CS301 Computer Architecture

DR GAYATHRI ANANTHANARAYANAN

gayathri@iitdh.ac.in

Materials in these slides are borrowed from textbooks and existing Architecture courses

What is Assembly Language

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

Why learn Assembly Language ?

* 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

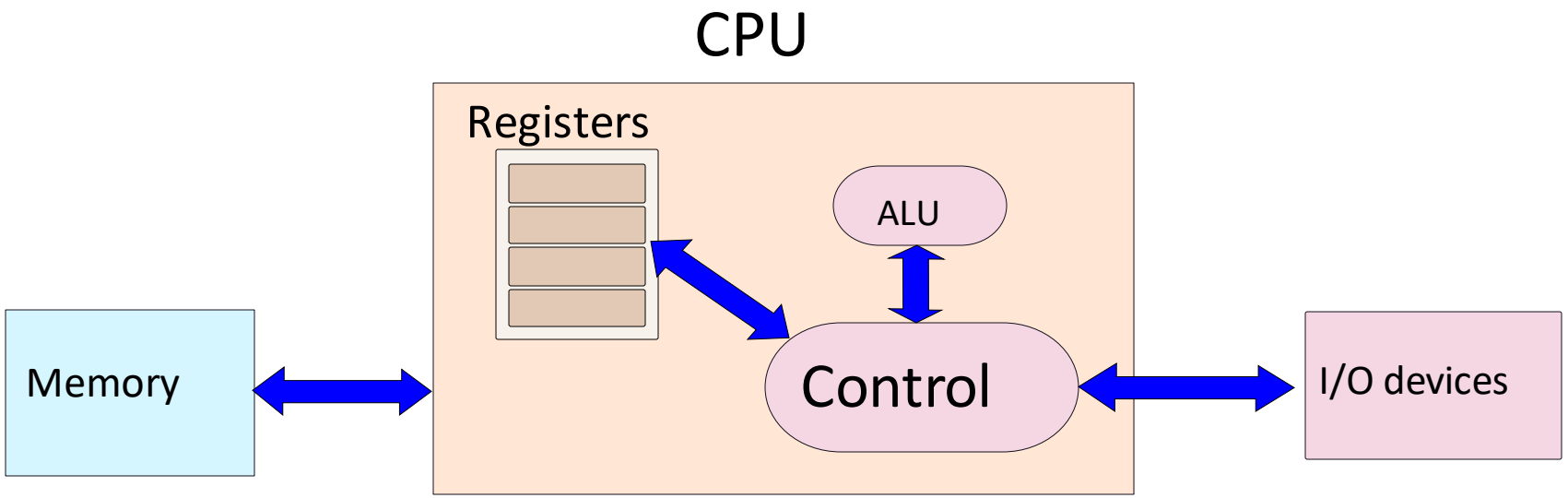
Assemblers

* Assemblers are programs that convert programs written in low level languages to machine code (0s and 1s)
* Examples :
  * nasm, tasm, and masm for x86 ISAs
  * 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)

Hardware Designers Perspective

* Learning the assembly language is the same as learning the intricacies of the instruction set
* Tells HW designers : what to build ?

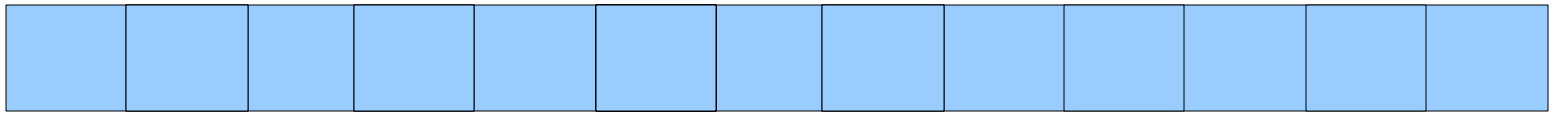
Machine Model – Von Neumann Machine with Registers



View of Registers

* Registers → named storage locations
  * in ARM : r0, r1, … 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

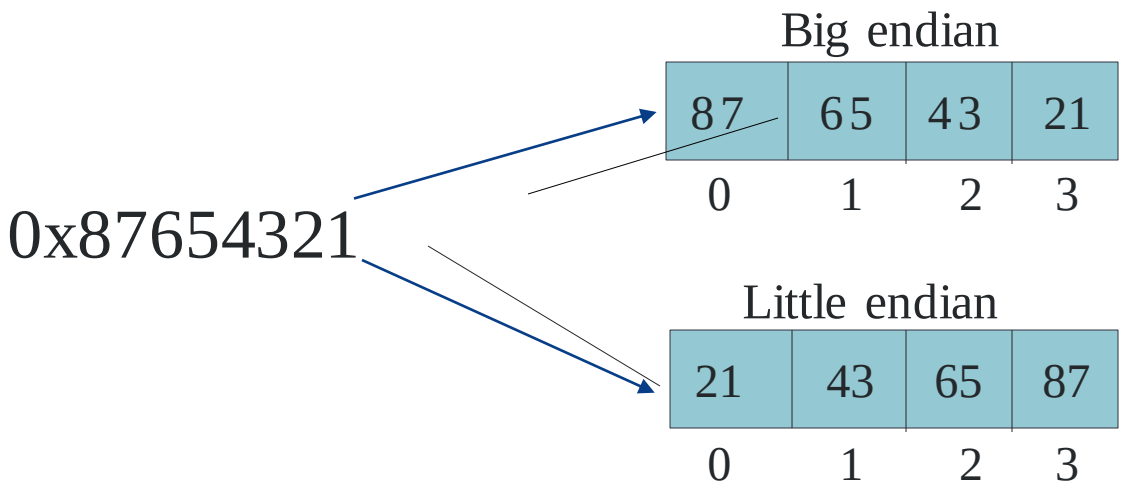


* Memory
  * One large array of bytes
  * Each location has an address
  * The address of the first location is 0, and increases by 1 for each subsequent location
* The program is stored in a part of the memory
* The program counter contains the address of the current instruction

Storage of Data in Memory

* Data Types
  * char (1 byte), short (2 bytes), int (4 bytes), long int (8 bytes)
* How are multibyte variables stored in memory ?
  * Example : How is a 4 byte integer stored ?
  * Save the 4 bytes in consecutive locations
  * Little endian representation (used in ARM and x86) → The LSB is stored in the lowest location
  * Big endian representation (Sun Sparc, IBM PPC) → The MSB is stored in the lowest location

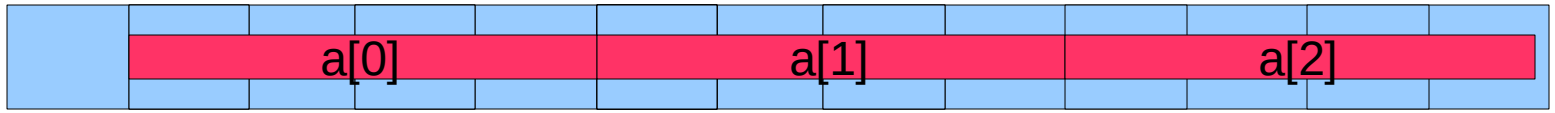
Little Endian vs Big Endian



* Note the order of the storage of bytes

Storage of Arrays in Memory

* Single dimensional arrays. Consider an array of integers : a[100]



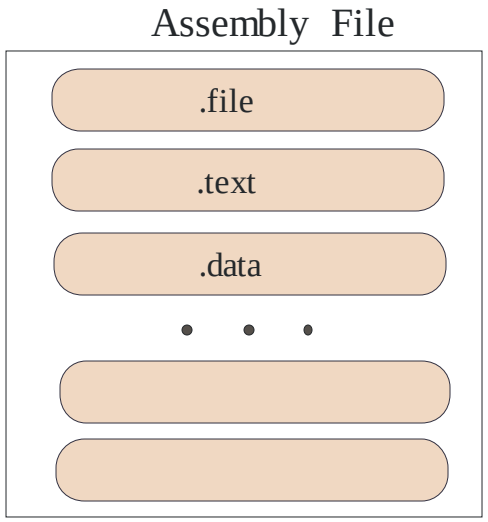
* Each integer is stored in either a little endian or big endian format
* 2 dimensional arrays :
  * int a[100][100]
  * float b[100][100]
  * Two methods : row major and column major

Mc Graw Hill Education

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 File Structure : GNU Assembler



* 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 a Statement

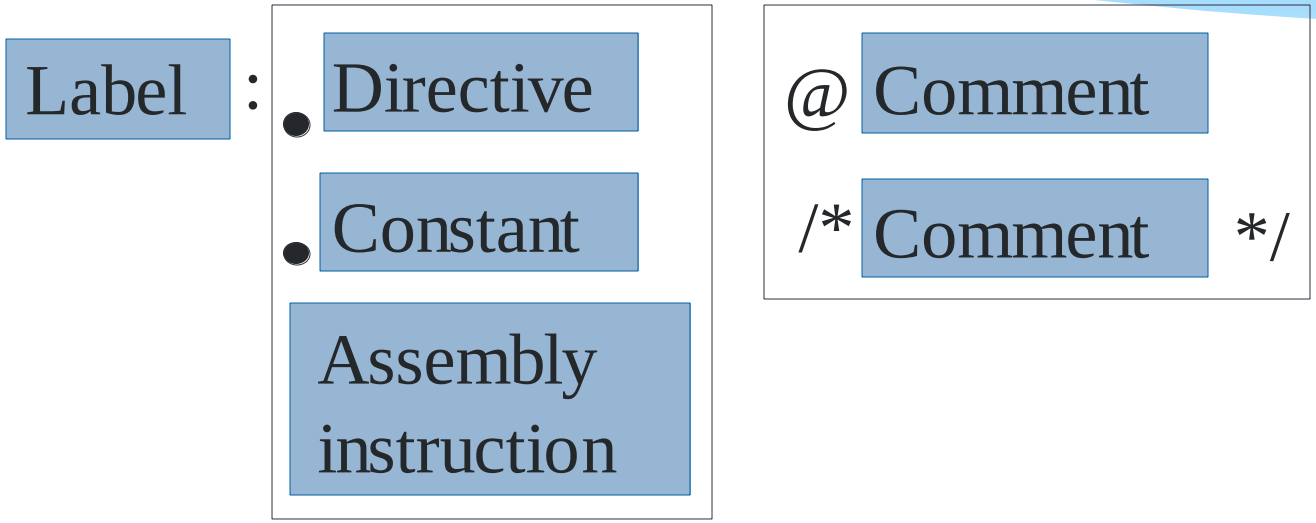
* instruction
  * textual identifier of a machine instruction
* operand
  * constant (also known as an immediate)
  * register
  * memory location

Examples of Instructions

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

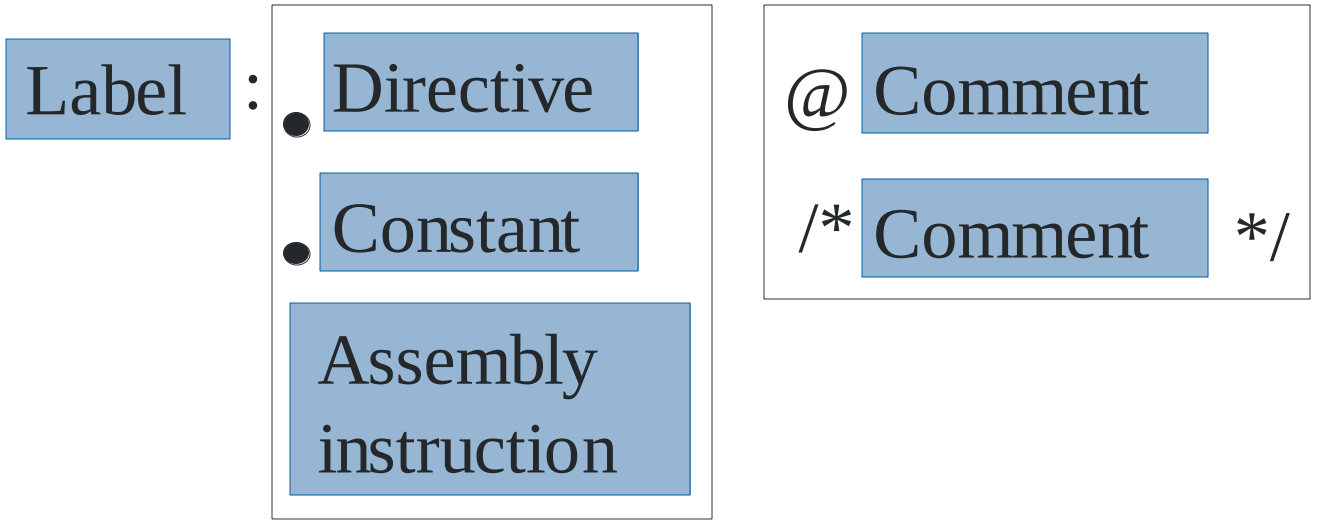
* subtract the contents of *r2* from the contents of *r1*, and save the result in *r3*
* multiply the contents of *r2* with the contents of *r1*, and save the results in *r3*

Generic Statement Structure



* label → identifier of a statement
* directive → tells the assembler to do something like declare a function
* constant → declares a constant

Generic Statement Structure - II



* 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

Nature of Operands

* Classification of instructions
  * If an instruction takes n operands, then it is said to be in the n-address format
  * Example : add r1, r2, r3 (3 address format)
* Addressing Mode
  * The method of specifying and accessing an operand in an assembly statement is known as the addressing mode.

Register Transfer Notation

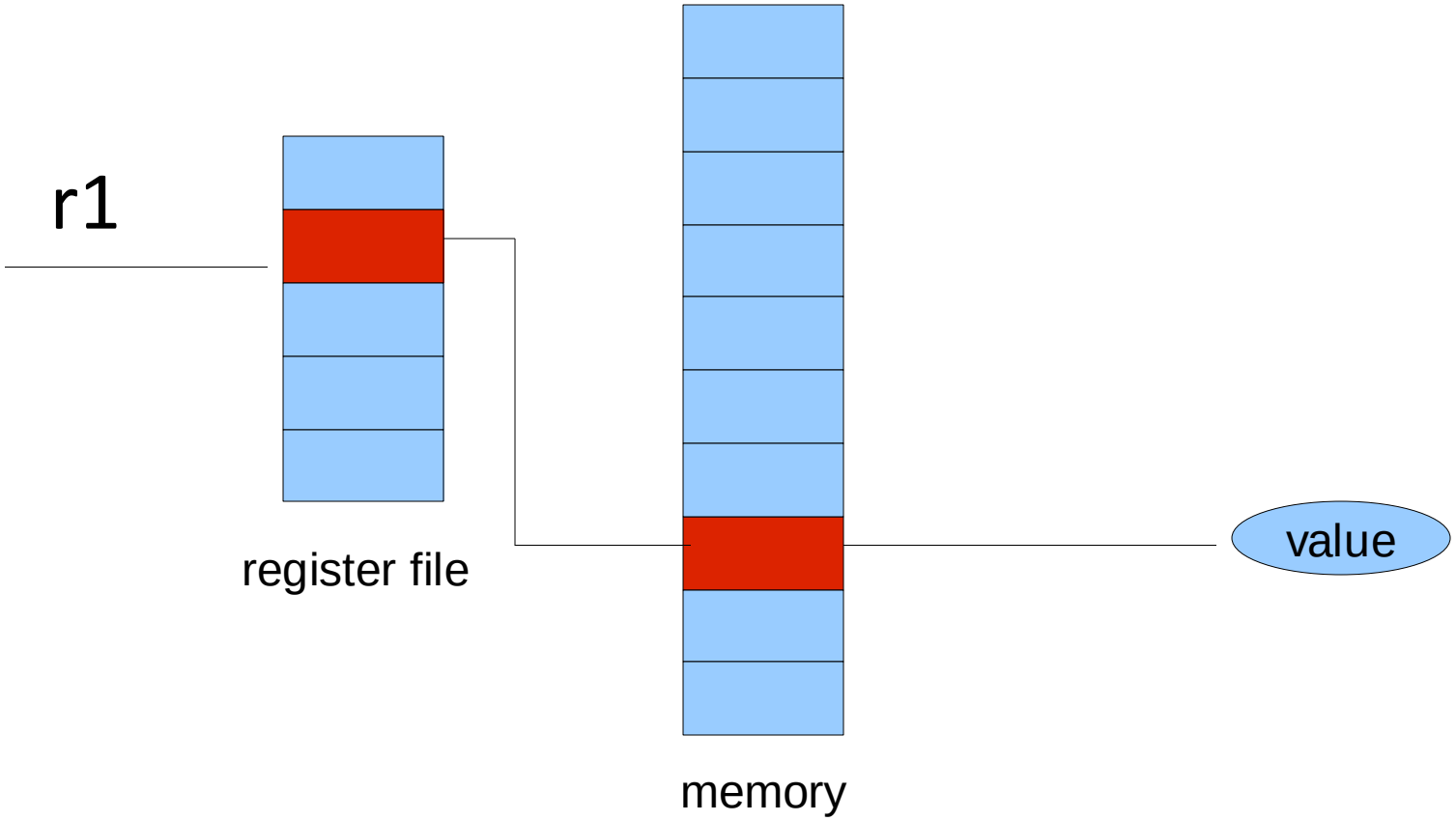
* This notation allows us to specify the semantics of instructions
* r1 ← r2
  * transfer the contents of register r2 to register r1
* r1 ← r2 + 4
  * add 4 to the contents of register r2, and transfer the contents to register r1
* r1 ← [r2]
  * access the memory location that matches the contents of r2, and store the data in register r1

Addressing Modes

* Let *V* be the value of an operand, and let r1, r2 specify registers
* Immediate addressing mode
  * V ← imm , e.g. 4, 8, 0x13, -3
* Register direct addressing mode
  * V ← r1
  * e.g. r1, r2, r3 …
* Register indirect
  * V ← [r1]
* Base-offset : V ← [r1 + offset], e.g. 20[r1] (V ← [20+r1])

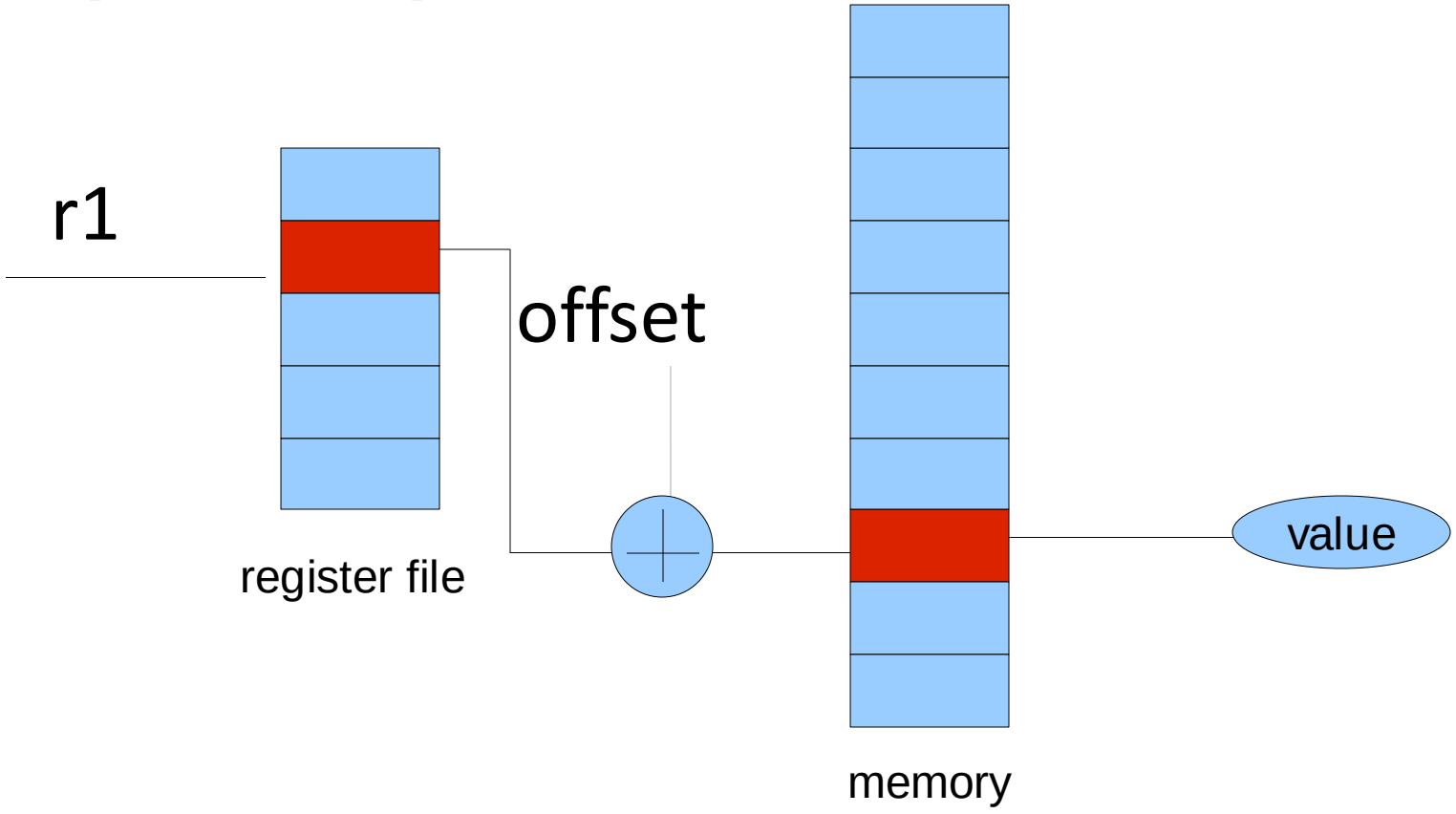
Register Indirect Mode

* V ← [r1]



Base-offset Addressing Mode

* V ← [r1+offset]



Addressing Modes - II

* Base-index-offset
  * V ← [r1 + r2 + offset]
  * example: 100[r1,r2] (V ← [r1 + r2 + 100])
* Memory Direct
  * V ← [addr]
  * example : [0x12ABCD03]
* PC Relative
  * V ← [pc + offset]
  * example: 100[pc] (V ← [pc + 100])

Base-Index-Offset Addressing Mode

* V ← [r1+r2 +offset]