Assembly Language

Outline

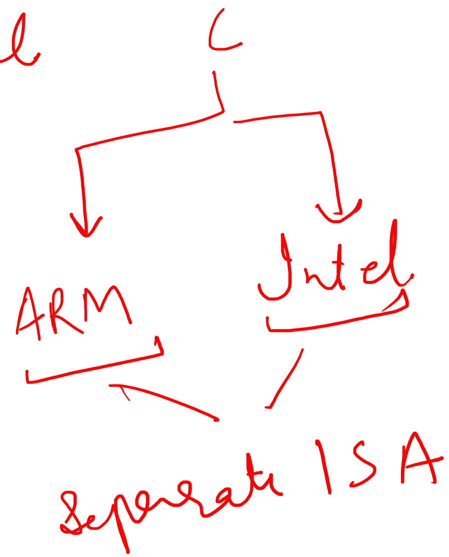
- ✓Overview of Assembly Language
- ✓Assembly Language Syntax
- ✓SimpleRisc ISA

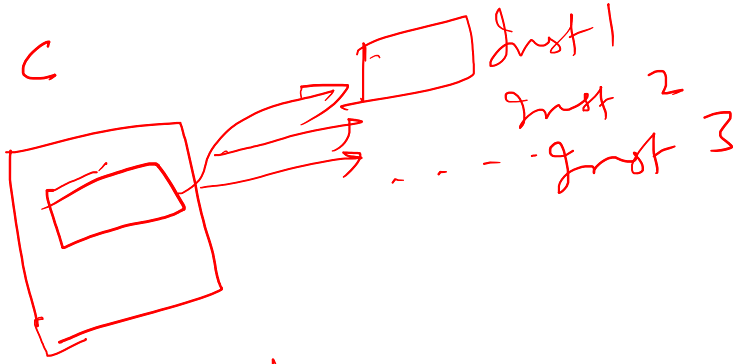
What is Assembly Language?

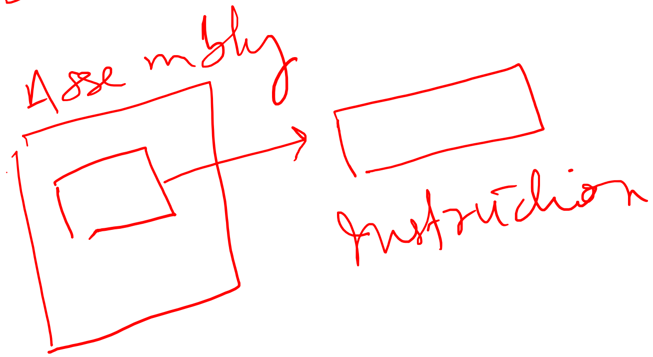
- The term "assembly language" refers to a family of low-level programming languages that are specific to an ISA. They have a generic structure that consists of a sequence of assembly statements.
- Typically, each assembly statement has two parts: (1) an instruction code that is a mnemonic for a basic machine instruction, and (2) and a list of operands.

Assembly Language Explained

Some high level language







Each statement in C has multiple Inst

Each statement in Assembly is Single Instr?

Why learn Assembly Language ?

- Time Efficient
- Less Memory Consumption
- Easy Hardware Manipulation

Intel → Intel Assembly

ARM → ARM Assembly

Assembly lang^n each Statement is single inst^r so more close to H/w

Smart lights

↳ Small controllers

→ Not designed using HLL

→ Efficiency of Assembly

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)

Compiler file

filename.c $\xrightarrow{\text{gcc -S}}$ filename.S

assembly file

gcc

a.out ←

Run Binary file ./a.out.

↳ Seperate for other OS's.

Hardware Designers Perspective

- Learning the assembly language is the same as learning the intricacies of the instruction set

- Tells HW designers : what to build ?

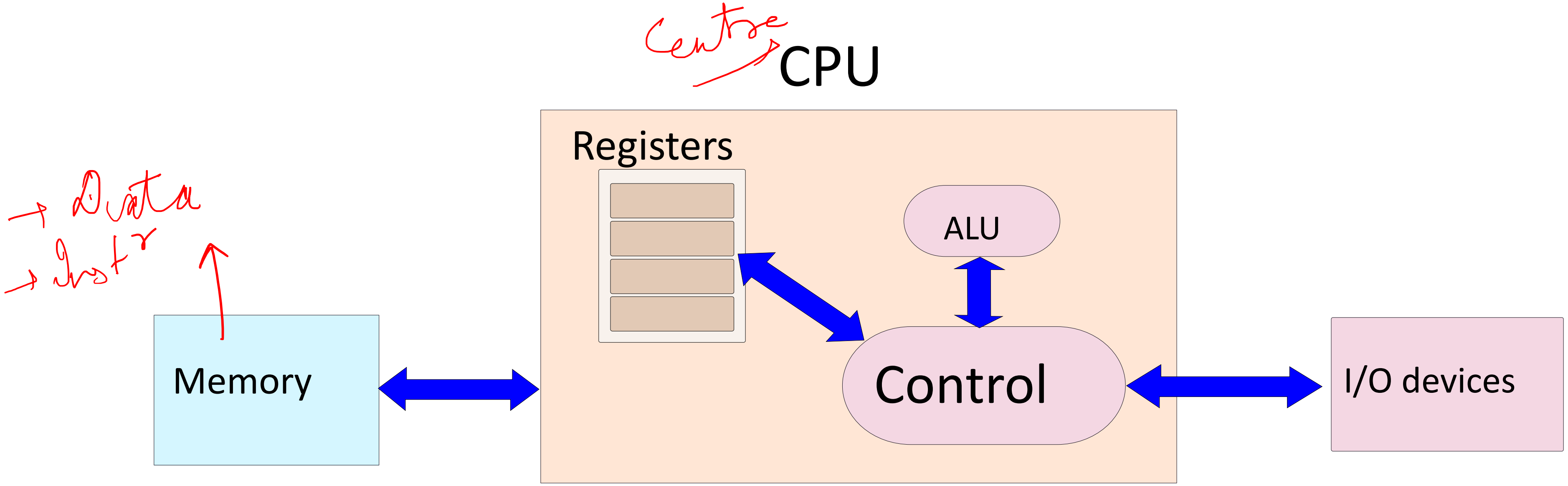
→ close to H/w

→ Interface b/w Software & H/w

→ Program different H/w's

→ Know's what to Build

Machine Model – Von Neumann Machine with Registers

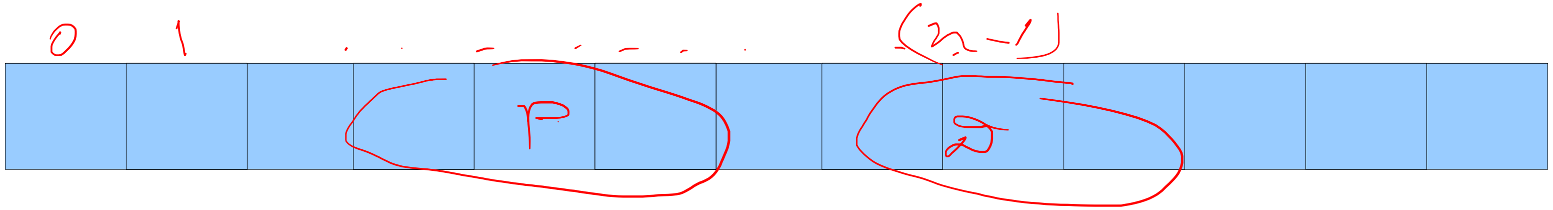


RISC
Processors
Follow this Architecture

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

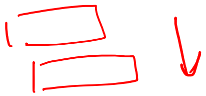
View of Memory



- 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

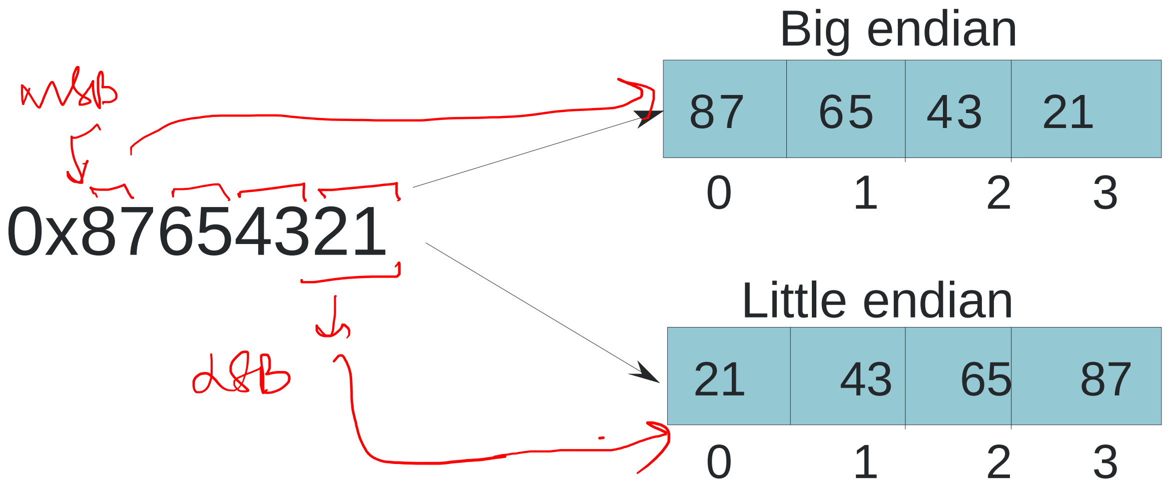
Storage of Data in Memory

- 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 ? → $i, i+1, i+2, i+3$
  - Save the 4 bytes in consecutive locations



  - Little endian representation → The LSB is stored in the lowest location
  - Big endian representation → The MSB is stored in the lowest location

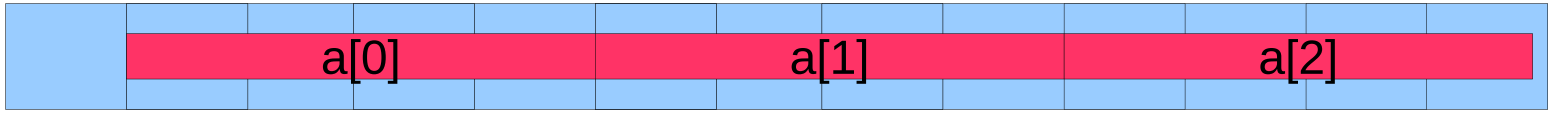
Little Endian vs Big Endian



- Note the order of the storage of bytes

Storage of Arrays in Memory

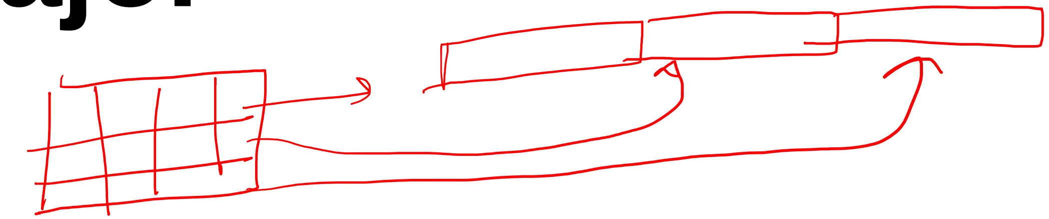
- Single dimensional arrays. Consider an array of integers : a[100]

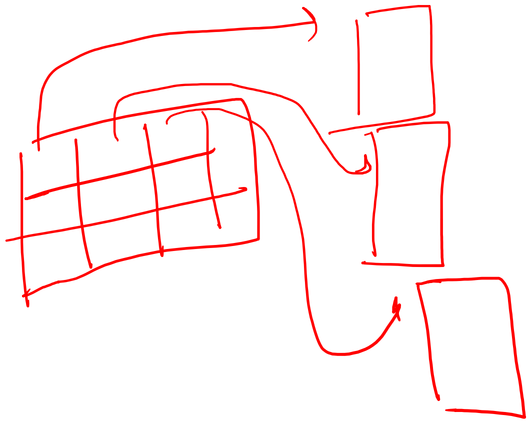


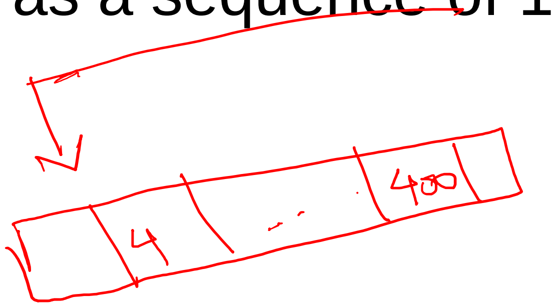
- Each integer is stored in either a little endian or big endian format
- 2 dimensional arrays :
  - int a[100][100]
  - float b[100][100]
  - Two methods : row major and column major

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





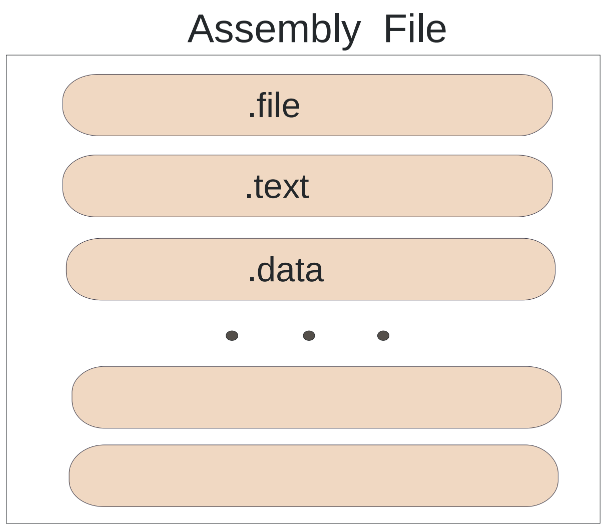


a[100] ← array of 100 integers

Each Integer is 4 Byte long

→ Stored in little|big En Format

Assembly File Structure : GNU Assembler

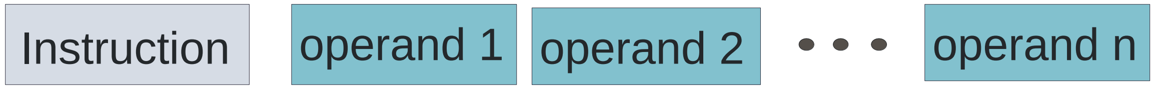


- Divided into different sections
- Each section contains some data, or assembly instructions

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 operand 1 operand 2 • • • operand n

- 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*

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

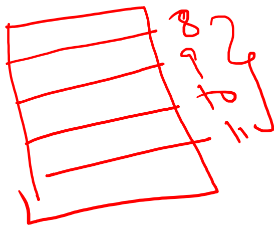
Nature of Operands

- 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]   Memory Load
  - access the memory location that matches the contents of r2, and store the data in register r1

r2 = 8



contents of locations

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
- 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])

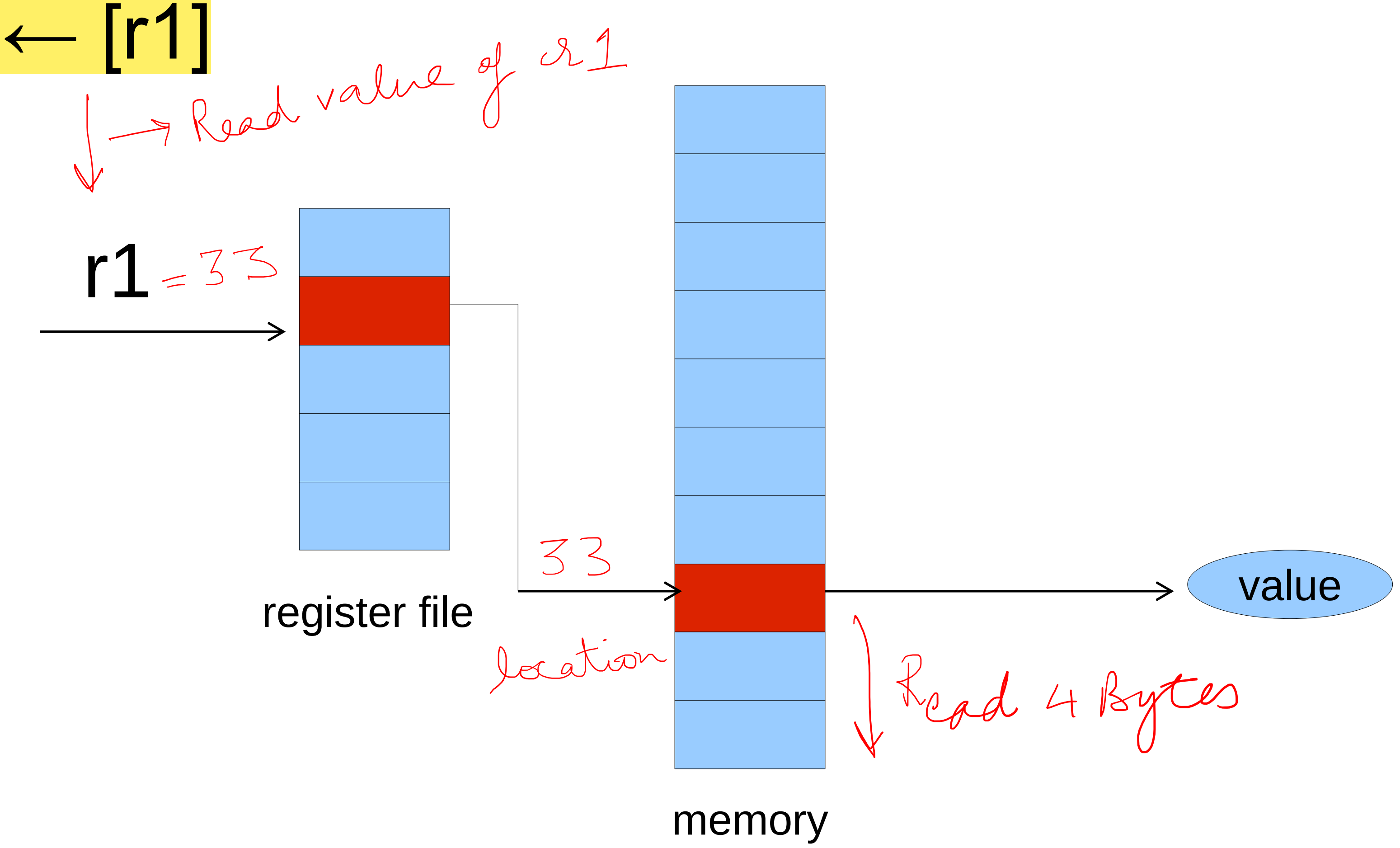
add r3, r2, 4

add r3, r2, [r1]

add r3, r2, 20[r1]

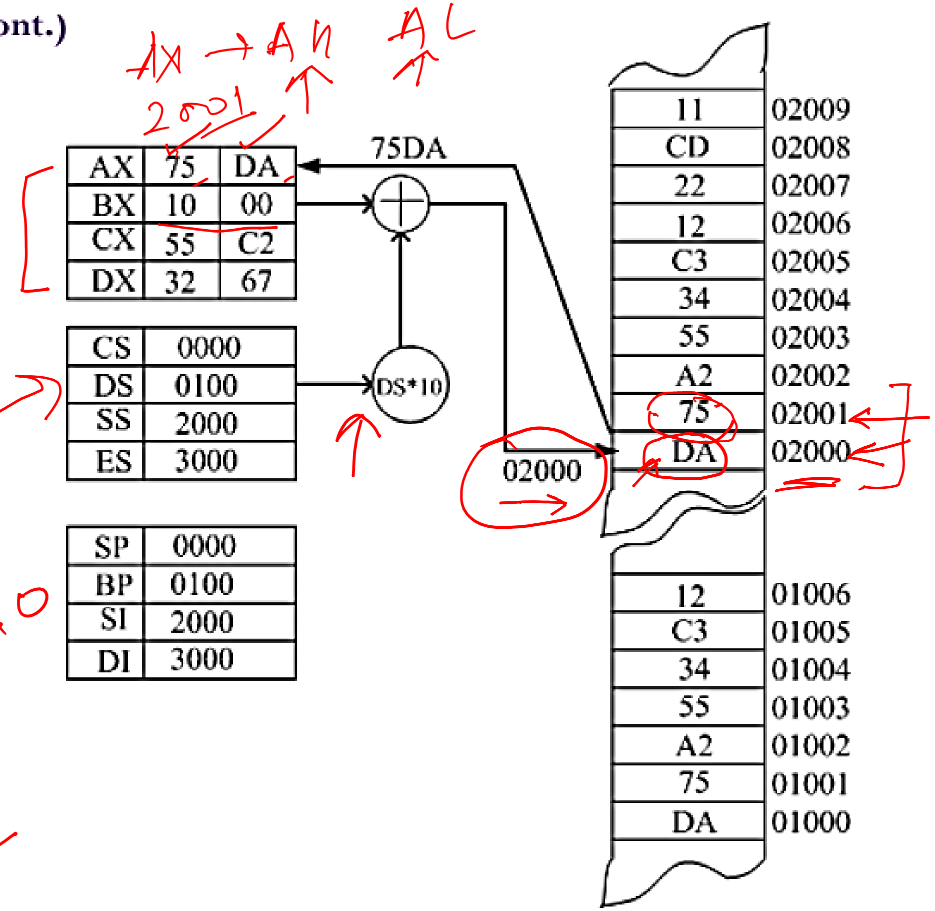
Register Indirect Mode

- V ← [r1]



Register Indirect addressing (Cont.)

Operation of ~~MOV AX, [BX]~~ instruction



AX	75	DA
BX	10	00
CX	55	C2
DX	32	67

CS	0000
DS	0100
SS	2000
ES	3000

SP	0000
BP	0100
SI	2000
DI	3000

11	02009
CD	02008
22	02007
12	02006
C3	02005
34	02004
55	02003
A2	02002
75	02001
DA	02000
12	01006
C3	01005
34	01004
55	01003
A2	01002
75	01001
DA	01000

AX → AH AL

2001

2 Bytes

by DS affected

8086 → BX = 1000

DS = 0100

DS Hard wired 0

0100[0]

BX 1000

02000

CS	3	4	8	A	0
IP +		4	2	1	4
Instruction address	3	8	A	B	4

SS	5	0	0	0	0
SP +		F	F	E	0
Stack address	5	F	F	E	0

DS	1	2	3	4	0
DI +		0	0	2	2
Data address	1	2	3	6	2

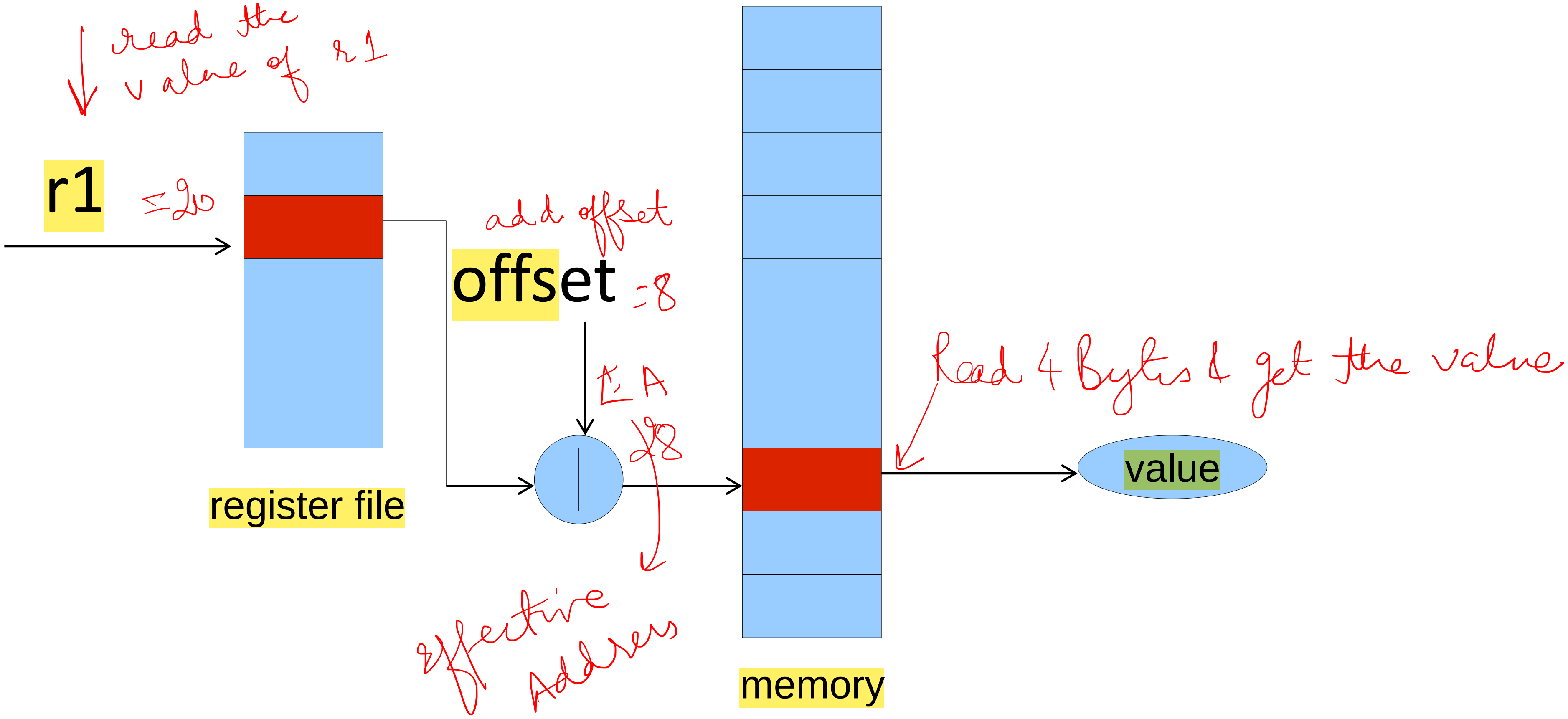
Segment address 0000

+ Offset

Memory address

Base-offset Addressing Mode

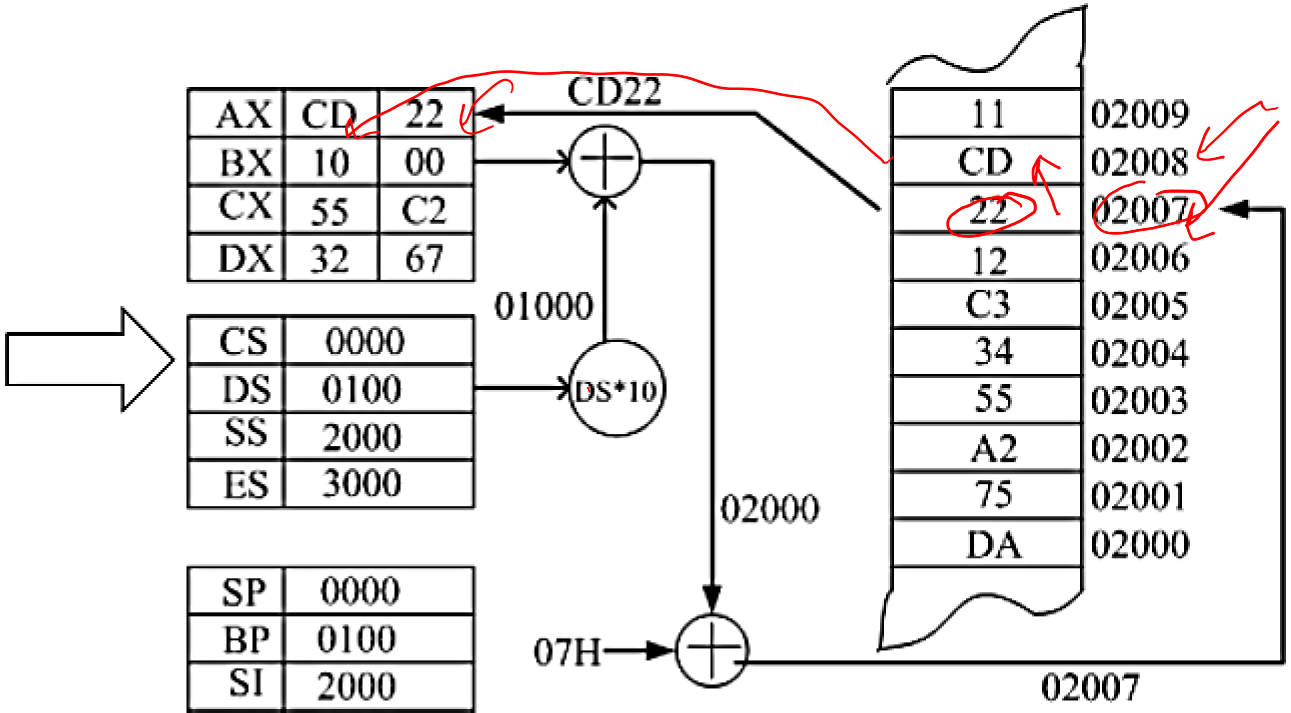
- V ← [r1+offset]



8086 → Default in DS
we have instr

Default Segment Register

Operation of
MOV AX, [BX+07H]
instruction



AX	CD	22
BX	10	00
CX	55	C2
DX	32	67

CS	0000
DS	0100
SS	2000
ES	3000

SP	0000
BP	0100
SI	2000
DI	0004

11	02009
CD	02008
22	02007
12	02006
C3	02005
34	02004
55	02003
A2	02002
75	02001
DA	02000

Bx = 1000
07H
offset =
DS = 0100

Effective Address ⇒ 2007 ⇒

01000
Bx 1000
07
offset 02007

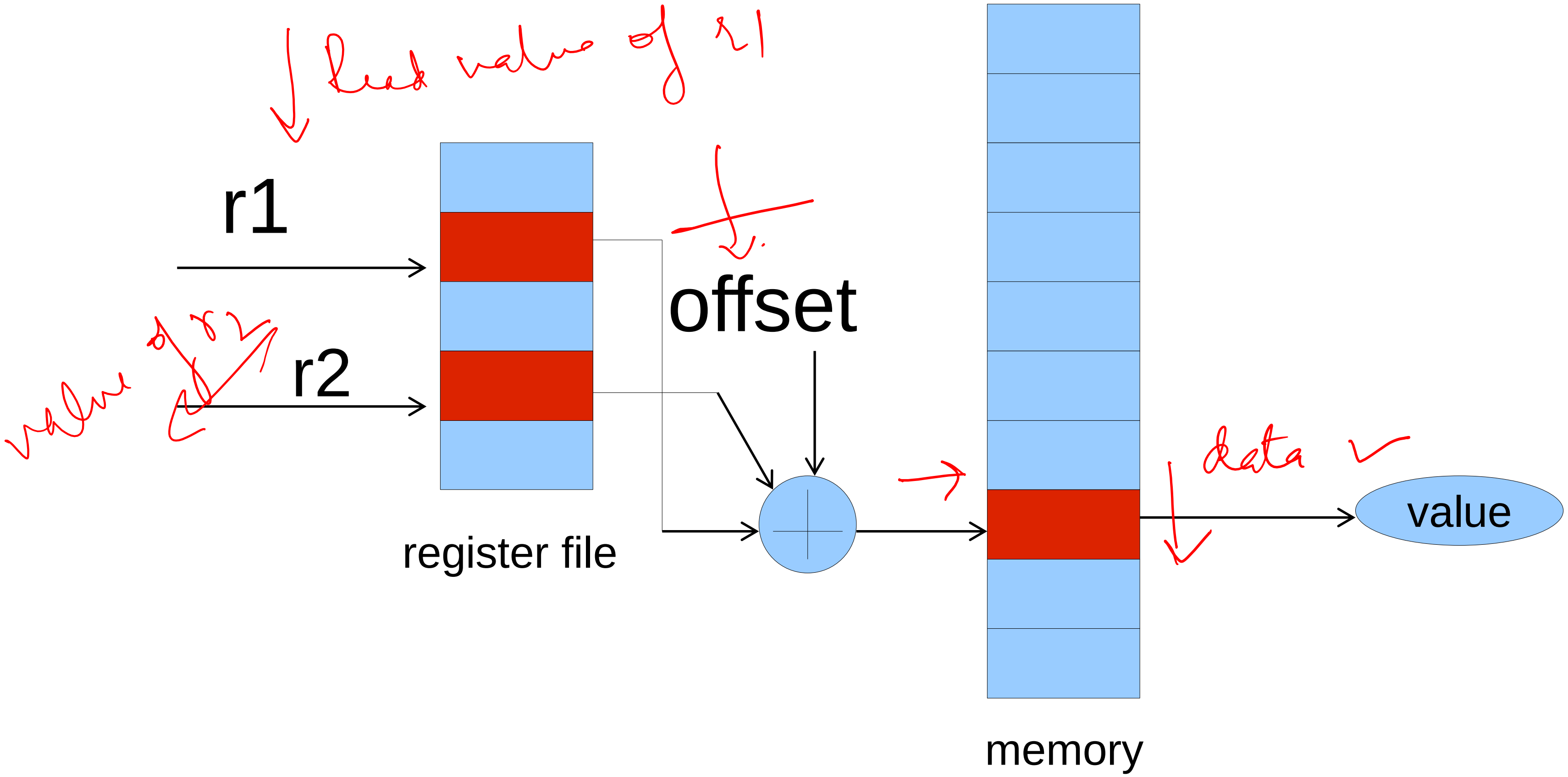
Addressing Modes - II

- 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 register
Index register
offset

Base-Index-Offset Addressing Mode

- V ← [r1+r2 +offset]



ISA

→ 8086 → 20 bit address Bus

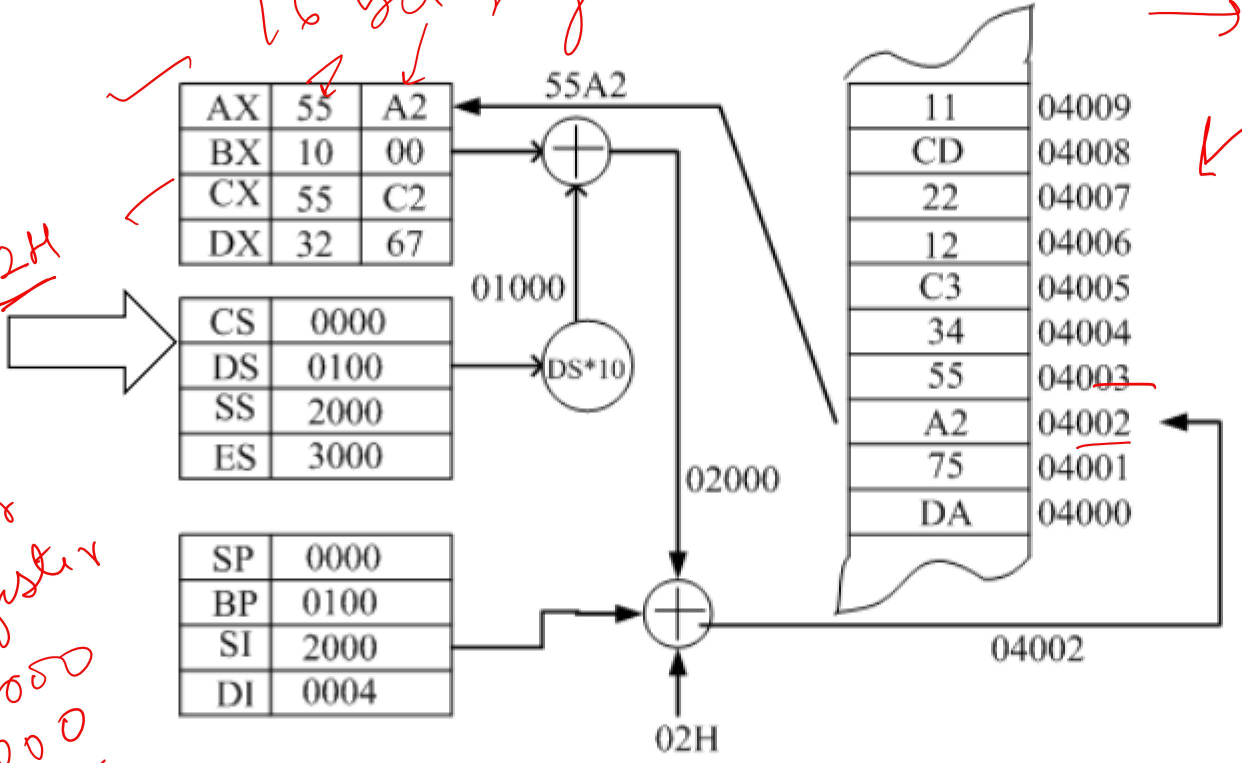
16 bit registers

Operation of
MOV AX, [BX+SI+02H]
instruction

02H

Base register
Index Register

01000
1000
2000
04000
+2
04002



AX	55	A2
BX	10	00
CX	55	C2
DX	32	67

CS	0000
DS	0100
SS	2000
ES	3000

SP	0000
BP	0100
SI	2000
DI	0004

11	04009
CD	04008
22	04007
12	04006
C3	04005
34	04004
55	04003
A2	04002
75	04001
DA	04000

Base register
[0]
H/W Zero

SimpleRisc

- Simple RISC ISA
- Contains only 21 instructions
- *SimpleRisc* assumes to have 16 registers numbered *r*0 . . . *R*15
- The first 14 registers are general purpose registers, and can be used for any purpose within the program.
- Register *r*14 is known as the stack pointer.
- Register *r*15 is known as the return address register, and it will also be referred as *ra*

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
  - Von Neumann model
  - One large array of bytes
- Special flags register → contains the result of the last comparison
  - flags.E = 1 (equality), flags.GT = 1 (greater than) → only instr

cmp instr.

✓ cmp r1, r2

r1 - r2

r1 = r2 ; flags.E = 1

r1 > r2 ; flags.G = 1

otherwise = 0

mov instruction

mov r1,r2	$r1 \leftarrow r2$
mov r1,3	$r1 \leftarrow 3$

way to write a Assembly language instr

opcode destination / src

mov r1, 3

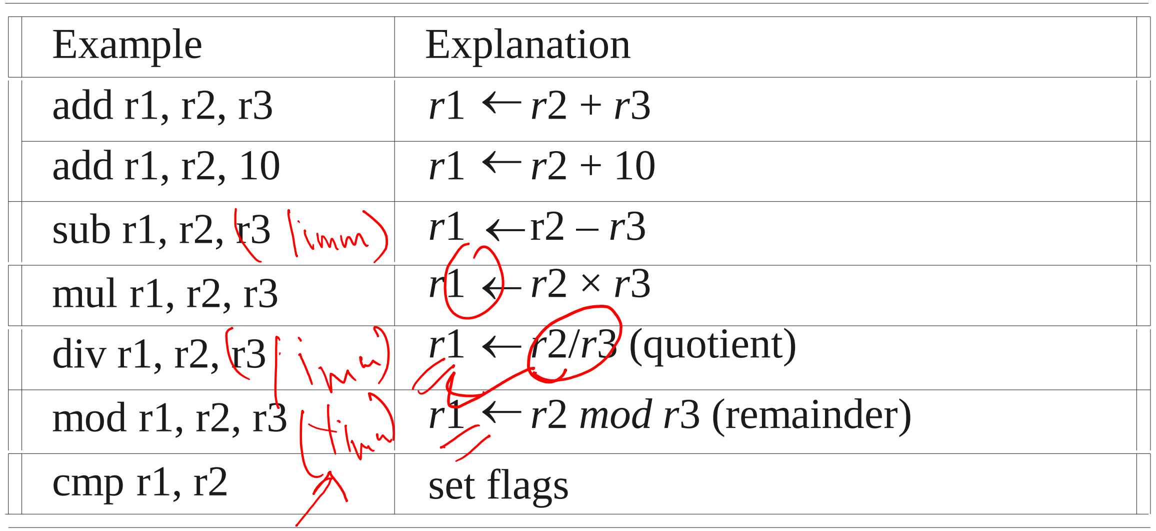
r1:= 3

- 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

Arithmetic/Logical Instructions

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

add r1, r2, (r3 | immediate)



Example	Explanation
add r1, r2, r3	$r1 \leftarrow r2 + r3$
add r1, r2, 10	$r1 \leftarrow r2 + 10$
sub r1, r2, (r3 | imm)	$r1 \leftarrow r2 - r3$
mul r1, r2, r3	$r1 \leftarrow r2 \times r3$
div r1, r2, (r3 | imm)	$r1 \leftarrow r2/r3$ (quotient)
mod r1, r2, r3 (| imm)	$r1 \leftarrow r2 \; mod \; r3$ (remainder)
cmp r1, r2	set flags

Cmp → only Instr in RISC that sets the Flags

r1 - r2 = 0 → flags.E = 1

r1 - r2 > 0 flags.GT = 1

Flags would remain set till the next compare Instr.

cmp → The Flags may be used by other Instr

cmp

Examples of Arithmetic Instructions

- Convert the following code to assembly

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

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

r0 = 3

r1 = 5

r2 ← c

r3 ← d

- 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 – Contd.

- Convert the following code to assembly

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

mul r2, r0, r1

mod r3, r2, 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 Contd.

- 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
```

r1 > r1

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

Compare Instruction Contd.

- 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.

31, 29
r1 r2

div r3, r1, r2
sub r4, r3, 50

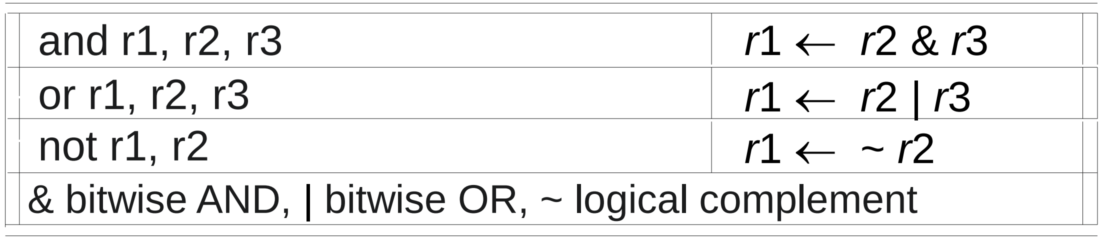
Immediate value

Answer:

SimpleRisc

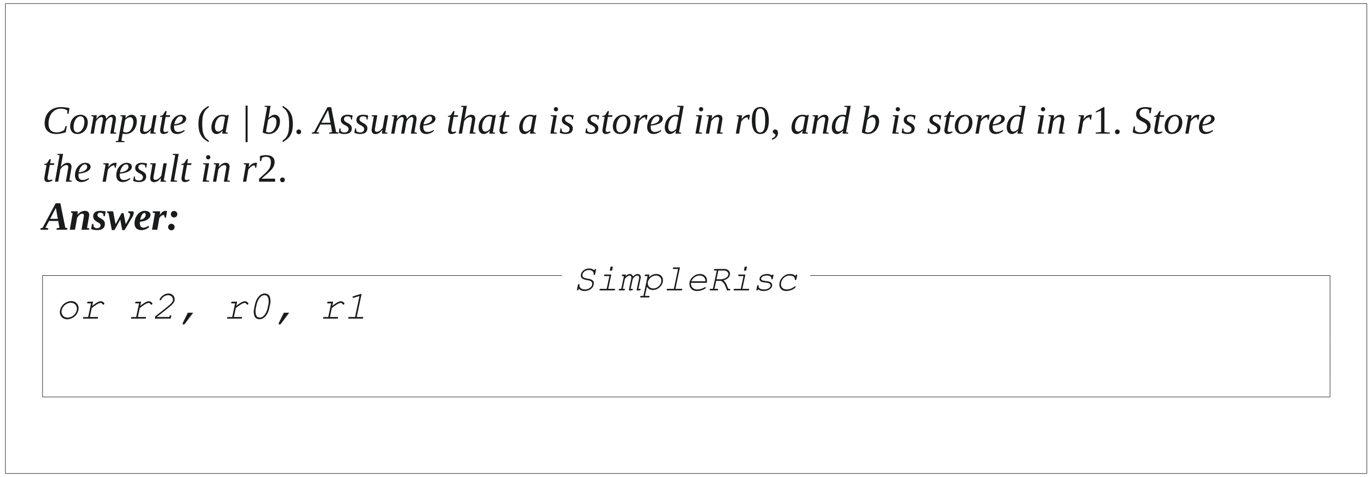
```
mov r1, 31
mov r2, 29
div r3, r1, r2
sub r4, r3, 50
```

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



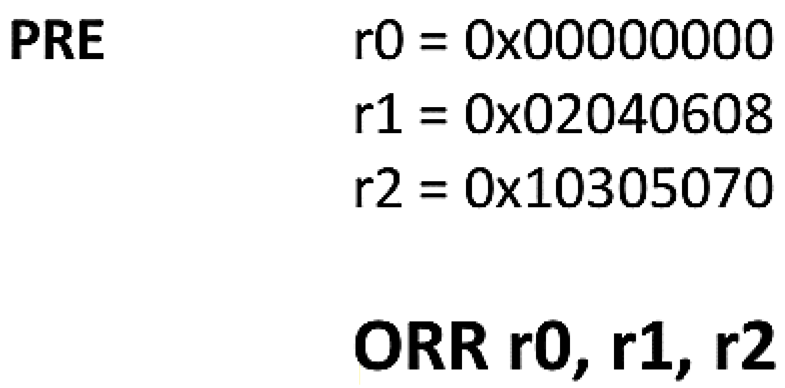
Compute $(a \mid b)$. *Assume that a is stored in* $r0$*, and b is stored in* $r1$*. Store the result in* $r2$.

Answer:

SimpleRisc

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

Logical OR Example



```
PRE          r0 = 0x00000000
             r1 = 0x02040608
             r2 = 0x10305070

             ORR r0, r1, r2
```

POST: ?

Shift Instructions

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

0 0 1 0 — $\boxed{2}$

0 1 0 0 ← 1 Shift

1 0 0 0 ← 2nd shift

⟹ $\boxed{8} = 2 \times 2^2 = 8$

0 1 0 0 >> 2 = $4/2^2 = 1$

0 0 0 1

No = -6

2's complement form

1 0 1 0 >> 1

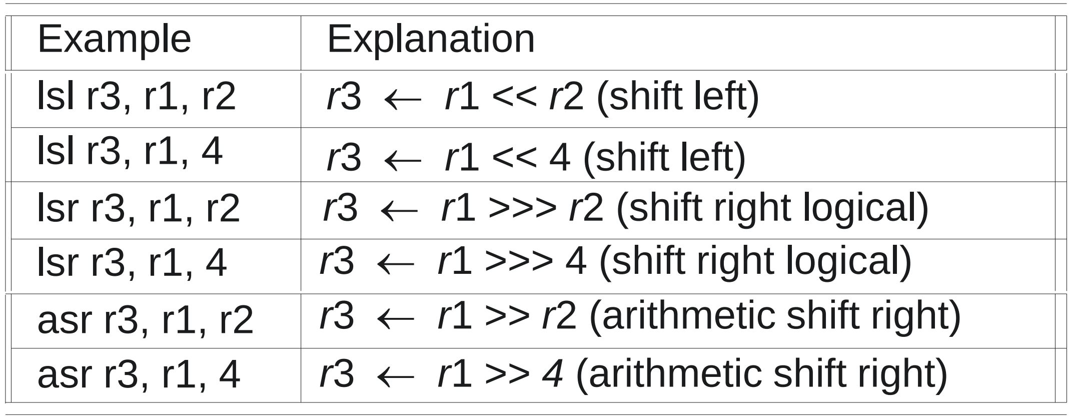
$\boxed{1}$ 1 0 1 = $\boxed{-3}$ = $-6/2$ = -3

sign bit

Divide by = 2^1

Shift Instructions – Contd.

- logical shift right (lsr) (>>> operator)
  - 1000 >>> 2 = 0010
  - same as dividing the unsigned representation by 2^n



Example	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	$r3 \leftarrow r1 \ggg r2$ (shift right logical)
lsr r3, r1, 4	$r3 \leftarrow r1 \ggg 4$ (shift right logical)
asr r3, r1, r2	$r3 \leftarrow r1 \gg r2$ (arithmetic shift right)
asr r3, r1, 4	$r3 \leftarrow r1 \gg 4$ (arithmetic shift right)

Example with Shift Instructions

- Compute 101 * 6 with shift operators

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

why not?

mul r0, r0, 6

$r1 = 101 \times 2$

$r2 = 101 \times 2^2$

$r3 = 101 \times 6$

In H/W

→ multiplication & Division Take lot of time (More No: of operations)

Example – Contd.

- Compute 102 * 7.5 with shift operators

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

Example – Contd.

Write simple assembly code in *SimpleRisc* to compute the following:

i) $a + b + c$

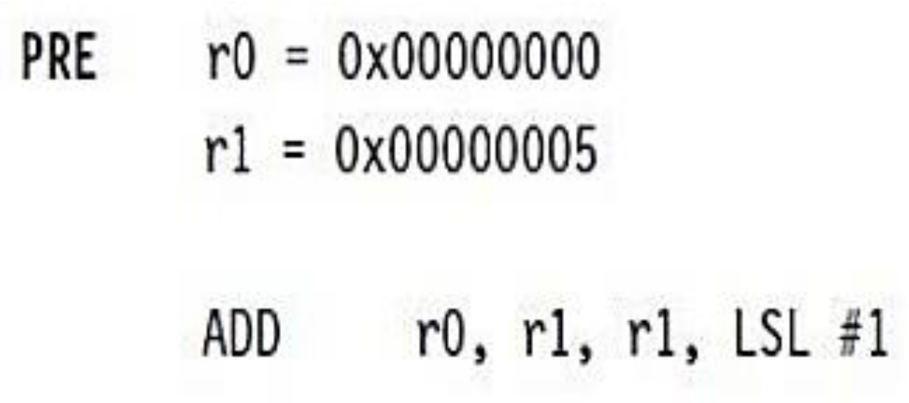
ii) $a + b - c/d$

iii) $(a + b) * 3 - c/d$

iv) $a/b - (c * d)/3$

v) (a<<2)-(b>>3)

Shift and Arithmetic Instructions



```
PRE     r0 = 0x00000000
        r1 = 0x00000005

        ADD     r0, r1, r1, LSL #1
```

POST: ?

r0 => 0x0000000f

r1 = 0x00000005

Example – Contd.

Write an assembly program to set the 5*th* bit of register *r*0 to the value of the 3*rd* bit of *r*1. Keep the rest of the contents of *r*0 the same. The convention is that the LSB is the first bit, and the MSB is the 32*nd* bit.

```
and R1,R1,4

lsl R1,R1,2

and R0,R0,0xffef

or R0,R0,R1
```

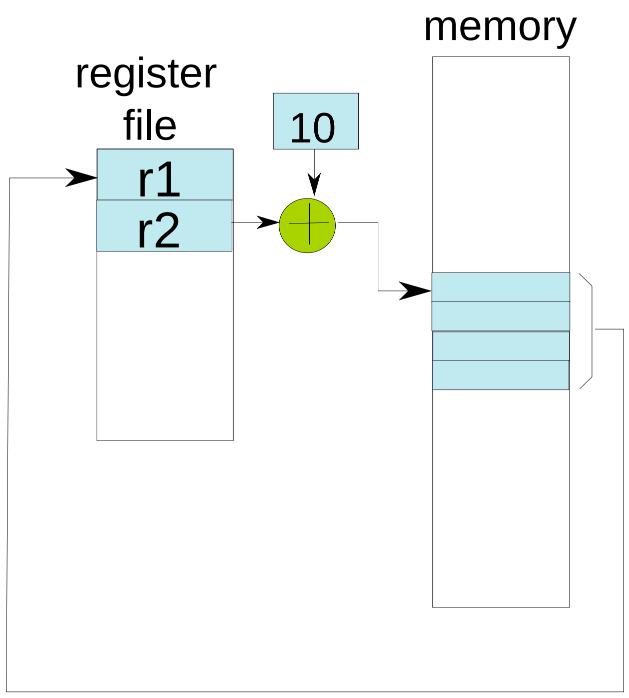
Load-store instructions

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

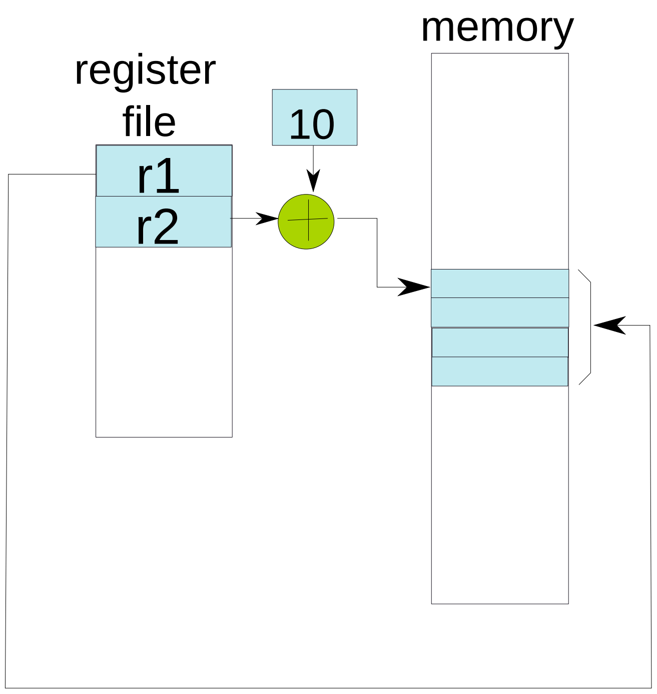
Load-Store Contd.

ld r1, 10[r2]



(a)

st r1, 10[r2]

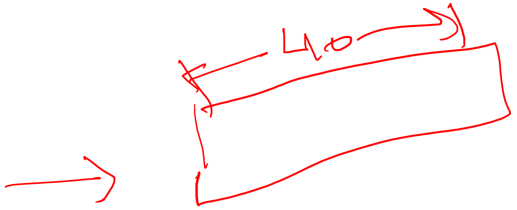


(b)

Example – Load/Store

- Translate :

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



arr[0] → r0 → r0+3

arr[1] = r0+4 — r0+7

arr[9] = r0+36

```
/* 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]
```

r1 = 5

arr[3] = 12[r0]

r2 = 8

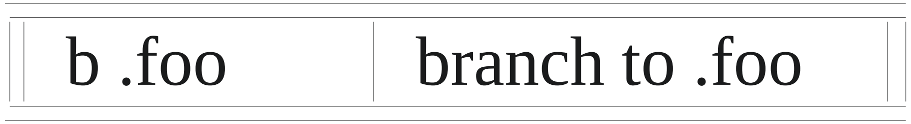
arr[4] = 16[r0]

arr[5] → 20[r0]

r3 ← arr[5] + arr[4]

Branch Instructions

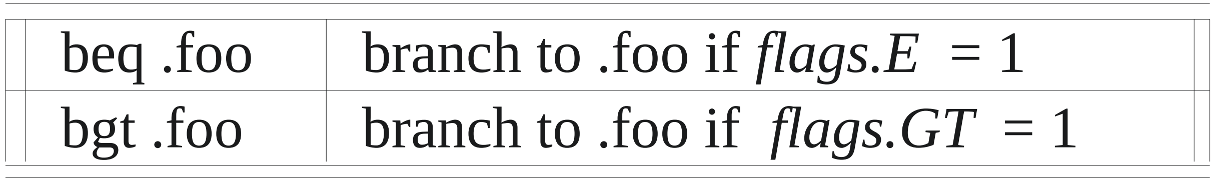
- Unconditional branch instruction



b .foo	branch to .foo

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

Conditional Branch Instructions



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

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

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
```

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

sign bit		AB	12

- unsigned : movu r1, 0xAB 12

00	00	AB	12

- high: movh r1, 0xAB 12

AB	12	00	00

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 Contd.

- Set r0 ← 0x 12 AB A9 2D

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

Example Contd.

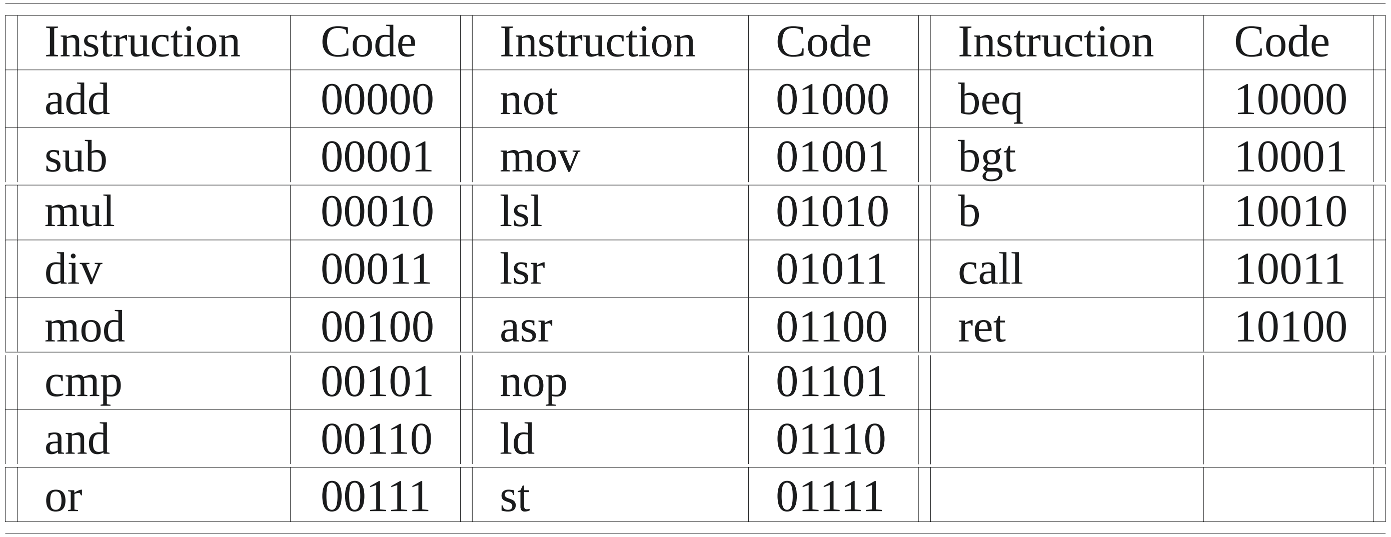
Write a program to load the value 0*xFFEDFC*00 into *r*0.

```
movh R0,0xFFED

addu R0,R0,0xFC00
```

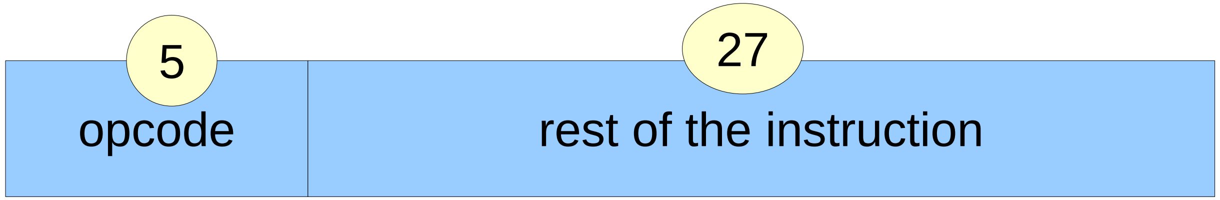
Encoding Instructions

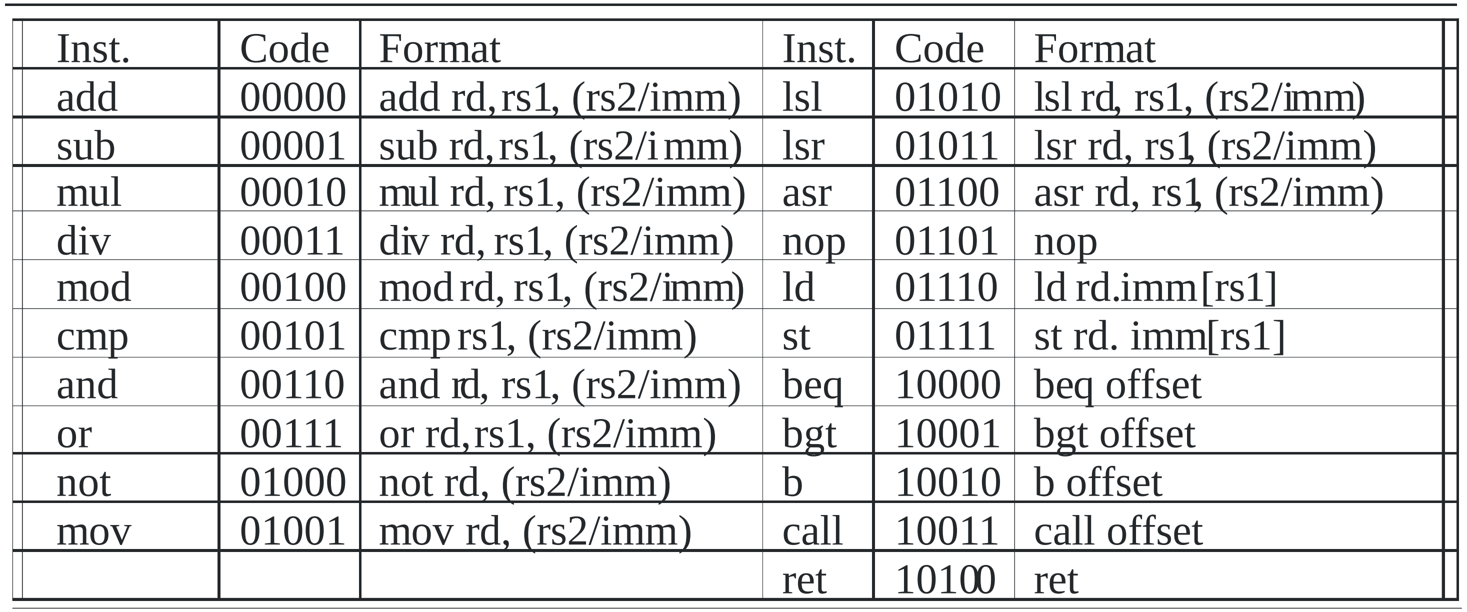
- Encode the SimpleRisc ISA using 32 bits.
- We have 21 instructions. Let us allot each instruction an unique code (opcode)



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

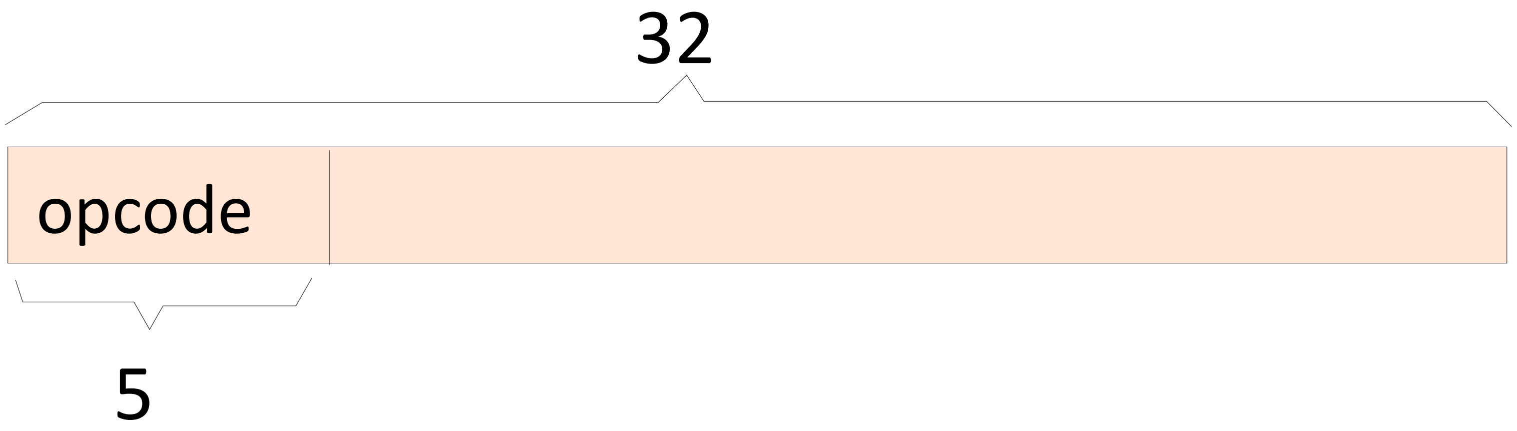




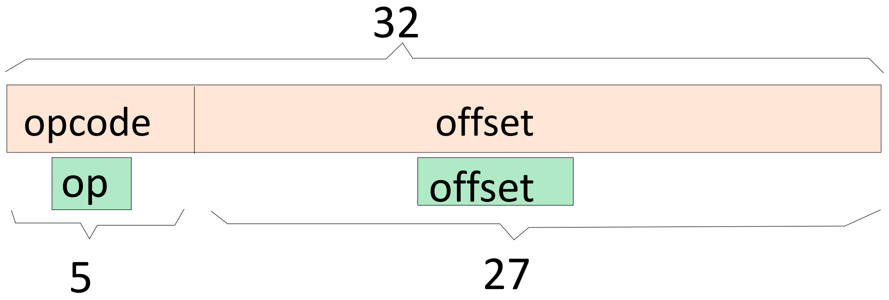
Inst.	Code	Format	Inst.	Code	Format
add	00000	add rd,rs1, (rs2/imm)	lsl	01010	lsl rd, rs1, (rs2/imm)
sub	00001	sub rd,rs1, (rs2/imm)	lsr	01011	lsr rd, rs1, (rs2/imm)
mul	00010	mul rd, rs1, (rs2/imm)	asr	01100	asr rd, rs1, (rs2/imm)
div	00011	div rd, rs1, (rs2/imm)	nop	01101	nop
mod	00100	mod rd, rs1, (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

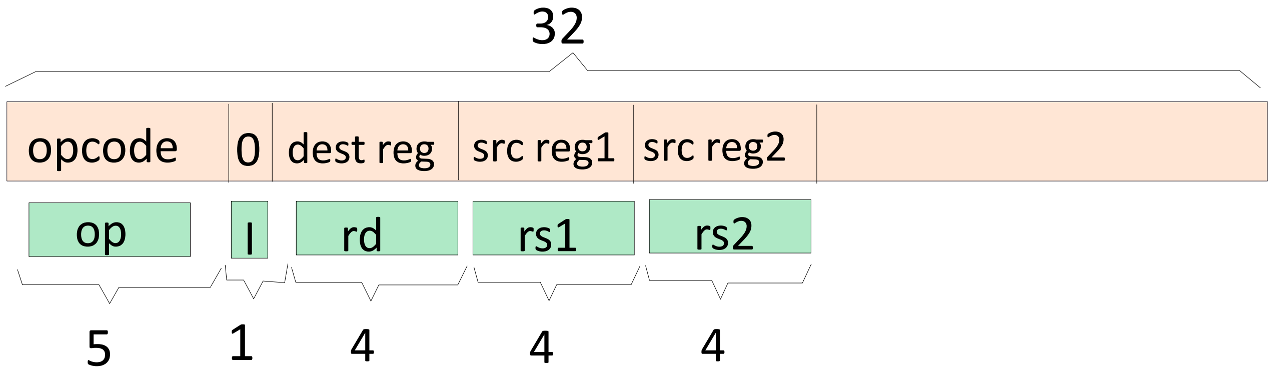


- Instructions – call, b, beq, bgt
- Use the branch format
- Fields :
  - 5 bit opcode
  - 27 bit offset (PC relative addressing)
  - Since the offset points to a 4 byte word address
    - The actual address computed is : PC + offset * 4

3-Address Instructions

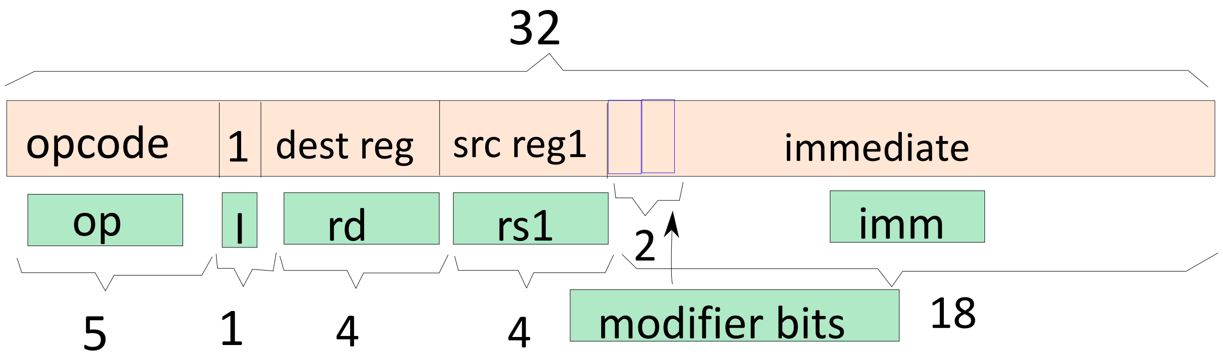
- Instructions – add, sub, mul, div, mod, and, or, lsl, lsr, asr
- Generic 3 address instruction
  - <opcode> rd, rs1, <rs2/imm>
- Let us use the I bit to specify if the second operand is an immediate or a register.
  - I = 0 → second operand is a register
  - I = 1 → second operand is an immediate
-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

Register Format



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

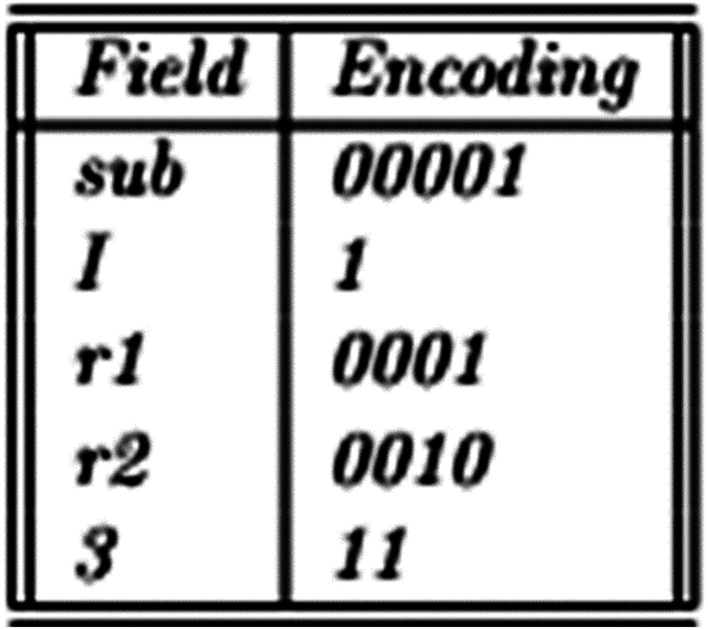
Immediate Format



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

Example

Encode the instruction: sub r1, r2, 3



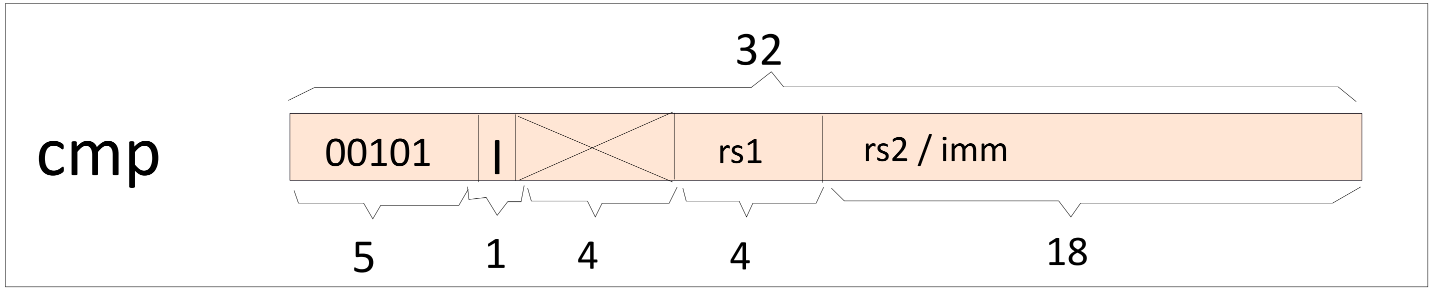
Field	Encoding
sub	00001
I	1
r1	0001
r2	0010
3	11

0x0C480003

2 Address Instructions

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

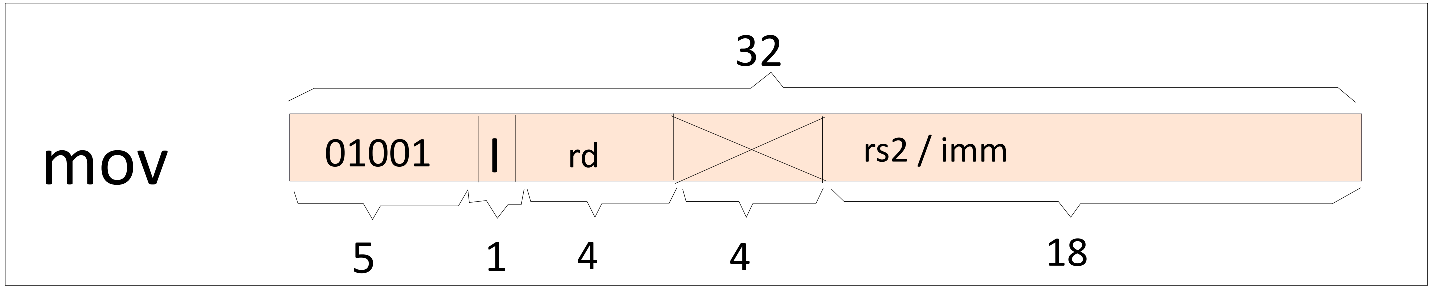
cmp, not, and mov



cmp

32

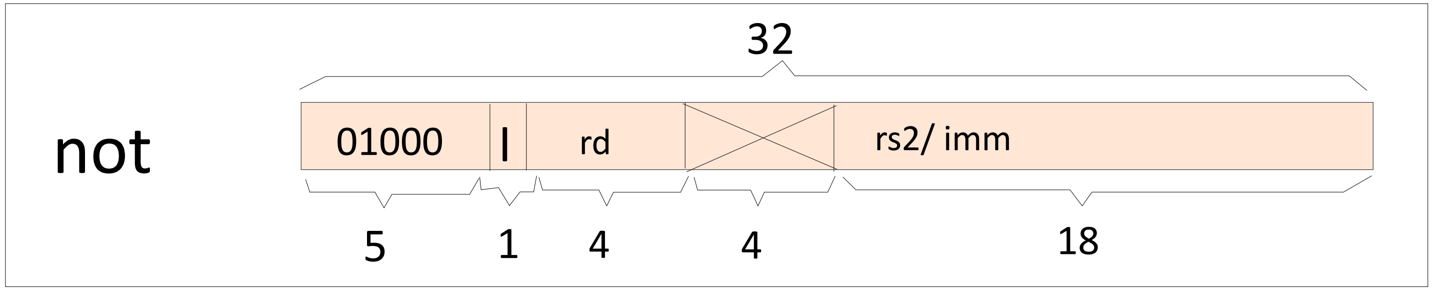
00101	I		rs1	rs2 / imm
5	1	4	4	18



mov

32

01001	I	rd		rs2 / imm
5	1	4	4	18



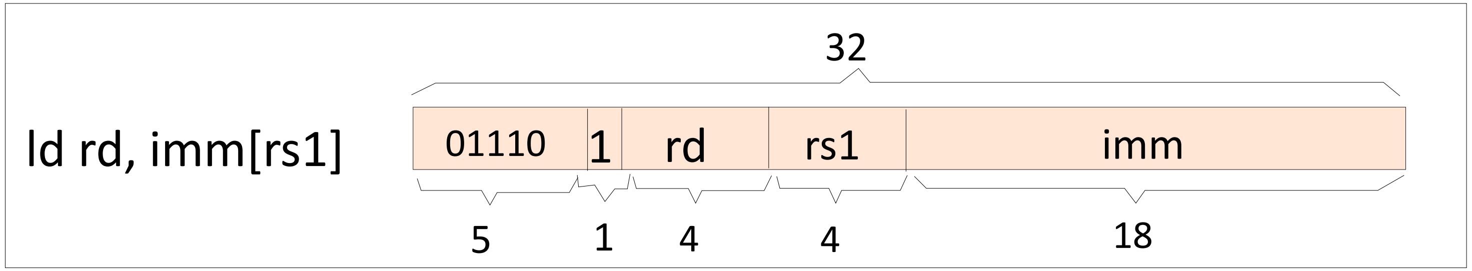
not

32

01000	I	rd		rs2/ imm
5	1	4	4	18

Load and Store Instructions

- ld rd, imm[rs1]
- rs1 → base register
- Use the immediate format.

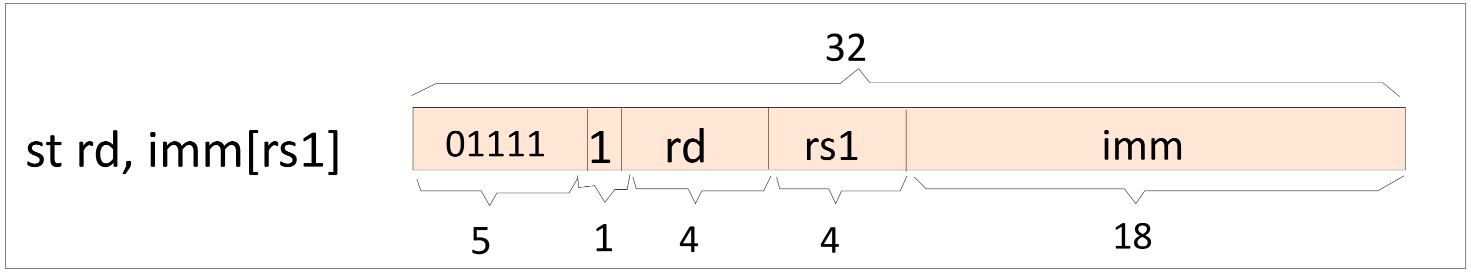


32

ld rd, imm[rs1]	01110	1	rd	rs1	imm
	5	1	4	4	18

Store Instruction

- 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]



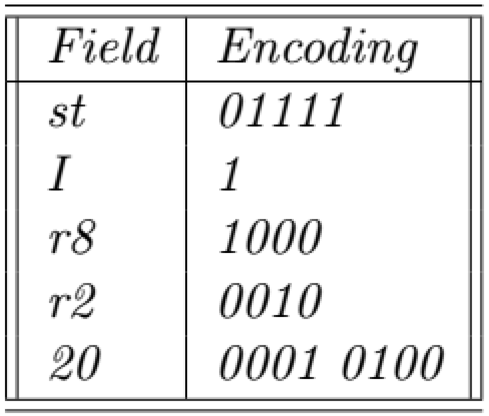
st rd, imm[rs1]

32

01111	1	rd	rs1	imm
5	1	4	4	18

Example

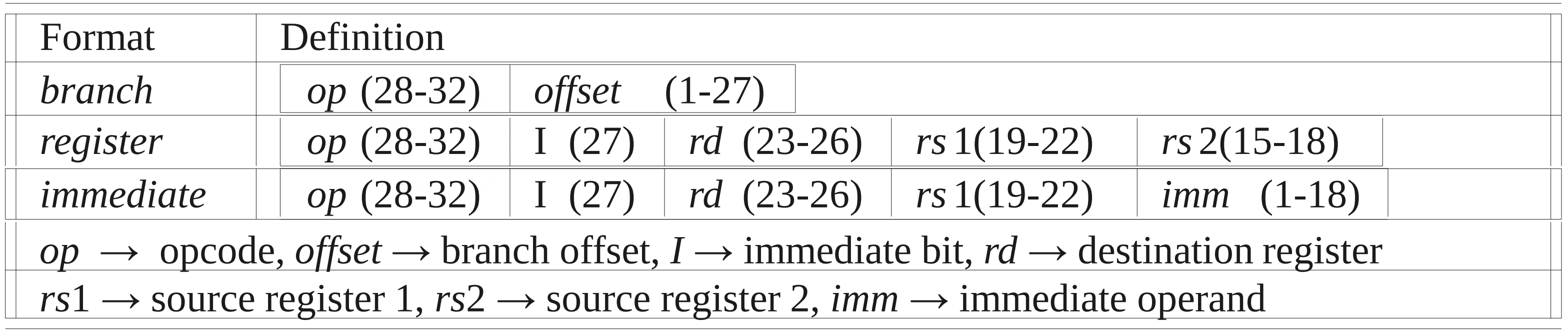
Encode the instruction: st r8, 20[r2].



Field	*Encoding*
st	*01111*
I	*1*
r8	*1000*
r2	*0010*
20	*0001 0100*

0x7E080014

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