111

# Machine Language Examples

Example: What is the machine language representation for these instructions?

1. `and r4 r7 r2`

# Machine Language Examples

2. sub r3 r0 r1

3. add r5 r5 r5

# Outline

- Introduction to Assembly Language
- Arithmetic and Logic Instructions
- **Input / Output Instructions**
- Arithmetic Instructions with Immediates
- Branches and Jumps
- Conditional Execution and Loops
- Memory Instructions
- Variables in Assembly Language
- Function Calls

# in

The input instruction `in` gets a word from input:

`in Rd`

- This instruction asks the user for a 16-bit number and places it into register *Rd*.
- The number specified by the user (of the simulator) can be specified using a signed number (-32,768 to 32,767) or a 4 digit hex number.

# out

The output instruction `out` displays the contents of a register onto the screen.

`out Rd`

- The instruction displays contents of *Rd* onto the screen in both decimal and hexadecimal format.



# Input / Output Instruction Notes

- Both the `in` and `out` instruction are R-type instructions ( $Rs_1$  and  $Rs_2$  fields are ignored).
- Opcode bits for `in`:           1110
- Opcode bits for `out`:       1111
- Input / output instructions are extremely simplified in ANNA (more on I/O later in the course).

# Ending the Program

- The `out` instruction can also be used to halt the program: simply use `r0` as the register.
  - The simulator will stop executing when an `out r0` instruction is executed.
- For readability purposes, the assembler directive `.halt` can (and should) be used instead of `out r0`.
  - The directive `.halt` has no operands – it is simply shorthand for an `out r0` instruction.

# Example Program

- Write a program that adds three numbers from input and displays the sum.

in r1

in r2

in r3

add r4 r1 r2

add r4 r4 r3

out r4

.halt

# Outline

- Introduction to Assembly Language
- Arithmetic and Logic Instructions
- Input / Output Instructions
- **Arithmetic Instructions with Immediates**
- Branches and Jumps
- Conditional Execution and Loops
- Memory Instructions
- Variables in Assembly Language
- Function Calls

# Immediates

- Some instructions have **immediates** — a constant value in an instruction.
- This allows assembly instructions to work with constant operands.
- For example, consider the C++ instruction:  
$$x = y + 3;$$

# addi

The add immediate instruction `addi` has the following format:

`addi Rd Rsl Imm6`                       $R(Rd) \leftarrow R(Rs_l) + Imm6$

- This instruction adds the contents of register  $Rs_l$  with the 6-bit immediate  $Imm6$ .
- What should be the range of the immediate ( $Imm6$ )?
  - The immediate is a signed two's complement number that can range from -32 to 31.

Examples:

`addi r3 r5 -12`                       $R(r3) \leftarrow R(r5) + -12$

`addi r2 r2 1`                       $R(r2) \leftarrow R(r2) + 1$  (increments `r2`)

# shf

The shift instruction has a similar format:

`shf Rd Rsl Imm6`

- This instruction shifts the contents of  $Rs_l$  and places the results in  $Rd$ .
- The number of the bits and direction of the shift is determined by the immediate:
  - If  $Imm6$  is positive  $\rightarrow$  shift the contents of  $Rs_l$  left by  $Imm6$  bits.
  - If  $Imm6$  is negative  $\rightarrow$  shift the content of  $Rs_l$  logically right by  $\text{abs}(Imm6)$  bits.

Examples:

`shf r3 r4 2`       $R(r3) \leftarrow R(r4) \ll 2$       (shift left 2 bits)

`shf r7 r6 -3`       $R(r7) \leftarrow R(r6) \gg 3$       (shift right 3 bits)

# Larger Immediates

- What if you want a larger immediate?
- One solution is have an instruction that initializes a register based on an immediate.
- Example: `li r3 12345`
- Unfortunately, there is a problem. What is the problem?



## lli / lui

The load register immediate instructions `lli` and `lui` are used in tandem to initialize registers.

The format for these instructions is:

<code>lli Rd Imm8</code>	load lower immediate
<code>lui Rd Imm8</code>	load upper immediate

# lli / lui

- The `lli` instruction is used to initialize the lower 8 bits (bits 7-0) of register  $Rd$ .
  - The lower 8 bits are copied verbatim from the immediate.
  - The upper 8 bits of register  $Rd$  are equal to bit 7 (sign bit) of the immediate (sign extension).
- The `lui` instruction is used to initialize the upper 8 bits (bits 15-8) of register  $Rd$ .
  - The 8 bits of the immediate are copied into the upper 8 bits of register  $Rd$ .
  - The lower 8 bits of register  $Rd$  are unchanged.

15	8	7	0
Upper half		Lower half	

# lli / lui

This results in the following consequences:

- Only an `lli` instruction is needed to initialize a register if it is 8 bits or less (-128 to 127).
  - Most immediates fall into the range.
- When executing `lli/lui` as a pair, the `lli` instruction must be executed before the `lui` instruction.
  - Why?

# Using lli / lui

In the ANNA assembler, there are three ways of specifying 8-bit immediates for `lui/lli`:

- signed 16 bit decimal number: The appropriate 8 bits are placed in the immediate field.
- 8 bit hexadecimal number: The 8 bits (two hex digits) represented by the hexadecimal number are placed in the immediate field.
- labels: Will be discussed later.

Example: These are the same (Note:  $1500 = 0x05dc$ )

```
lli r4 1500
```

```
lli r4 0xdc
```

```
lui r4 1500
```

```
lui r4 0x05
```

# Fetch and Execute Loop

Fetch and execute loop for immediate instructions:

1. Fetch the instruction at the offset in memory indicated by the PC.
2. Set  $PC \leftarrow PC + 1$ .
3. Get the value of the source register and/or immediate.
  - `lll i` does not get a value from a register
  - `lui` reads register  $Rd$  to get the lower 8 bits.
4. Perform the specified operation.
5. Place the result into the destination register.

# Conversion to Machine Language

There are two different instruction formats for instructions with immediates based on the size of the immediate:

- I6-type (`addi`, `shf`)

15	12	11	9	8	6	5	0
Opcode		$Rd$		$Rs_l$		$Imm6$	

- I8-type (`lll`, `lui`)

15	12	11	9	8	7	0
Opcode		$Rd$		$Imm8$		

As before the opcodes are four bits:

Opcode mnemonic	Opcode bits
<code>shf</code>	0101
<code>lll</code>	0110
<code>lui</code>	0111
<code>addi</code>	1100

# Class Problem

What is the machine code representation for each of the following instructions:

1. `lli r3 -13`

2. `lui r3 -13`

3. `addi r4 r5 -1`

# Outline

- Introduction to Assembly Language
- Arithmetic and Logic Instructions
- Input / Output Instructions
- Arithmetic Instructions with Immediates
- **Branches and Jumps**
- Conditional Execution and Loops
- Memory Instructions
- Variables in Assembly Language
- Function Calls



# Conditional Branches

- Conditional branches and jumps allow the programmer to jump to another portion of code.
  - In other words, they set the PC to something other than  $PC+1$ .
- Conditional branch instructions are based on a condition: if (*condition\_test*) go to *target\_address*
  - Only jumps to *target\_address* if *condition\_test* is true.
  - Continue to  $PC+1$  if false.

# bez / bgz

There are two conditional branch instructions in ANNA:

`bez  $Rd$   $Imm8$`  branch if equal to zero

`bgz  $Rd$   $Imm8$`  branch if greater than zero

- $Rd$  is used in the condition test
  - `bez`: Branches if the word in  $Rd$  is equal to zero.
  - `bgz`: Branches if the word in  $Rd$  is strictly greater than zero.
- $Imm8$  is used to compute the target address.
  - The *target address* is computed:  $PC + 1 + Imm8$ .
  - The immediate is an 8 bit signed number.

# PC-relative addressing

- **PC-relative addressing:** process of creating an address by using the current value of the PC.
- In essence, the offset is the number of instructions away you want to jump.
  - Negative offset: jump backwards
  - Positive offset: jump forwards
- Typical use for branches is for if-else statements and loops: the target address is usually not that far away.

# Converting to Machine Language

- Branches are I8-type instructions:
  - bez opcode bits: 1010
  - bgz opcode bits: 1011

# Fetch and Execute Loop

Fetch and execute loop for branch instructions:

1. Fetch the instruction at the offset in memory indicated by the PC.
2. Set  $PC \leftarrow PC + 1$ .
3. Get the value of the register.
4. Compare the value of the register to 0.
5. (if taken) Compute the target address.  $(PC + Imm8)$
6. (if taken) Update PC.

# Unconditional Branches

What if I need to use an unconditional branch?

# Big Jumps

- What is the max you can jump in either direction using bez and bgz?
  - $2^{(8-1)}$
- What if I need to jump farther than what the offset provides?



# Offsets

When I add/remove instructions from the program, it's a pain to change all of the offsets.





# Labels

Labels can optionally precede any instruction. Rules:

- Only one label per instruction (must be on same line).
- A label name can only be declared once.
- A label is a string followed by a colon (colon is not part of label).
- The label can be referenced in the immediate field of other instructions.
  - Cannot be used for I6-type instructions.
  - Must be preceded by ‘&’ sign.
  - The assembler replaces the label using the address of the instruction marked by the label.

# Labels for Branches

- When a label is used as an immediate for a branch instruction, the assembler will automatically determine the proper offset.
- Example:



- An error occurs if the address is too far away.
  - Error thrown if the difference is larger than the range of an 8-bit immediate

# Labels for Branches Example

lli r4 10

out r4

out r1

lli r3 1

LOOP: add r1 r1 r3

sub r5 r4 r1

bgz r5 &LOOP

out r1

.halt

# Outline

- Introduction to Assembly Language
- Arithmetic and Logic Instructions
- Input / Output Instructions
- Arithmetic Instructions with Immediates
- Branches and Jumps
- **Conditional Execution and Loops**
- Memory Instructions
- Variables in Assembly Language
- Function Calls

# Converting C++ into ANNA

- In this section, we will be discussing examples on how to convert if statements and loops written in C++ into ANNA assembly language.
- We will assume that `int` in C++ refers to a 16-bit integer.

# Comparing Two Numbers

How do we compare two numbers?




# If Statements

Convert this code to ANNA assembly.

Assume `x` is in `r3` and `y` is in `r4`.

```
int x, y;
if (x == y)
    x++;
else
    y++;
x = x - y;
```



# If Statements

Repeat the previous example but change the if statement to...

```
if (x >= y)
```



```
if ( (x == y) && (x > 0) )
```





# Loops

Convert this code to ANNA assembly. Assume `sum` is in `r1` and `i` is in `r2`.

```
sum = 0;
for (i = 0; i < 100; i++) {
    sum = sum + i;
}
```

- Similar to:

```
sum = 0
i = 0
while(i < 100){
    sum = sum + i;
    i++;
}
```

# Loops

Convert this code to ANNA assembly. Assume `sum` is in `r1` and `i` is in `r2`.

```
sum = 0;
for (i = 0; i < 100; i++) {
    sum = sum + i;
}
```

`lli r1 0` #stores the sum

`lli r2 0` #store the variable “i”

`lli r3 100` #the loop\_max

`LOOP: add r1 r1 r2` #sum = sum + i

`addi r2 r2 1` #i = i + 1

`sub r4 r3 r2` #r4 = loop\_max - i

`bgz r4 &LOOP` #checking whether loop should end or continue

`out r2` #print the final value of “i”

`out r1` #print final value of sum

`.halt`

# Class Problem

Convert the following code to ANNA.

Assume `a` is in `r2` and `b` is in `r3`.

```
int a = 1;  
int b = 1;  
do {  
    b = b | (b << 1);  
    a++;  
} while (a < 5);
```

# Outline

- Introduction to Assembly Language
- Arithmetic and Logic Instructions
- Input / Output Instructions
- Arithmetic Instructions with Immediates
- Branches and Jumps
- Conditional Execution and Loops
- **Memory Instructions**
- Variables in Assembly Language
- Function Calls

# lw

The load word `lw` instruction obtains a word from memory and places into a register:

$$\text{lw } Rd\ Rs_1\ Imm6 \qquad R(Rd) \leftarrow M[R(Rs_1) + Imm6]$$

- $Rd$  is the destination register that receives the word from memory.
- $Rs_1$  and  $Imm6$  are used to compute the effective address – the address of the word you want to obtain.

# Memory Addresses

- The **effective address** is computed by adding the contents of  $Rs_1$  with the immediate.
- Addresses in ANNA are unsigned. The value of register  $Rs_1$  is treated as an unsigned number.
- The immediate is signed and represents an offset from the address.
- If the resulting addition results in a negative address, the ANNA simulator will wraparound to a larger unsigned address.

## SW

The store word `SW` instruction writes a word to memory:

$$\text{SW } Rd \ Rs_1 \ Imm6 \qquad M[R(Rs_1) + Imm6] \leftarrow R(Rd)$$

- $Rd$  is the register that contains the data to store.
- $Rs_1$  and  $Imm6$  are used to compute the effective address (same computation as a load).

# Memory Instruction Notes

- ANNA is a *load/store architecture*. Only lw and sw instructions are allowed to access memory directly.
- Both instructions are I6-type instructions:
  - lw opcode bits: 1000
  - sw opcode bits: 1001



# Fetch and Execute Loop

Fetch and execute loop for memory instructions:

1. Fetch the instruction at the offset in memory indicated by the PC.
2. Set  $PC \leftarrow PC + 1$ .
3. Compute the effective address.
4. (SW only): Get the value of the data register.
5. Access memory.
6. (LW only): Place the result into the destination register.

# Outline

- Introduction to Assembly Language
- Arithmetic and Logic Instructions
- Input / Output Instructions
- Arithmetic Instructions with Immediates
- Branches and Jumps
- Conditional Execution and Loops
- Memory Instructions
- **Variables in Assembly Language**
- Function Calls

# Variables in Assembly Language

- Variables are often stored in memory as there are not enough registers to store all variables.
- However, variables must be placed in registers before they can be used in operations. One possible, yet very slow, method:
  1. Place the addresses of the desired variables into registers.
  2. Load the operands into registers.
  3. Perform the desired operation.
  4. Store the result into memory.
- If a variable is used frequently, it is better to keep in a register for performance purposes. Register allocation is a key responsibility carried out by the compiler.

# Variables in Assembly Language

## Example

- Translate  $c = a - b;$  into ANNA assembly. Assume  $a$  is at address  $0x1000$ ,  $b$  is at address  $0xac23$ , and  $c$  is at address  $0x78ac$ .

# .fill

The assembler directive `.fill` tells the assembler to fill in the next word in memory with the entire 16 bit immediate value:

`.fill Imm16`

- *Imm16* is a signed 16 bit immediate.
- Immediate can be either a signed immediate or a four digit hexadecimal number.

# Initializing Variables

- The `.fill` directive is used to initialize variables:
  - A label is used to identify the variable (often the same name as the variable).
  - `.fill` is used to initialize the value of the variable.
- Example: Translate the variable declaration  
`int count = 5;` into ANNA assembly:

# Using Labels for Addresses

- Once labels are used to declare variables, they can be used in `lui` and `lli` instructions.
  - The immediate corresponds to the *address* of the word defined by the label.
- Example (assume `count` is at address `0x5678`):  
`lli r4 &count` is identical to `lli r4 0x78`  
`lui r4 &count` is identical to `lui r4 0x56`
- Useful so programmers do not have to keep track of the addresses of the variables they declare.

# Memory Sections

- For assembly programs that require memory beyond registers, it is useful to divide the assembly program into a code section and a data section.
- Real programs, from a memory perspective, consists of four sections:
  - *Code*: Machine code instructions
  - *Data*: Global variables / constants used by the program
  - *Stack*: Stores activation records (which store parameters and local variables)
  - *Heap*: Memory dynamically allocated by the program (using `new`)



# Example

Write code that initializes `x` to 25, then executes `y = x + 2;`

# code section

```
lli r1 &x
```

```
lui r1 &x    # r1 = &x
```

```
lli r2 &y
```

```
lui r2 &y    # r2 = &y
```

```
lw  r7 r1 0  # r7 = x
```

```
addi r7 r7 2  # r7 = x + 2
```

```
sw  r7 r2 0  # y = x + 2
```

```
.halt
```

# data section

```
x: .fill 25
```

```
y: .fill 0
```

# Class Problem

Write an assembly sequence to reflect for this code segment:

```
x = 5;
```

```
y = 52;
```

```
x = x + (y >> 2); // >> is right shift
```

# Pointers

- **Pointers** store addresses of other variables.
  - We have already seen registers hold addresses.
- Example: Convert `p = &x;` into ANNA assembly. Assume that `x` and `p` are declared using labels in the data section.

# Pointer Dereference Example

Convert `*p = 3;` into ANNA assembly.

# Dynamic Memory

Pointers are often used to point to dynamically-allocated memory. This can be accomplished in ANNA assembly language:

- Memory is set aside for heap data.
- A small chunk of this memory is used as a table (or logically equivalent data structure) that keeps track of which memory is used and which memory is available.
- Underlying implementation for new and delete manipulate this table.
- Different algorithms / data structures can be used to managing dynamic memory. These are discussed in CPSC 3400.

# Arrays

Arrays are sequentially laid out in memory:

# Declaring Arrays

Convert `int list[5];` into ANNA assembly.

# Referencing Arrays

Convert `list[3] = 10;` into ANNA assembly.



# Base + Offset Addressing

- The form of addressing used in the previous example is called **base + offset**:
  - The register holds the base (start of the array)
  - The immediate holds the offset (index of the array)
- This is effective for arrays with small constant indices.

# Referencing Arrays

Convert `y = list[x];` into ANNA assembly.  
Assume that `x` is in `r5` and `y` is in `r6`.

# Multidimensional Arrays

Consider a two dimensional array: `int a[3][4];`

- Number of rows: 3
- Number of columns: 4

<code>a[0][0]</code>	<code>a[0][1]</code>	<code>a[0][2]</code>	<code>a[0][3]</code>
<code>a[1][0]</code>	<code>a[1][1]</code>	<code>a[1][2]</code>	<code>a[1][3]</code>
<code>a[2][0]</code>	<code>a[2][1]</code>	<code>a[2][2]</code>	<code>a[2][3]</code>

# Multidimensional Arrays

Since memory is modeled using a one-dimensional array, the 2D array gets flattened.

*row-major order*: rows are stored sequentially (used by C++)

a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[2][0]	a[2][1]	a[2][2]	a[2][3]
---------	---------	---------	---------	---------	---------	---------	---------	---------	---------	---------	---------

*column-major order*: columns are stored sequentially

a[0][0]	a[1][0]	a[2][0]	a[0][1]	a[1][1]	a[2][1]	a[0][2]	a[1][2]	a[2][2]	a[0][3]	a[1][3]	a[2][3]
---------	---------	---------	---------	---------	---------	---------	---------	---------	---------	---------	---------

# Multidimensional Array Addresses

How do you determine the address of a `[row] [col]`?

Row major order:

Column major order:

# Outline

- Introduction to Assembly Language
- Arithmetic and Logic Instructions
- Input / Output Instructions
- Arithmetic Instructions with Immediates
- Branches and Jumps
- Conditional Execution and Loops
- Memory Instructions
- Variables in Assembly Language
- **Function Calls**

# j alr

The final ANNA assembly instruction `j alr` (jump and link register):

`j alr Rd Rsl`

- Jumps to an address stored in a register and saves  $PC + 1$  in a different register.
- $Rd$  contains the address of the function to jump to.
- $Rs_l$  will contain the return address ( $PC + 1$ ).

Operation summary for `j alr`:

$R(Rs_l) \leftarrow PC + 1$

$PC \leftarrow R(Rd)$

# More on jalr

- It is used predominantly in function calls.
- Uses direct addressing (not PC-relative).
- R-type instruction ( $Rs_2$  ignored)
- Opcode bits: 1101



# Fetch and Execute Loop

Fetch and execute loop for jump and link register instruction:

1. Fetch the instruction at the offset in memory indicated by the PC.
2. Set  $PC \leftarrow PC + 1$ .
3. Store PC into register  $Rs_1$ .
4. Get the value of the register of  $Rd$ .
5. Update PC.

# Using jalr in function calls

To call the function `foo`:

To return from `foo`:

# Function Call Caveats

- No parameters were passed into `foo`.
- The function `foo` did not return a value.
- It is assumed that `foo` did not overwrite the return address stored in `r2`.
- All registers and memory locations are "global" in that they are shared by all functions.
  - A problem occurs if the calling function has designated that `x` is stored in `r3` and then `foo` overwrites `r3`.

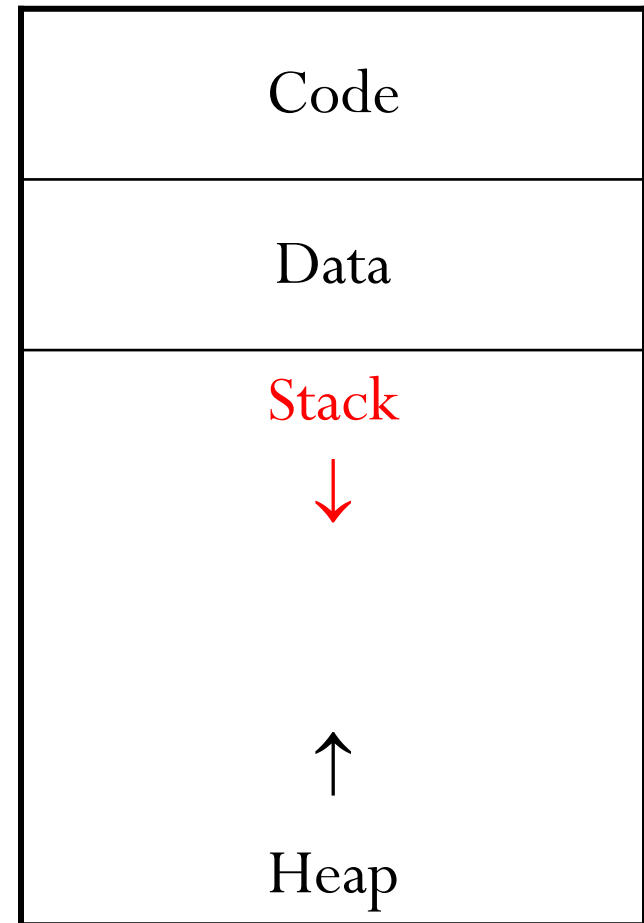
# Structs and Classes

- Structs and classes are typically laid out in the order the data members are presented.
  - Padding (extra space) may be necessary to address any alignment restrictions.
- A base + offset addressing scheme is used to access data members.
  - The base register is the start of the object.
  - The compiler knows the offset of each data member.
- Classes have some additional data:
  - Additional data to keep track of type information (for inheritance, dynamic binding).
  - Pointer to a table that has starting points for each member function (is shared among all objects of a class).

# Program Stack

The **program stack** keeps track of function calls.

- On function calls, an activation record is pushed onto the stack.
- On function return, an activation record is popped off the stack.



# Activation Records

An **activation record** can contain the following:

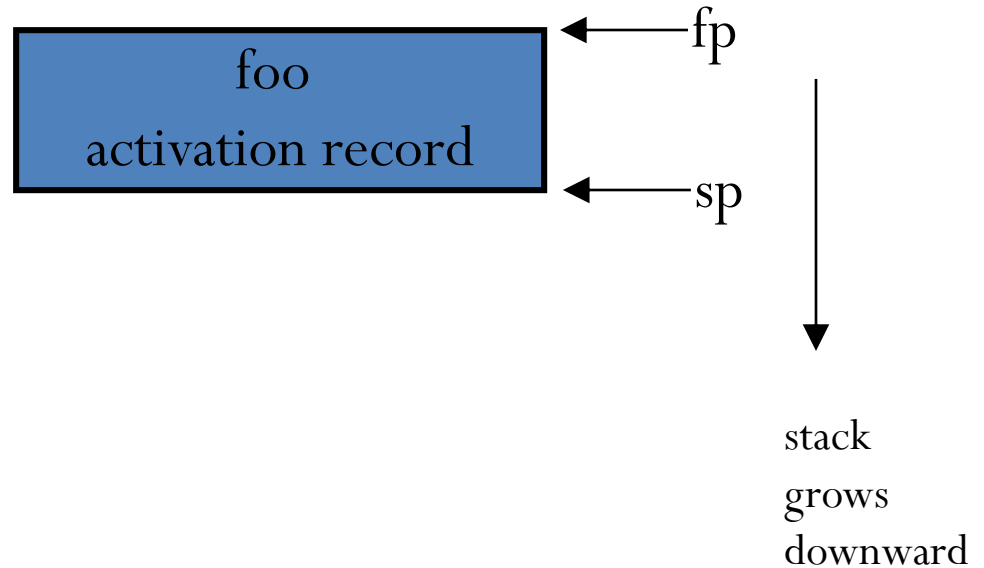
- Parameters passed into the function
- Local variables
- Return address - point in the code where the called function needs to return to when completed
- Saved values from previous functions (restored at the end of the function)
- Additional temporary values for long expressions

# Stack and Frame Pointers

- Two pointers are commonly used to keep track of the stack:
  - The *stack pointer* points to the top of the stack.
  - The *frame pointer* points to the start of the current activation record.
- These pointers are typically stored in registers.
  - Some instruction sets will have special registers designated for these pointers.

# Program Stack Example

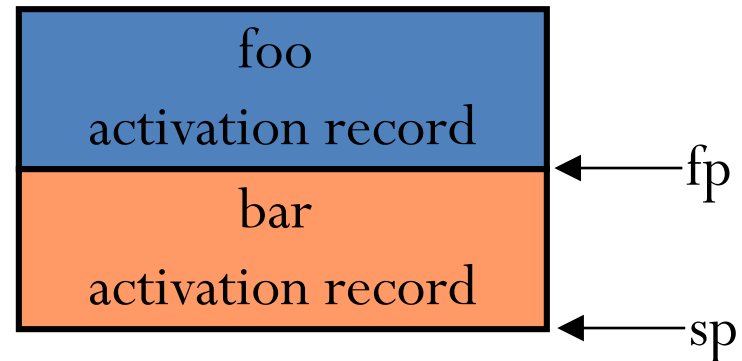
```
void foo()  
{  
    int x, y[2];  
    bar(x);  
}  
void bar(int x)  
{  
    int a[3];  
    sort();  
}
```





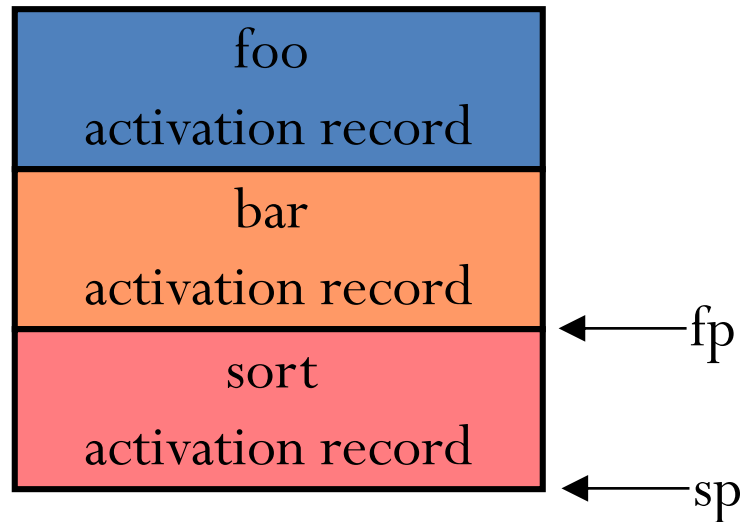
# Program Stack Example

```
void foo()  
{  
    int x, y[2];  
    bar(x);  
}  
void bar(int x)  
{  
    int a[3];  
    sort();  
}
```



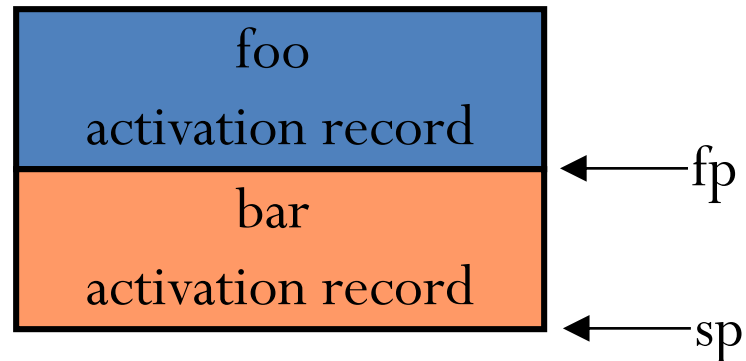
# Program Stack Example

```
void foo()  
{  
    int x, y[2];  
    bar(x);  
}  
void bar(int x)  
{  
    int a[3];  
    sort();  
}
```



# Program Stack Example

```
void foo()  
{  
    int x, y[2];  
    bar(x);  
}  
void bar(int x)  
{  
    int a[3];  
    sort();  
}
```



# Program Stack Example

```
void foo()  
{  
    int x, y[2];  
    bar(x);  
}  
void bar(int x)  
{  
    int a[3];  
    sort();  
}
```



# Activation Record Example

```
int bar(int a, int b)
{
    int c;
    c = a + b;
    return c + 2;
}

int foo(int x, int y, int
z)
{
    int sum1;
    int sum2;
    sum1 = bar(x, y);
    sum2 = bar(y, z);
    return sum1 + sum2;
}
```

```
int main()
{
    int a = 2;
    int b = 3;
    int c = 4;
    int total;
    total = foo(a, b,
c);
    total = total +
bar(b, c);
    cout << "total: "
<< total;
    return 0;
}
```

# Activation Record Example

Step 1  
Call to foo(a, b, c)

main	a = 2
	b = 3
	c = 4
	total = ?
fp →	x = 2
	y = 3
	z = 4
foo	return addr
	old fp (main)
	sum 1 = ?
	sum 2 = ?
sp →	

Step 2  
Call to bar(x, y)

main	a = 2
	b = 3
	c = 4
	total = ?
foo	x = 2
	y = 3
	z = 4
	return addr
foo	old fp (main)
	sum 1 = ?
	sum 2 = ?
fp →	a = 2
	b = 3
bar	return addr
	old fp (foo)
	c = ?
sp →	

Step 3  
Return from bar(x, y)

main	a = 2
	b = 3
	c = 4
	total = ?
fp →	x = 2
	y = 3
	z = 4
foo	return addr
	old fp (main)
	sum 1 = 7
	sum 2 = ?
sp →	

Step 4  
Call to bar(y, z)

main	a = 2
	b = 3
	c = 4
	total = ?
foo	x = 2
	y = 3
	z = 4
	return addr
foo	old fp (main)
	sum 1 = 7
	sum 2 = ?
fp →	a = 3
	b = 4
bar	return addr
	old fp (foo)
	c = ?
sp →	

# Activation Record Example

Step 5  
Return from bar(y, z)

main	a = 2
	b = 3
	c = 4
	total = ?
fp →	x = 2
foo	y = 3
	z = 4
	return addr
	old fp (main)
sp →	sum 1 = 7
	sum 2 = 9

Step 6  
Return from foo(a, b, c)

fp →	a = 2
main	b = 3
	c = 4
sp →	total = 16

Step 7  
Call to bar(b,c)

main	a = 2
	b = 3
	c = 4
	total = 16
fp →	a = 3
bar	b = 4
	return addr
	old fp (main)
sp →	c = ?

Step 8  
Return from bar(b,c)

fp →	a = 2
main	b = 3
	c = 4
sp →	total = 25

# Function Calls and Assembly Language

- With the program stack, each function now has their own section of memory for local variables.
- However, registers remain "global" and shared across the functions.
  - Can save and restore registers in the activation record.
- Assembly language code is responsible for maintaining the program stack and its contents.
- Base + offset addressing can be used with to access parameters and local variables in the activation record.



# Function Calls and Performance

- Since there is significant work in manipulating the stack during function calls and returns, function calls are slow.
- If performance is a concern, programmers may try to inline function calls by inserting the equivalent code into the function instead of incurring the overhead of performing the call and return.
  - This comes at an expense of readability, modularity, and maintainability.
- Some compilers will attempt to inline small functions automatically.
  - In C++, the keyword `inline` can be used to tell the compiler that you want a function to be inlined. The request is non-binding – the compiler may or may not honor the request.

**Thank You!**