Assembly Language & ISA

Slide courtesy: Smruti Ranjan Sarangi

Slides adapted by: Dr Sparsh Mittal

Features of an ISA (instr. Set architecture)

* Example of instructions in an ISA
  * Arithmetic instructions : add, sub, mul, div
  * Logical instructions : and, or, not
  * Data transfer/movement instructions
* Complete
  * It should be able to implement all the programs that users may write.

Features of an ISA – II

- Concise
  - The instruction set should have a limited size. Typically an ISA contains 32-1000 instructions.
- Generic
  - Instructions should not be too specialized, e.g. add14 (adds a number with 14) instruction is too specialized
- Simple
  - Should not be very complicated.

Designing an ISA

* Important questions that need to be answered :
  * How many instructions should we have ?
  * What should they do ?
  * How complicated should they be ?

Two different paradigms : RISC and CISC

RISC
(Reduced Instruction Set Computer)

CISC
(Complex Instruction Set Computer)

RISC vs CISC

A reduced instruction set computer (RISC) implements simple instructions that have a simple and regular structure. The number of instructions is typically a small number (64 to 128). Examples: ARM, IBM PowerPC, HP PA-RISC

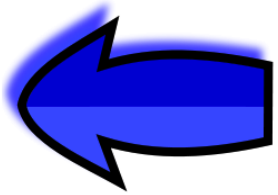
A complex instruction set computer (CISC) implements complex instructions that are highly irregular, take multiple operands, and implement complex functionalities. Secondly, the number of instructions is large (typically 500+). Examples: Intel x86, VAX

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

Outline

* Overview of Assembly Language
* Assembly Language Syntax
* SimpleRisc ISA



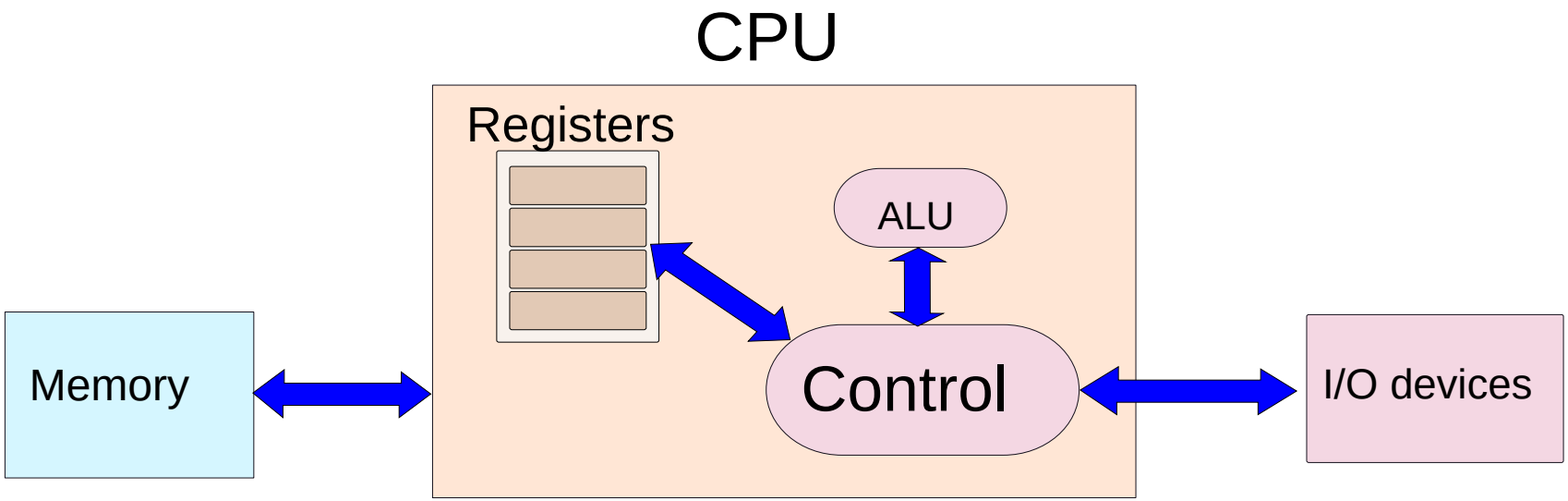
What is Assembly Language

* 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.
* 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.

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)

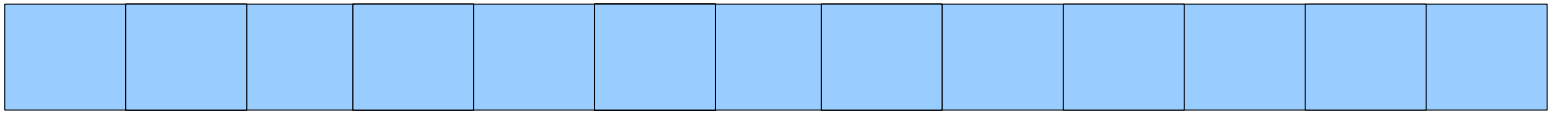
Machine Model



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

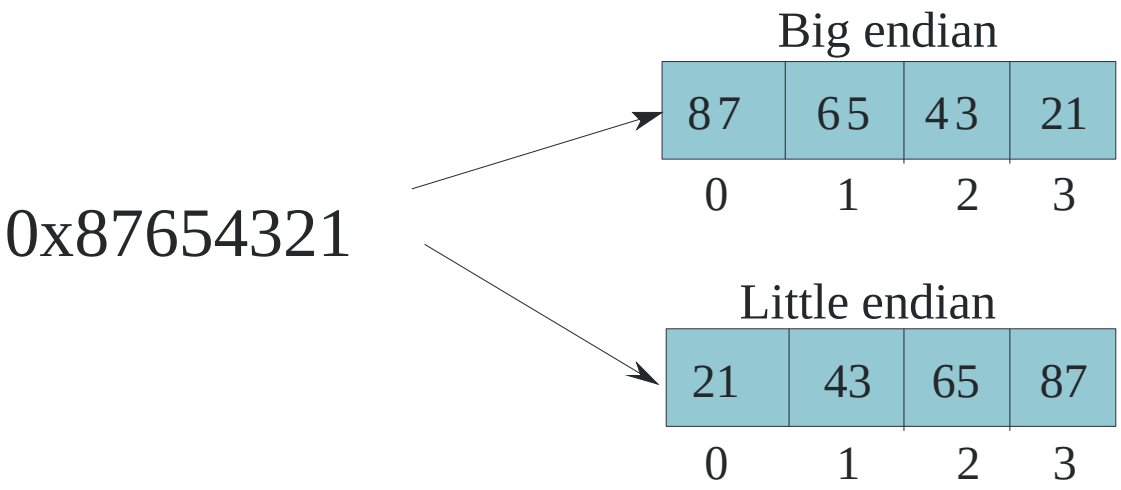


* Memory
  * One large array of bytes
  * Each location has an address
  * The address of the first location is 0, and increases by 1 for each subsequent location
* The program is stored in a part of the memory
* The program counter contains the address of the current instruction

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

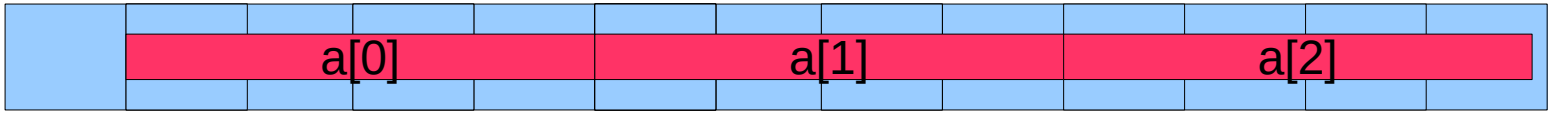
Little Endian vs Big Endian



* Note the order of the storage of bytes

Storage of Arrays in Memory

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



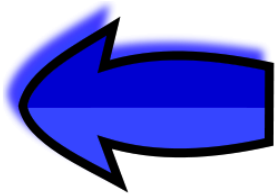
* Each integer is stored in either a little endian or big endian format
* 2 dimensional arrays :
  * int a[100][100]
  * float b[100][100]
  * Two methods : row major and column major

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

Outline

* Overview of Assembly Language
* Assembly Language Syntax
* SimpleRisc ISA



Structure of a Statement

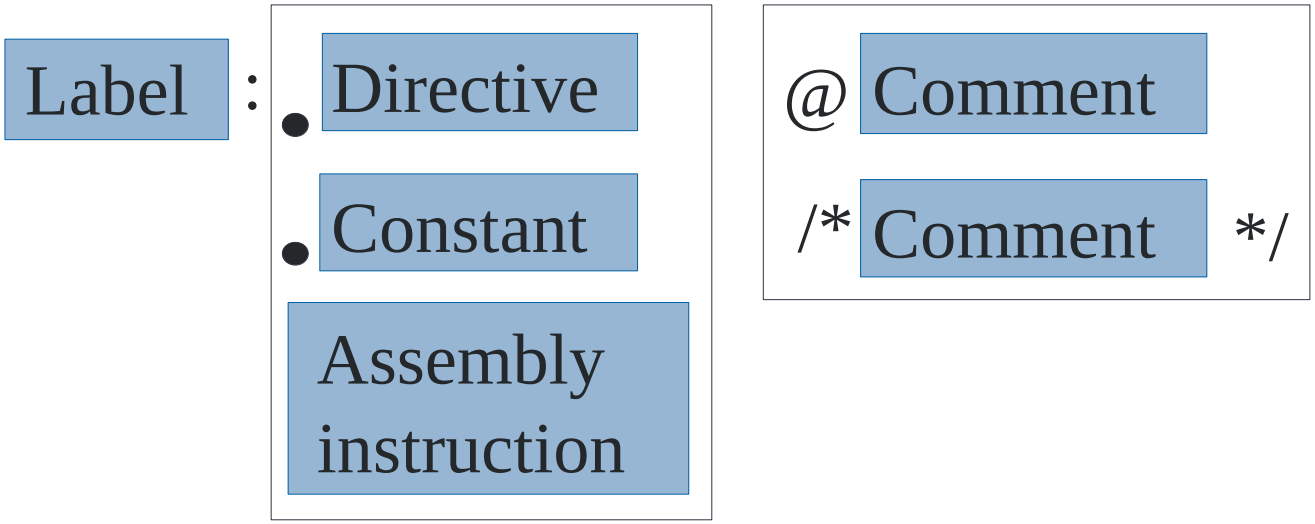
- instruction
  - Name of a machine instruction
- operand
  - constant (also known as an immediate)
  - register
  - memory location

Examples of Instructions

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

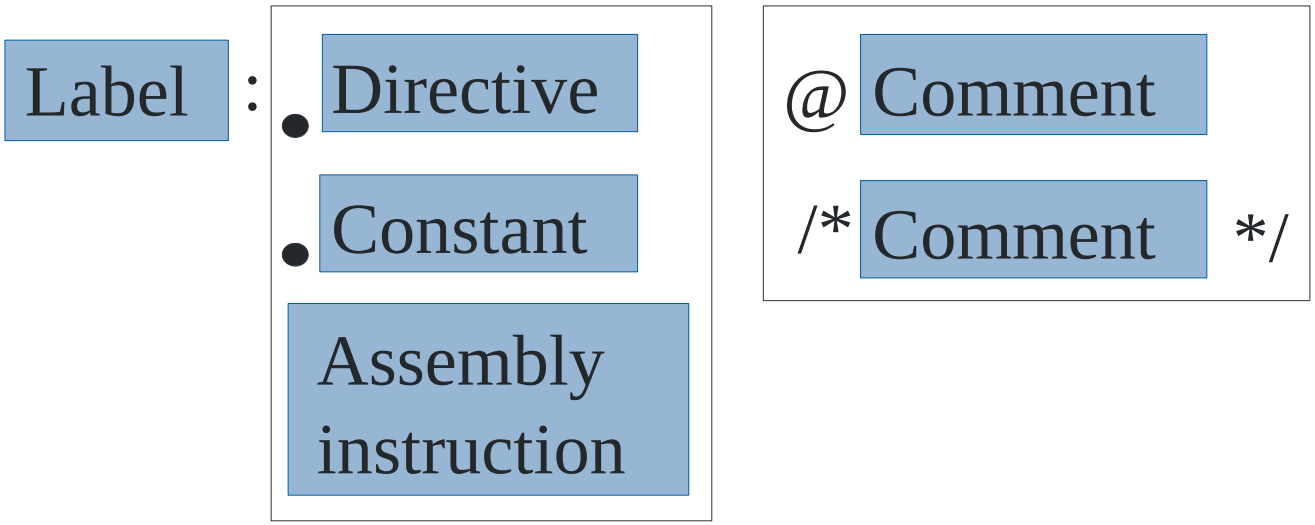
* subtract the contents of *r2* from the contents of *r1*, and save the result in *r3*
* multiply the contents of *r2* with the contents of *r1*, and save the results in *r3*

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

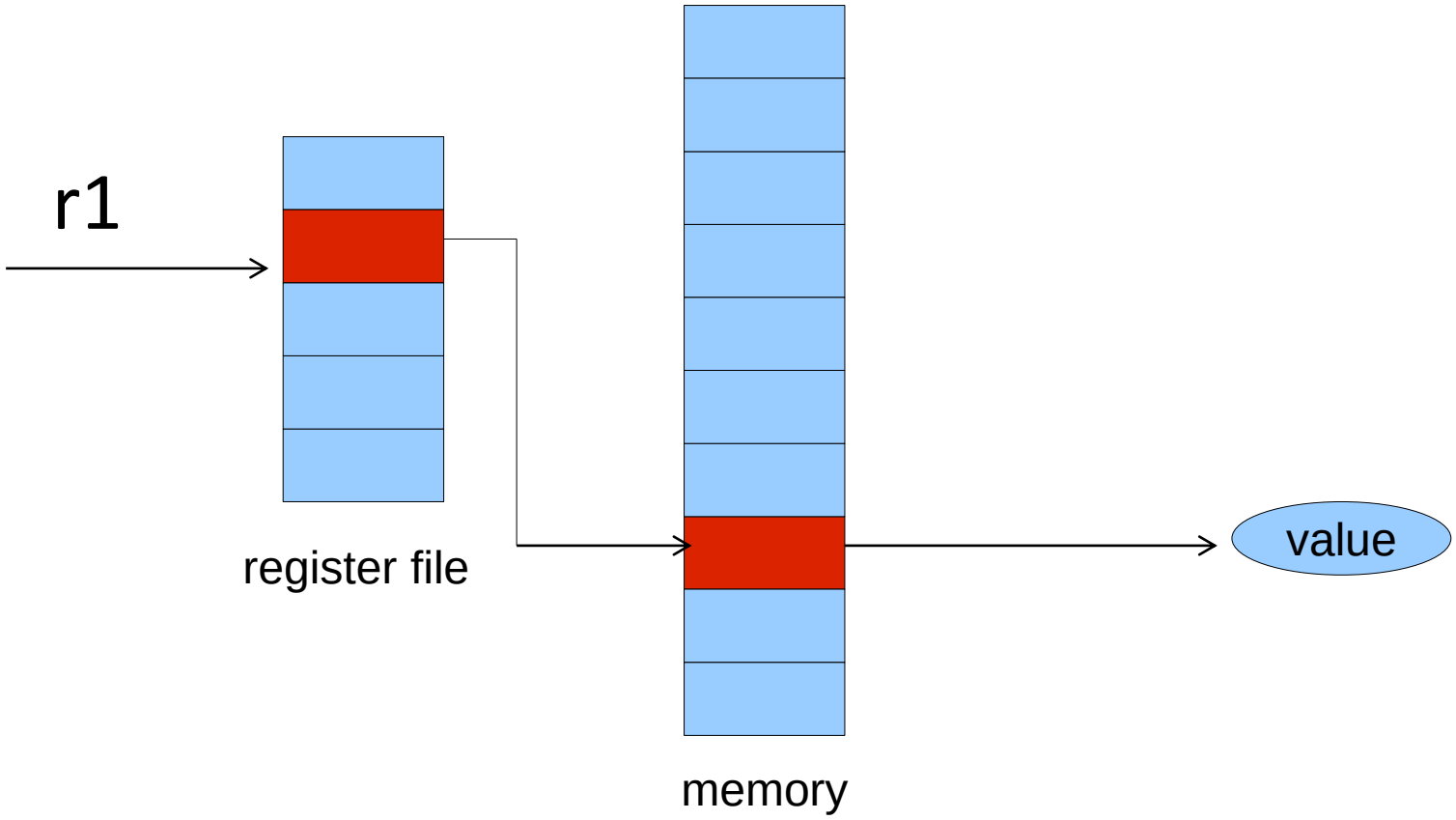
- This notation allows us to specify the semantics of instructions
- r1 ← r2
  - transfer the contents of register r2 to register r1
- r1 ← r2 + 4
  - add 4 to the contents of register r2, and transfer the contents to register r1
- r1 ← [r2]
  - access the memory location that matches the contents of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

Addressing Modes

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

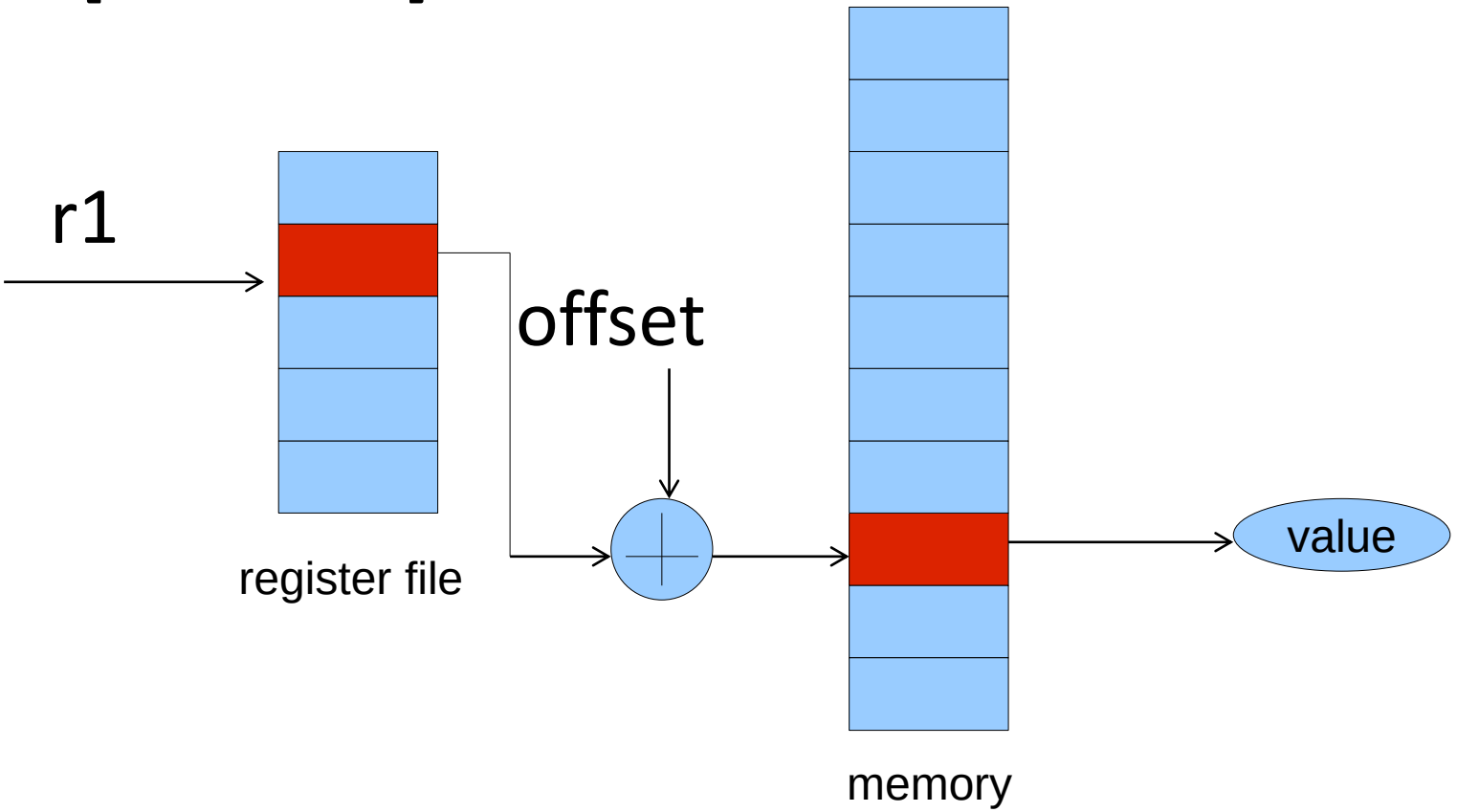
Register Indirect Mode

* V ← [r1]



Base-offset Addressing Mode

* V ← [r1+offset]

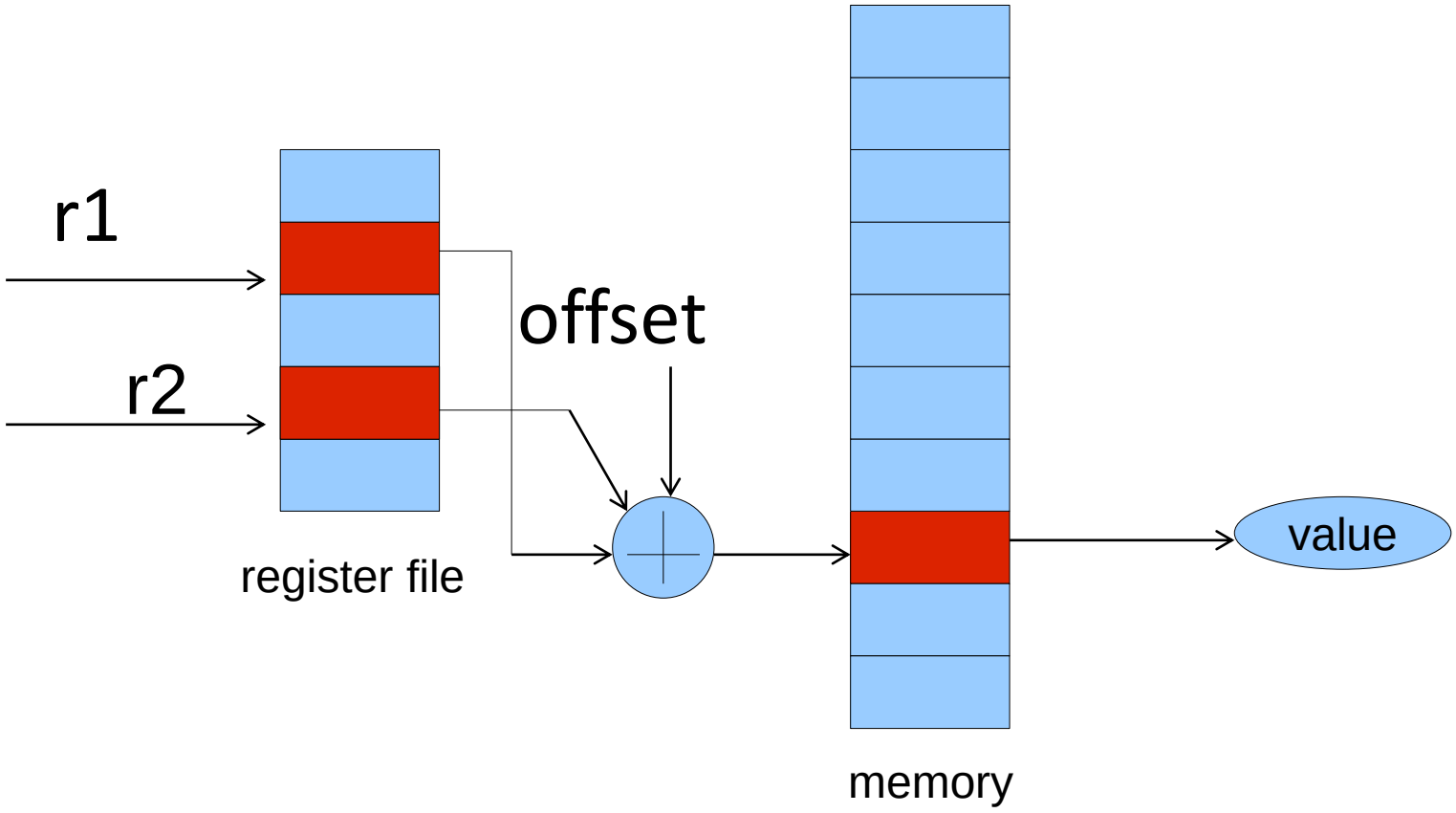


Addressing Modes - II

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

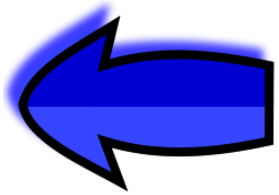
Base-Index-Offset Addressing Mode

* V ← [r1+r2 +offset]



Outline

* Overview of Assembly Language
* Assembly Language Syntax
* SimpleRisc ISA



SimpleRisc

* Simple RISC ISA
* Contains only 21 instructions
* We will design an assembly language for SimpleRisc
* Design a simple binary encoding,
* and then implement it ...

Registers

- SimpleRisc has 16 registers
  - Numbered : r0 … 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
  - One large array of bytes
- Special flags register → contains the result of the last comparison
  - flags.E = 1 (equality), flags.GT = 1 (greater than)

nop instruction

* nop → does nothing
* Example : nop

mov instruction

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 -2^{15} to $2^{15} - 1$

Arithmetic/Logical Instructions

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

Example	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	$r1 \leftarrow r2 \times r3$
div r1, r2, r3	$r1 \leftarrow r2/r3$ (quotient)
mod r1, r2, r3	$r1 \leftarrow r2$ *mod* $r3$ (remainder)
cmp r1, r2	set flags

Examples of Arithmetic Instructions

* Convert the following code to assembly

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

* Assign the variables to registers
  * a ← r0, b ← r1, c ← r2, d ← r3

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

Examples - II

* Convert the following code to assembly

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

* Assign the variables to registers
  * a ← r0, b ← r1, c ← r2, d ← r3

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

Compare Instruction

* Compare 3 and 5, and print the value of the flags

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

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

* flags.E = 0, flags.GT = 0

Compare Instruction

* Compare 5 and 3, and print the value of the flags

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

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

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

Compare Instruction

* Compare 5 and 5, and print the value of the flags

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

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

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

Example with Division

Write assembly code in SimpleRisc to compute: 31 / 29 - 50, and save the result in r4.

Answer:

SimpleRisc

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

Logical Instructions

and r1, r2, r3	$r1 \leftarrow r2 \,\&\, r3$
or r1, r2, r3	$r1 \leftarrow r2 \mid r3$
not r1, r2	$r1 \leftarrow \sim r2$
& bitwise AND, | bitwise OR, ~ logical complement	

* The second argument can either be a register or an immediate

Compute (*a* | *b*). *Assume that a is stored in r*0, *and b is stored in r*1. *Store the result in r*2.

SimpleRisc

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

Shift Instructions

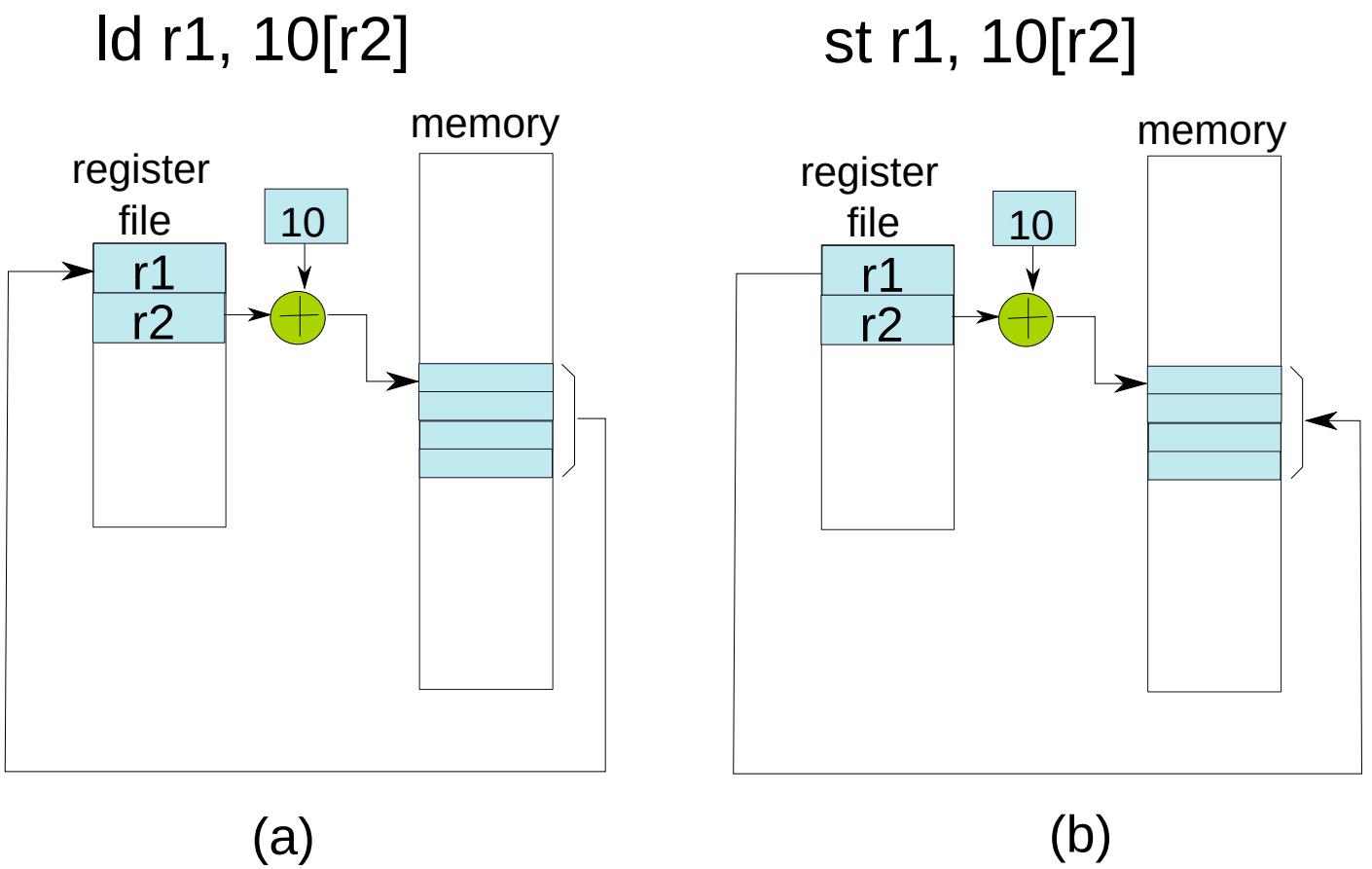
- Logical shift left (lsl) (<< operator)
  - 0010 << 2 is equal to 1000
  - (<< n) is the same as multiplying by 2^n
- logical shift right (lsr) (>>> operator)
  - 1000 >>> 2 = 0010
  - same as dividing the unsigned representation by 2^n
- Arithmetic shift right (asr) (>> operator)
  - 0010 >> 1 = 0001
  - 1000 >> 2 = 1110
  - same as dividing a signed number by 2^n

Load-store instructions

ld r1, 10[r2]	$r1 \leftarrow [r2 + 10]$
st r1, 10[r2]	$[r2+10] \leftarrow r1$

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

Load-Store


ld r1, 10[r2]
memory
register
file
10
r1
r2
(a)
st r1, 10[r2]
memory
register
file
10
r1
r2
(b)

Example – Load/Store

* 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]
```

Branch Instructions

* Unconditional branch instruction

b .label	branch to .label

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

Conditional Branch Instructions

beq .label	branch to .label if *flags.E* = 1
bgt .label	branch to .label if *flags.GT* = 1

* The flags are set only by cmp instructions
* beq (branch if equal)
  * If flags.E = 1, jump to .label
* bgt (branch if greater than)
  * If flags.GT = 1, jump to .label

Examples

* 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
```

Ex: Compute factorial of variable num

C

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

SimpleRISC

```
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*/
```

Modifiers

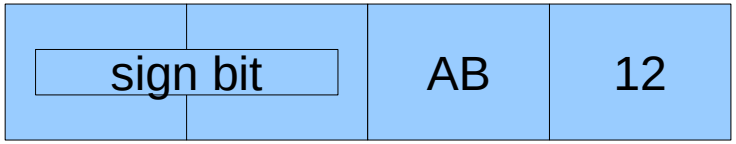
- We can add the following modifiers to an instruction that has an immediate operand
- Modifier :
  - default : mov → treat the 16 bit immediate as a signed number (automatic sign extension)
  - (u) : movu → treat the 16 bit immediate as an unsigned number
  - (h) : movh → left shift the 16 bit immediate by 16 positions

Mechanism

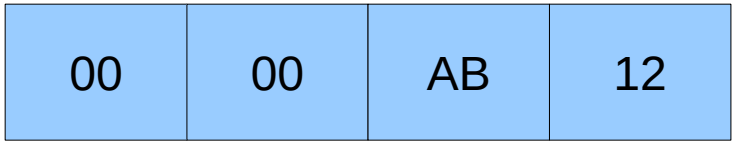
- The processor internally converts a 16 bit immediate to a 32 bit number
- It uses this 32 bit number for all the computations
- Valid only for arithmetic/logical insts
- We can control the generation of this 32 bit number
  - sign extension (default)
  - treat the 16 bit number as unsigned (u suffix)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

More about Modifiers

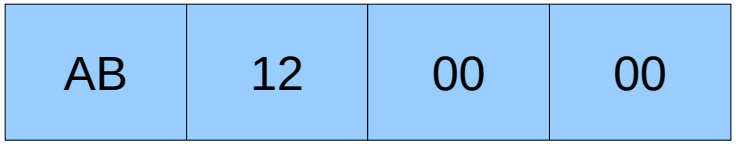
* default : mov r1, 0xAB 12



* unsigned : movu r1, 0xAB 12



* high: movh r1, 0xAB 12



Examples

* 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
```

Example

* Set r0 ← 0x 12 AB A9 2D

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