Overview of Assembly Language

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 seq of assembly statements

→ Each assembly statement has 2 parts:

1) An instruction code that is a mnemonic for a basic machine instruction.

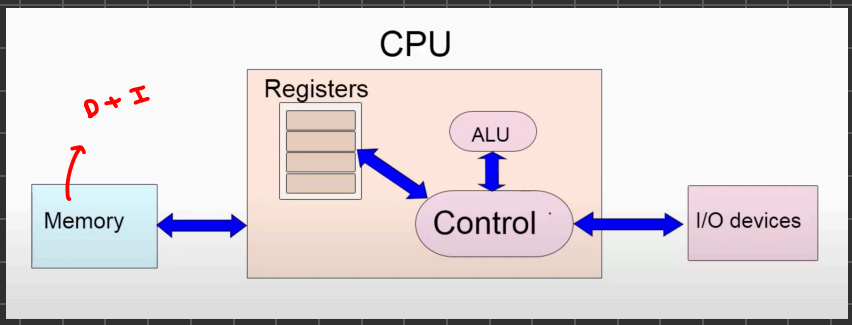
2) A list of operands

→ Low level ⇒ close to the hardware

Assemblers : Programs that convert programs written in low level languages to machine code (0s & 1s)

→ Learning the assembly language is the same as learning the intricasies of the ISA. It tells the HW designers what to build.

Machine Model: Von Neumann Machine with Registers



View of Registers:

→ Registers are named storage locations

→ Machine specific registers are called MSR

→ Registers with special fⁿ:

* stack pointer
* program counter (PC)
* return address

View of Memory

→ One large array of bytes

→ each location has an address

→ program is stored in a part of the memory. Data is stored in another part

→ program counter contains the address of the current instruction.

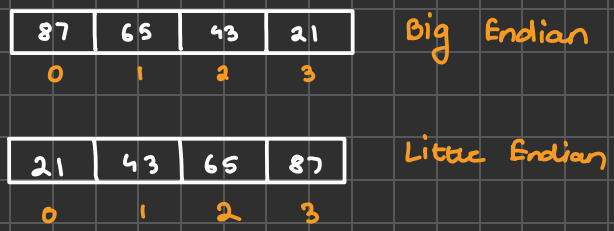
Storage of Data in Memory

→ Typical Data types :

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

→ How are multibyte variables stored in memory ?

Eg: 0xc8765432l



Storage of Arrays in Memory

→ Two methods used for storing 2D arrays :

1) Row Major (C, python)
2) Column Major (Fortran, MATLAB)

→ Multidimensional Arrays are stored as a sequence of 1D arrays.

Assembly Language Syntax

Assembly File Structure : GNU Assembler

→ divided into different sections

→ each section contains some data, or assembly instructions.

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

Structure of an Assembly Statement

Instruction	Operand 1	Operand 2	.....	Operand n

→ Instruction : Textual identifier of machine instruction

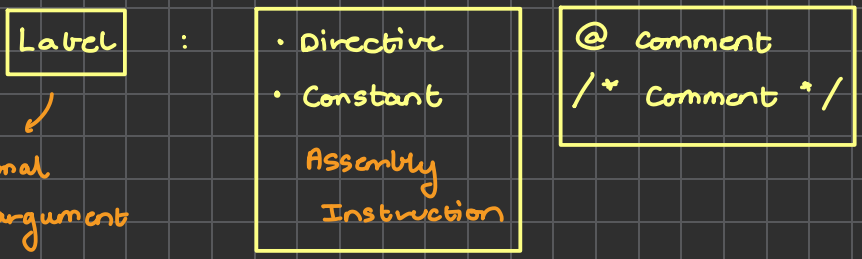
→ Operand :

* constant (known as immediate)
* register
* memory location

Eg: `sub r3, r1, r2` (r3 = r1 - r2)

`mul r3, r1, r2` (r3 = r1 * r2)

Generic Statement Structure



→ Label : identifier of a statement

→ Directive : tells the assembler to do something like declare a function

→ Constant : declares a constant

Types of Instructions

1) Data Processing instructions

2) Data Transferring instructions

3) Branch instructions
   → branch to a given label

4) Special instructions
   → interact with peripheral devices & other programs
   → set machine specific parameters

Nature of Operands

→ Classification of instructions : if an instruction takes n operands, it is said to be in the n-address format

→ Addressing Mode : The method of specifying & accessing an operand in an assembly statement

Register Transfer Notation

Eg : 1) r1 ← r2

2) r1 ← r2 + 4

3) r1 ← [r2]

access the memory location that matches the contents of r2 & store that data in r1.

Addressing Modes

1) Immediate Addressing Mode :

V ← imm Eg: 4, 8, -3, 0x487

2) Register direct AM

V ← r1

3) Register indirect AM

V ← [r1]

4) Base - offset AM :

V ← [r1 + offset]

Eg : 20[r1] (V ← [r1 + 20])

5) Base - index - offset AM

V ← [r1 + r2 + offset]

Eg : 100[r1, r2] (V ← [r1 + r2 + 100])

6) Memory direct AM

v ← [addr]

7) PC Relative AM

v ← [PC + offset]

Eg: 100 [PC] (v ← [PC + 100])

SimpleRISC ISA

→ Contains only 21 instructions
→ we will design an assembly language for SimpleRisc
→ Design a simple binary encoding & then implement it.

Registers

→ SimpleRisc has 16 registers
  * Numbered : r0, ..., r15
  * r14 is referred to as the stack pointer (SP)
  * r15 is 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 comparision

Eg: flags.E = 1 (equality)

flags.GT = 1 (greater than)

1) mov instruction

```
mov r1, r2      // r1 ← r2
mov r1, 3       // r1 ← 3
```

→ Transfer 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

* simple Risc has 16 bit intermediates (2's complement)
* range -2^{15} to $2^{15} - 1$

2) Arithmetic Instructions

1)
```
add r1, r2, r3
add r1, r2, 3
```

2)
```
sub r1, r2, r3
sub r1, r2, 3
```

3)
```
mul r1, r2, r3
mul r1, r2, 3
```

4) `div r1, r2, r3` — $r1 \leftarrow r2 / r3$

`div r1, r2, 3` — $r1 \leftarrow r2 / 3$

5) `mod r1, r2, r3` — $r1 \leftarrow r2 \,\%\, r3$

`mod r1, r2, 3` — $r1 \leftarrow r2 \,\%\, 3$

6) `cmp r1, r2` — set flags

`cmp r1, 3`

Q) Convert the following code to assembly language

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

S1) Assign vars to registers

a ← r0 , b ← r1 , c ← r2 , d ← r3

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

9)

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

a ← r0, b ← r1, c ← r2, d ← r3

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

3) Logical Instructions &, |, ~ (bitwise operands)

and	r1, r2, r3	r1 ← r2 & r3
or	r1, r2, r3	r1 ← r2 | r3
not	r1, r2	r1 ← ~r2

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

4) Shift Instructions

→ Logical Shift Left (lsl) (<< operator) → works for 2's complement -ve no.

* <<n ⇒ multiplying with 2^n

→ Arithmetic shift right (asr) (>> operator)

* same as dividing a signed no. by 2^n

→ Logical shift right (lsr) (>>> operator)

Eg :

```
lsl  r3, r2, r1
asr  r3, r2, r1
lsr  r3, r2, r1
```

can be an immediate as well

Q) Compute 101 * 6 with shift operators

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

$r_0 \leftarrow 101$

$r1 \leftarrow r0 << 1$ $(r1 \leftarrow r0 \times 2)$

$r2 \leftarrow r0 << 2$ $(r2 \leftarrow r0 \times 4)$

$r3 \leftarrow r1 + r2$

$(r3 \leftarrow r0 \times (2+4))$

$\Rightarrow$ 101×6

→ shift instructions are very fast in comparision to mul / div

5) Load - store Instructions

```
ld  r1, 10[r2]
st  r1, 10[r2]
```

r1 ← [r2 + 10]

[r2 + 10] ← r1

6) Branch Instructions

1) Unconditional Branch Instructions

```
b .foo          branch to .foo
```

b <name of label>

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

2) Conditional Branch Instructions

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

Eg: 1) 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
```

2) Compute the factorial of the variable num

In C:

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

In SimpleRisc:

```
mov r1, 1
mov r2, r0
.loop:
    mul r1, r1, r2
    sub r2, r2, 1
    cmp r2, 1
    bgt .loop
```

Modifiers

→ We can add the following modifiers to an instruction that has an immediate operand

→ Modifier :

1) default : mov → treat the 16 bit immediate as a signed no. (automatic sign extension)

2) (u) : movu → treat the 16 bit immediate as an unsigned no.

3) (h) : movh → left shift the 16 bit immediate by 16 positions.

Eg : set r0 ← 0xc 12 AB A9 20

```
movh  r0, 0xc 12 AB
addu  r0, 0xc A9 20
```

Functions & Stacks

Implementing Functions

→ To call a function, we need to set : pc ← A

→ We also need to store the location of the PC that we need to come after the function returns

→ This is known as the return address

→ We can thus call any function, execute its instructions, & then return to the saved return address

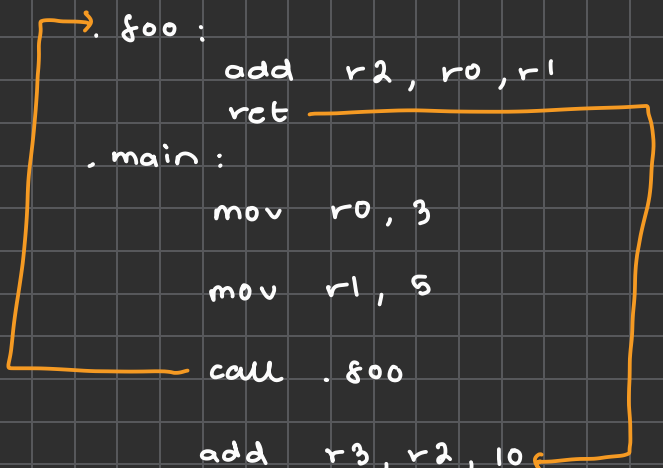
→ PC of call instruction : p

PC of return address : p + 4

↓

because every instruction takes 4 bytes

Eg :



Problems with this mechanism:

1) Space problem:
   * we have a limited no. of registers
   * we cannot pass more than 16 arguments
   * sol^ : use memory also

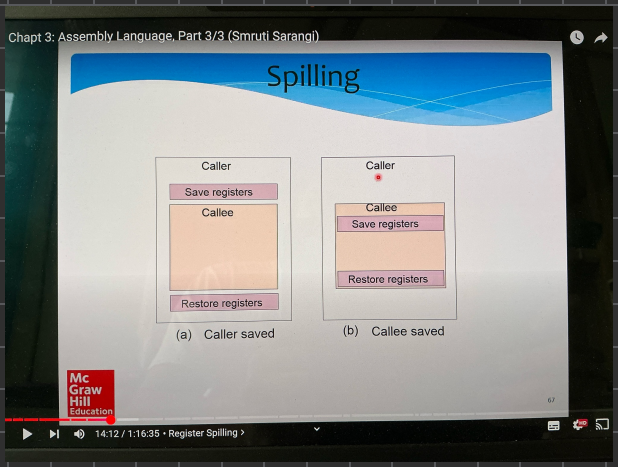
2) Overwrite Problem:
   * What if a f^ calls itself? (recursive call)
   * The callee can overwrite the registers of the caller
   * sol^n : spilling

Register Spilling:

* The caller can save the set of registers it needs
* call the function
* & then restore the set of registers after the function returns.
* known as the **caller saved scheme**

Callee saved Scheme:

* The callee saves the registers & then later restores them.



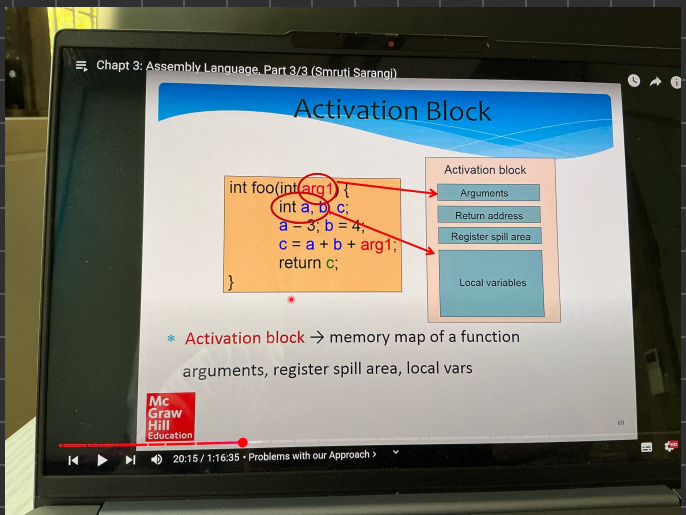
Problems with Our Approach

→ Using memory, & spilling, solves both the space problem & overwrite problem

→ However, there needs to be:

* A strict agreement b/w the caller & the callee regarding the set of memory locations that need to be used
* Secondly, after a f^n has finished execution, all the space that it uses needs to be reclaimed.

Activation Block: effectively the region of memory that is tasked to keep all the values that are needed for the function to execute.



How to use activation block ?

→ Assume caller saved spilling

→ Before calling the f^, spill the registers

→ Allocate the activation block of the callee

→ Write the arguments to the activation block of the callee, if they do not fit in the registers

→ Call the function.

→ In the called f^:

* Read the args & then transfer to registers if req

* Save the return address if the called f^ can call other f^

* Allocate space for local vars
* Execute the f^n

→ Once the f^n ends:

* Restore the value of the return address reg
* Write the return values to registers, or the activation block of the caller
* Destroy the activation block of callee
* Call the ret instruction
* & return to caller

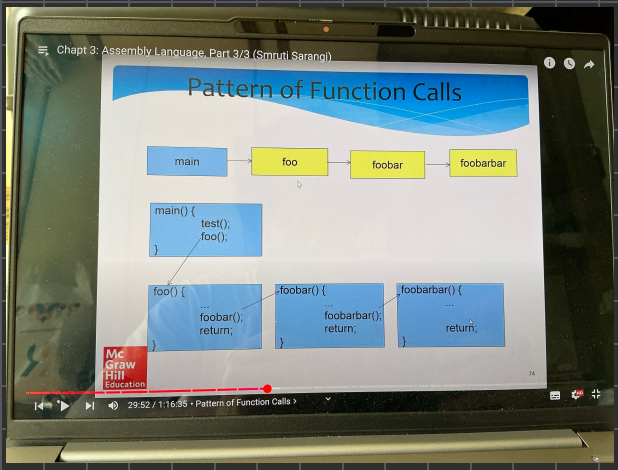
→ The caller:

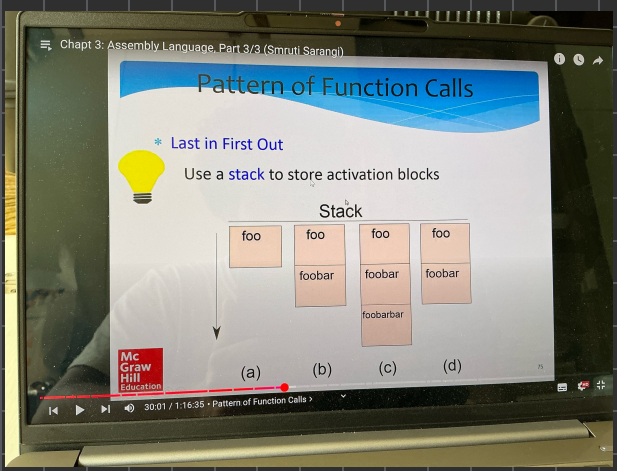
* Retrieves the return values from the registers from its activation block
* Restore the spilled registers
* Continue...

Organising Activation Blocks

→ All the info of an executing f^n is stored in its activation block

→ These blocks need to be dynamically created & destroyed millions of times





Note: We can store activation blocks in form of a STACK. (LIFO)

Working With A Stack

→ Allocate a part of the memory to save the stack

→ Traditionally stacks are downward growing

* The first activation block starts at highest address
* Subsequent activation blocks are allocated lower addresses

→ The stack pointer register (sp(14)) points to the beginning of an activation block

→ Allocating an activation block :

$sp \leftarrow sp - \langle const \rangle$

→ de-allocating an activation block:

$sp \leftarrow sp + \langle const \rangle$

What has the stack solved ?

1) Space problem
2) Overwrite problem
3) Management of Activation Blocks

The stack needs to be primarily managed in software

Call & Ret Instructions

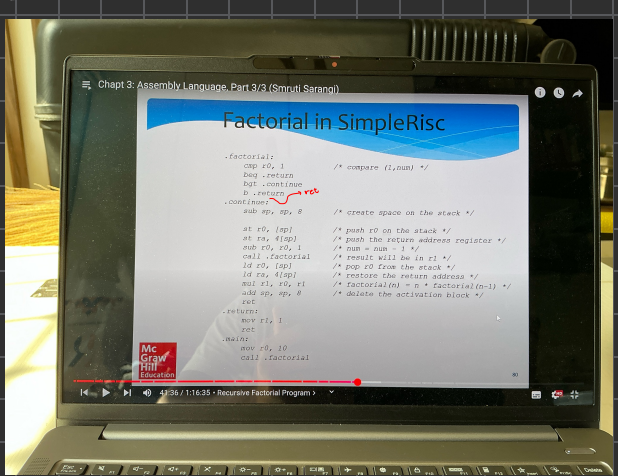
call .foo ra ← PC + 4
PC ← address(.foo)

ret PC ← ra

ra (or r15) is the return address register

Eg: Recursive Factorial Program

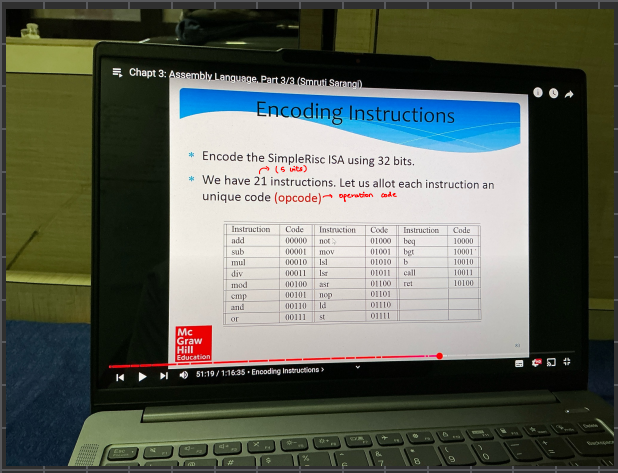
```
C:  int factorial(int num) {
        if (num <= 1) return 1;
        else return num * factorial(num-1);
    }
    void main() {
        int result = factorial(10);
    }
```

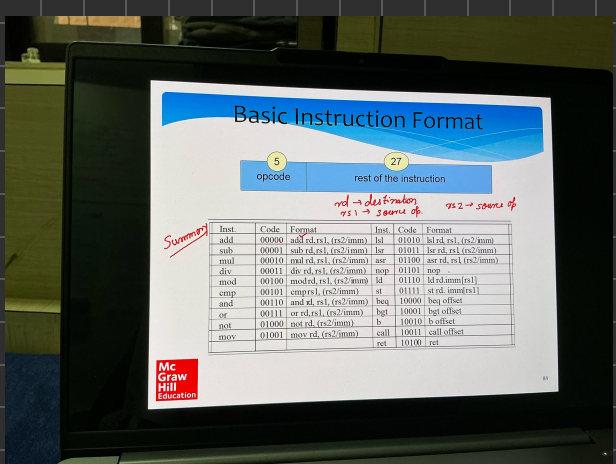


nop instruction

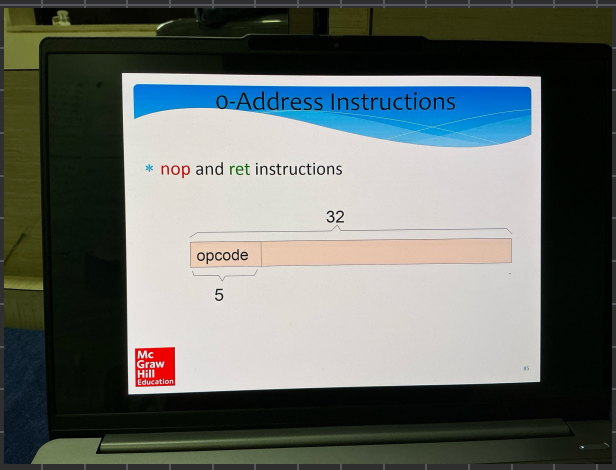
→ nop : does nothing

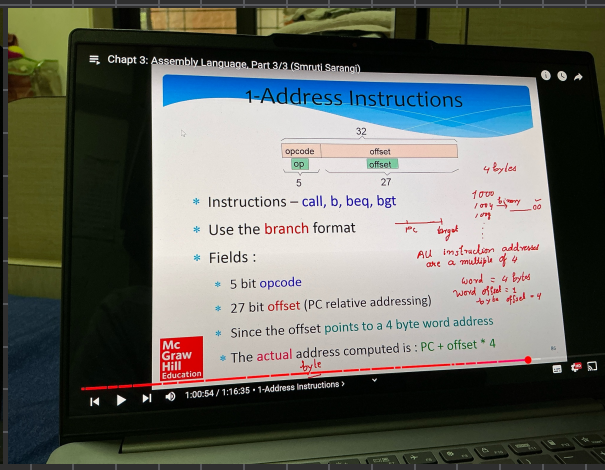
SimpleRisc Encoding

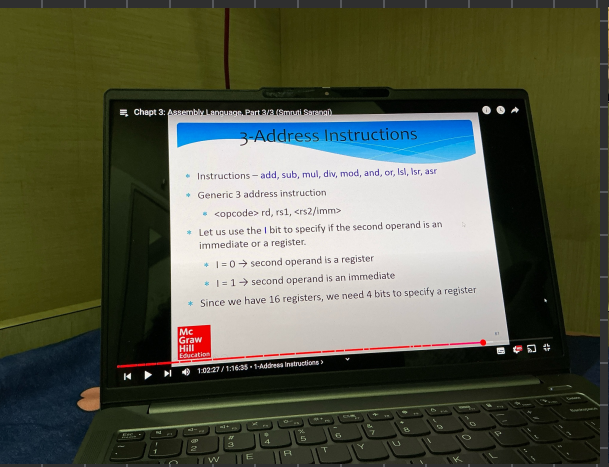


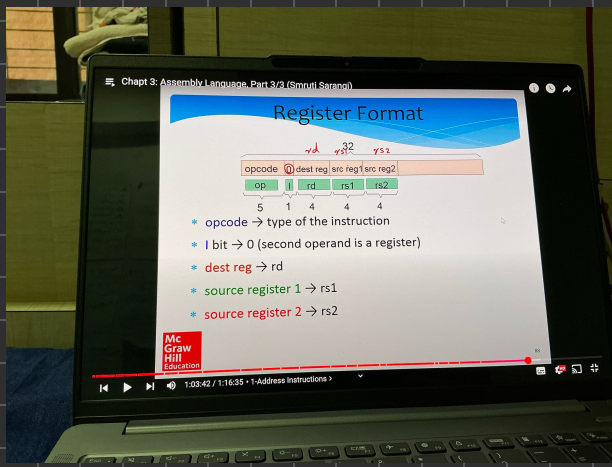


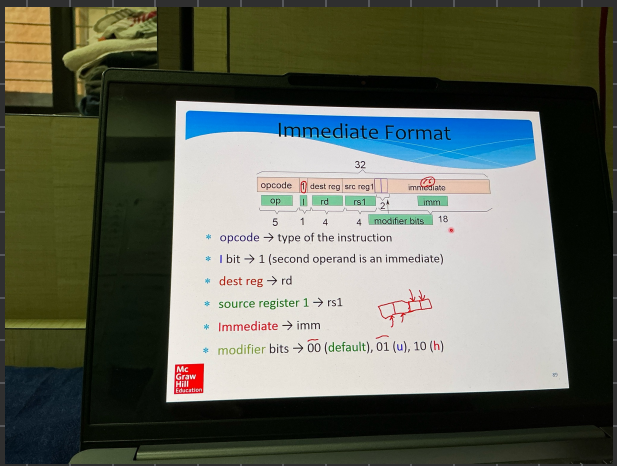
Note: If MSB = 1 ⇒ Branch instruction









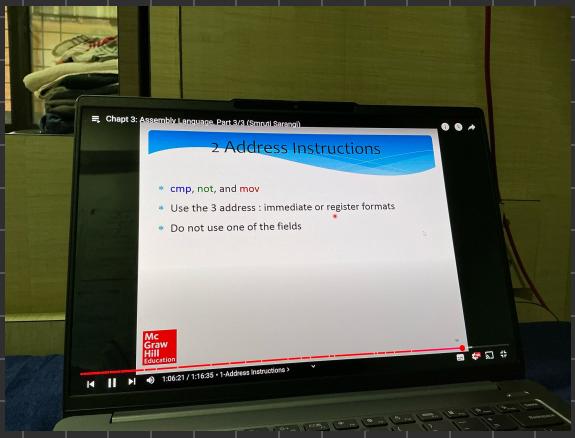


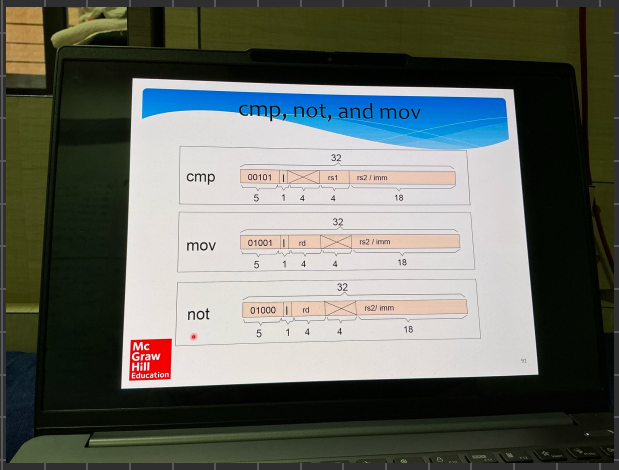
31:27

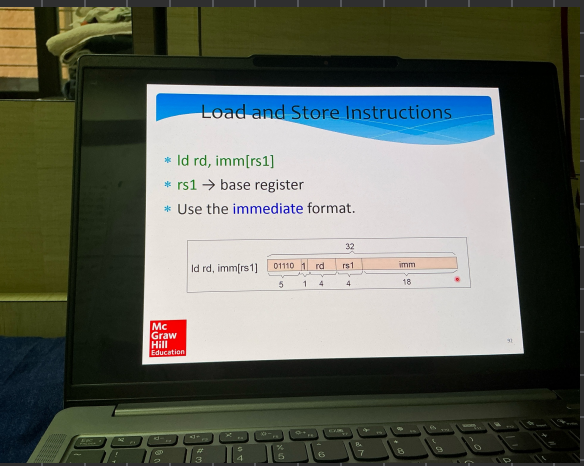
26

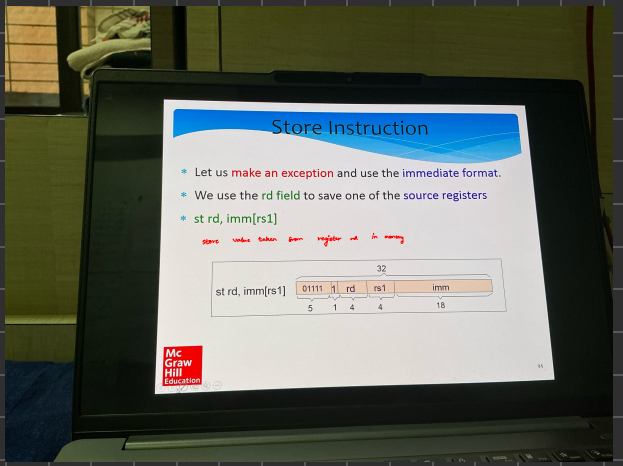
25: 22

21: 18









→ Only the store instruction has a destination in memory. it does not have a register destination.

Store instruction : 2 register sources

1 immediate

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

Mc Graw Hill Education