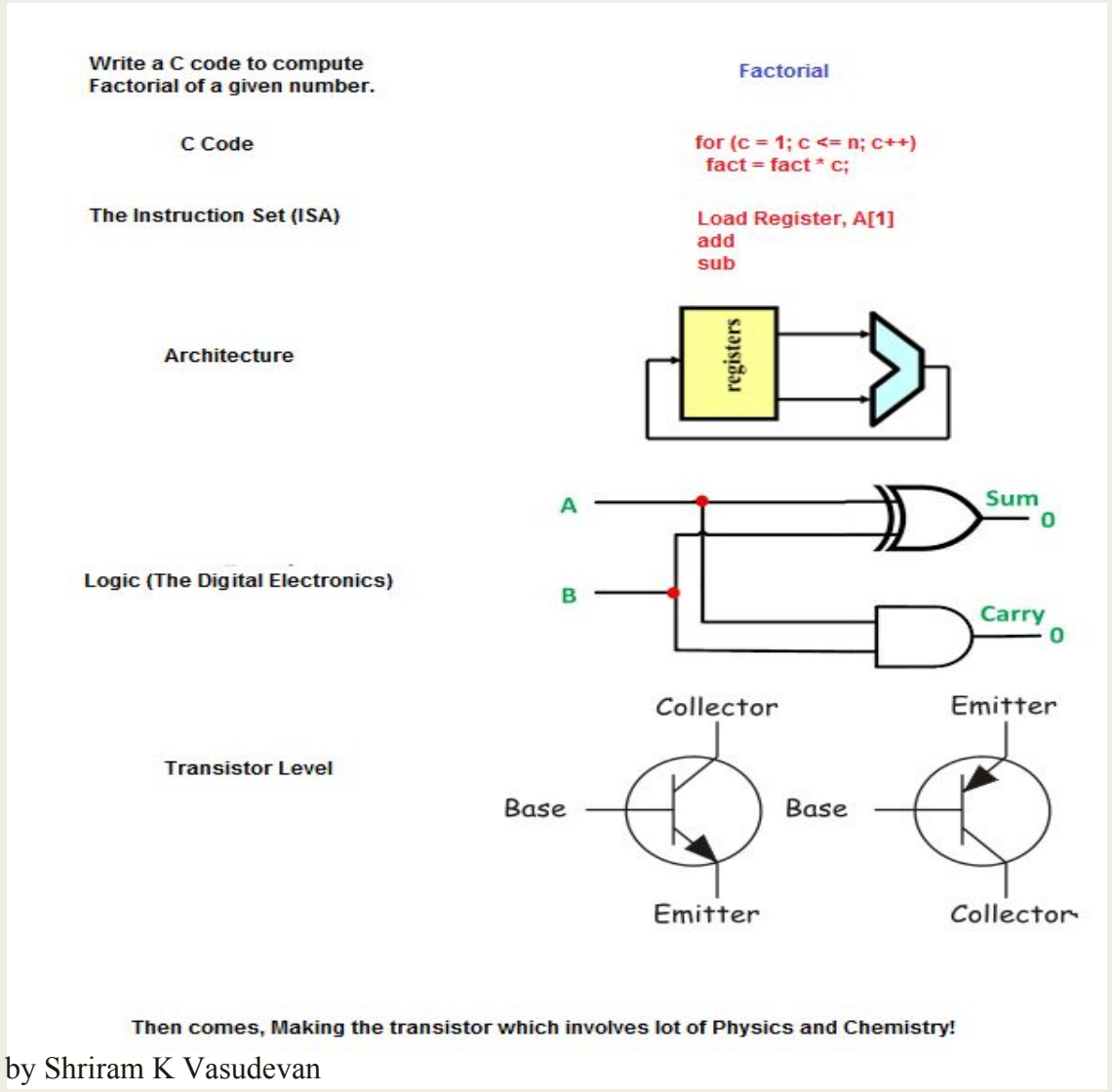
COMPUTER ORGANIZATION AND ARCHITECTURE

Shriram K Vasudevan

Session – 1

Can you look at this diagram for a min?



What is Computer Architecture and Organization?

- **Architecture will always concerned with what to do?**
- **Organization will take care of how to do?**
- **In simple terms, architecture is concerned with the design and organization is concerned with the building process.**
- **Architecture shall carry out the analysis and zeroing in about the instructions, modes of addressing and usage of registers.**
- **Computer architecture carries the precedence than Computer organization.**
  - *I.e. only after the design, the implementation can be done.*
- **It is to be compared with high level design.**
- **For example, if a calculator is built, it is to be decided if it will support the division operation is an architectural analysis. Likewise, all the features which a computer has are defined at architectural phase.**

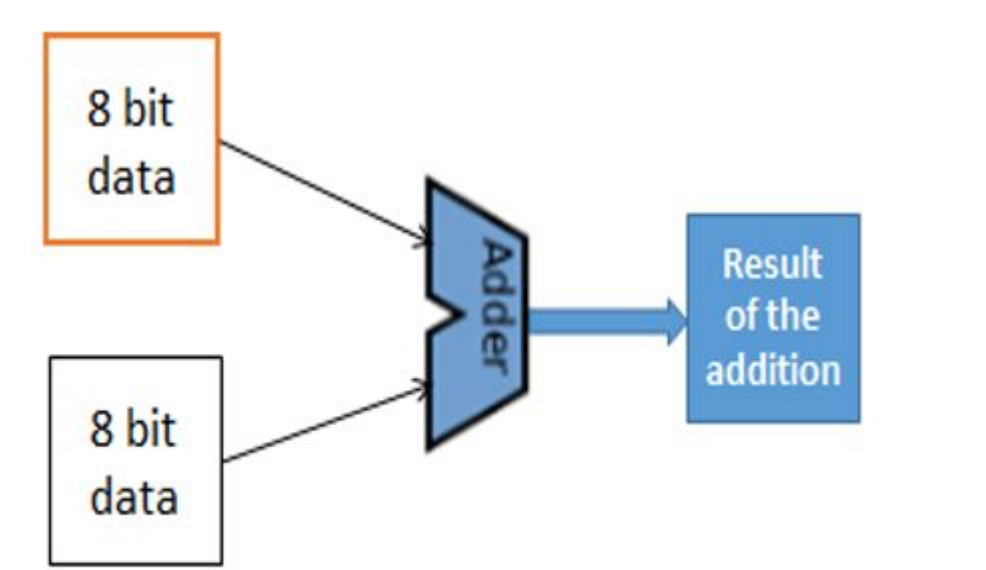
Contd.,

- Organization is the next phase to architecture.
- Or, organization is dependent on architectural decisions.
- Realization of the architectural decisions happens through the organization.
- The same example of Division operation is now to be seen from the organization perspective.
  - ***Having decided to support Division operation, organization shall enable to have the division done through the repeated subtraction or through division unit.***
  - *This decision of if to go with repeated subtraction or otherwise is the role of Organization.*
  - *Organization shall also help in identifying or uncovering design issues or challenges.*

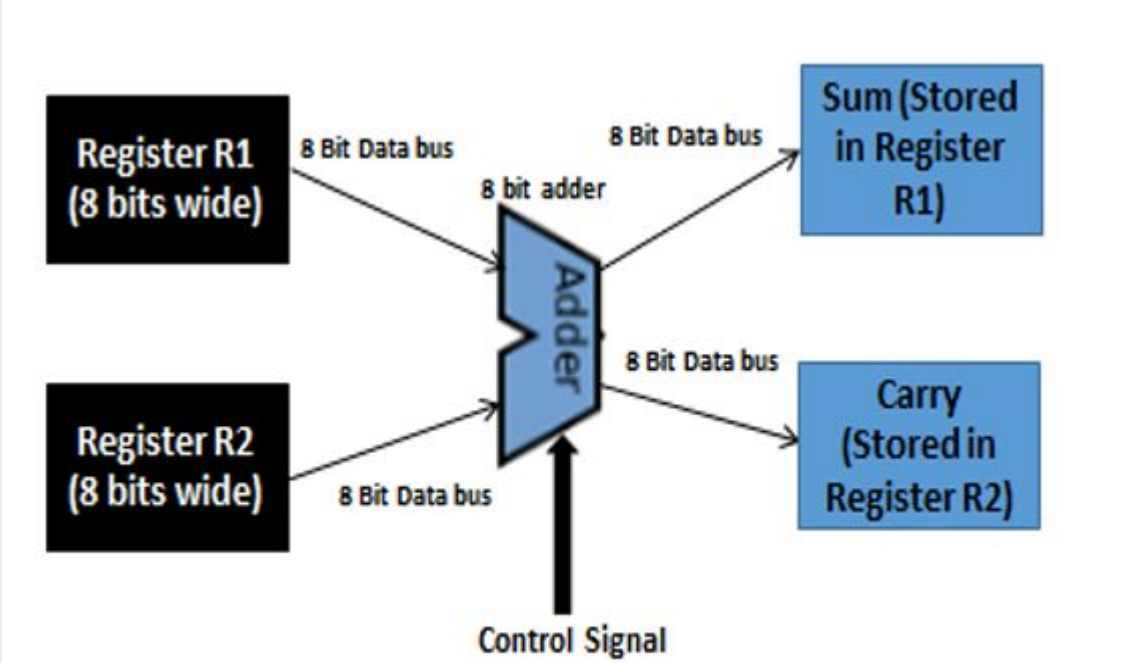
Let us compare!

S.No.	Computer Architecture	Computer Organization
1	High level design and feature description.	Realization of the high level design through implementation.
2	Connected to Instruction set, Support for data types, memory considerations etc.	More from implementation aspect is given importance here. I.e. selection of circuit components, peripherals, ALU implementation etc.
3	Logic will be given importance.	Implementation of the logic will be given importance.
4	What to do? – Will be the burning question.	How to do? – Will be the burning question.
5	Example: Decision to support the operation Multiplication	Example: Multiplication to be implemented through repeated addition. (8085 microprocessor works this way)

Hence, we can conclude this way!



Architecture to add two numbers and to store result.



Organization for adding two numbers and to store result.

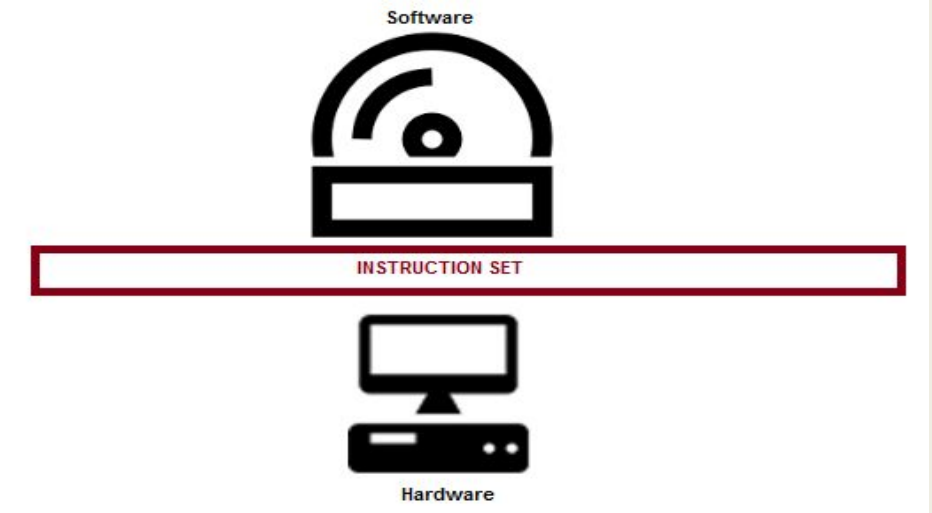
COMPUTER ORGANIZATION AND ARCHITECTURE

Shriram K Vasudevan

Session – 2

ISA – Instruction Set Architecture

- Instruction Set Architecture, ISA, is an interface between the hardware and software and it is indeed playing the role of an instructor to instruct the hardware about what to be done through the software instructions.
- In other words, ISA helps to utilize the hardware to its complete ability through the software instructions.
- It is a mediator and acts as a frontier amid the software and hardware.



Contd.,

- ISA helps in defining the operations, modes of operation supported, addressing, storage related inputs, how to use, perhaps, how to access the operands through instructions etc.
- In short, or more precisely, one could say that, a processor is detailed through the instruction set architecture.
- There are many processors and there could be many ISAs.
- The ISA would support multiple instructions and instruction types based on the processor selected.
- The instructions specified by the ISA shall be well interpreted by the processor and corresponding action shall be initiated. ISA is a collection of instructions and formats supported for the processor. Through ISA, i.e. instructions one can access the resources.

Contd.,

A = B + C (ADD A, B, C)

- Where,
  - *A has the result of the operation*
  - *B and C are the operands which are to be added. (B and C are called registers, which has values inside)*
  - *"+" is the operation, i.e. addition.*
  - *"=" is the assignment operation, after the expression is evaluated, the result shall be assigned to A.*
- **The same addition expression mentioned above shall be accomplished through the instruction ADD A, B, C where B and C are the source registers with A being the destination register.**

Contd.,

- Now, it is ISA's role shall come here in deciding the following very important considerations.
  - *What is the operation to be performed?*
  - *Where to store the operands?*
  - *Where to store the result of the operation?*
  - *How many operands are needed to perform (this) operation?*
  - *Does the ISA support the operation specified? (For an instance 8085 does not support multiplication and division).*
  - *What is the type of the operation? (it can be classified, reader shall be introduced to this)*
  - *What is the format of the operation?*
  - *Size of the operands.*
  - *What are the addressing modes supported? How to address the operands?*

COMPUTER ORGANIZATION AND ARCHITECTURE

Shriram K Vasudevan

Session – 3

Types of ISA – An understanding

- Some call it classes, some call it types.
- We can retain the term types of ISA for the discussion purpose. There are four types of ISA.
- They are listed as follows, one after another.
  - Accumulator
  - Stack
  - Register (General purpose)
  - Register (Load and Store)

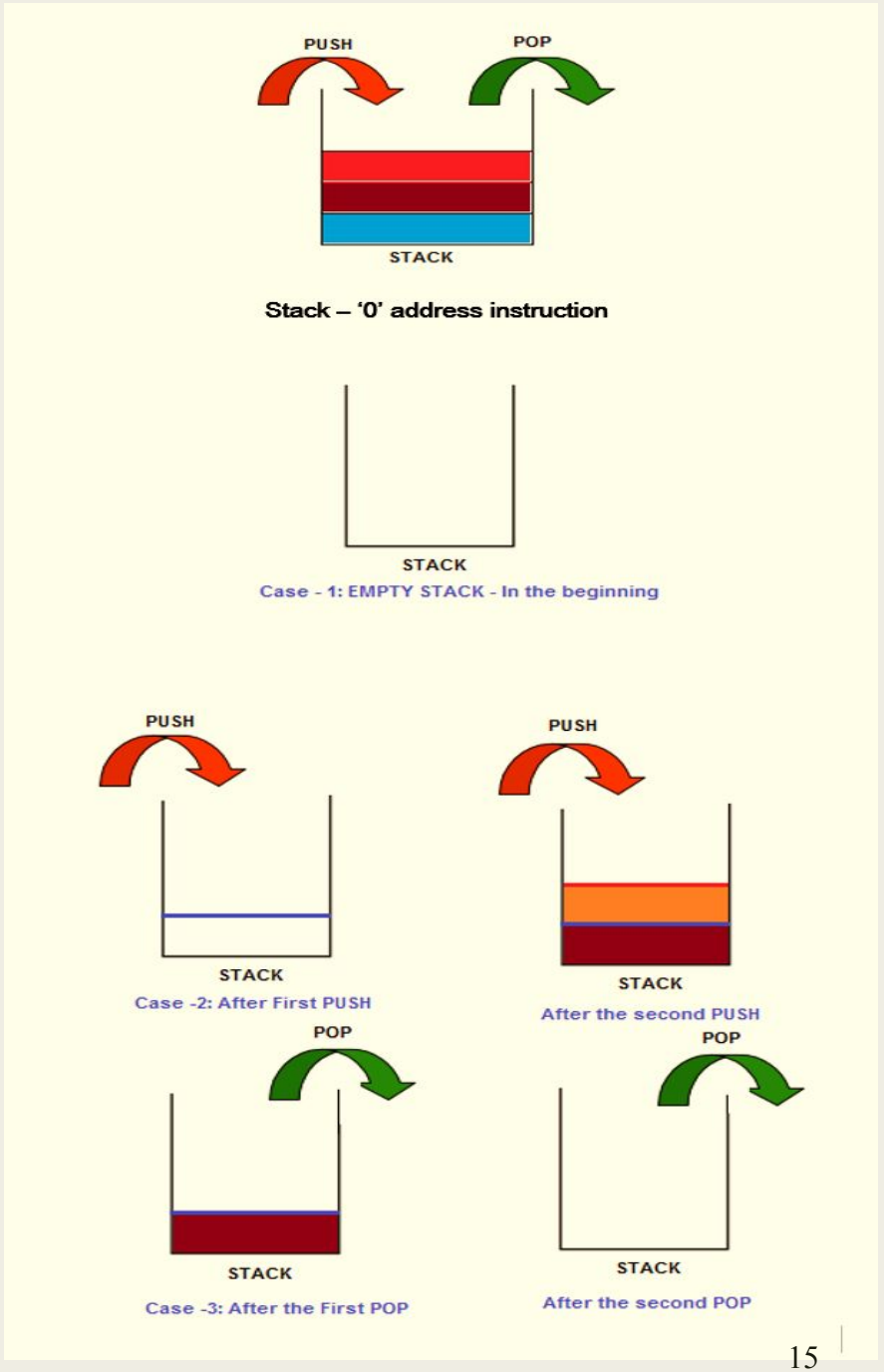
Contd.,

- **Accumulator:**
- This is normally referred as 1-address type of ISA.
- In this approach, the content from a memory location or a specified register shall be added with the content available in the accumulator register and the result obtained shall also be stored back into the accumulator.
- Here, in this approach, accumulator is the main component and carries more focus. This approach is accumulator centric.

```
ADD R1;

ACC = ACC + Content [Memory or R1]
```

Contd.,

- **Stack:**
- This is often regarded as 0 address instruction.
- Here, the user need not have to mention any operand and the instruction is operand less.
- Means, a PUSH instruction shall push the content into the stack and POP shall pop the content out from the stack to a register.
- The contents shall be added to the stack through the reference to the top of the stack.
- The instructions to be remembered are “PUSH and POP”. This is a very simple approach and effective too.



Contd.,

- Register: (general purpose approach)
- This is an approach where, general purpose registers shall be involved in the operation.
- Here, in this approach the contents from the registers (operands) shall be used and when the result is available shall be stored in another register.
- It could even be accumulator to store the result.
- Here, there are two options. One, there can be 2 operands.
- Secondly, there can be three operands.

ADD A, B

ACC = ACC + Content [B]; Two address instruction.

ADD A, B, C

REGISTER A = Content [B] + Content [C]; Three address instruction.

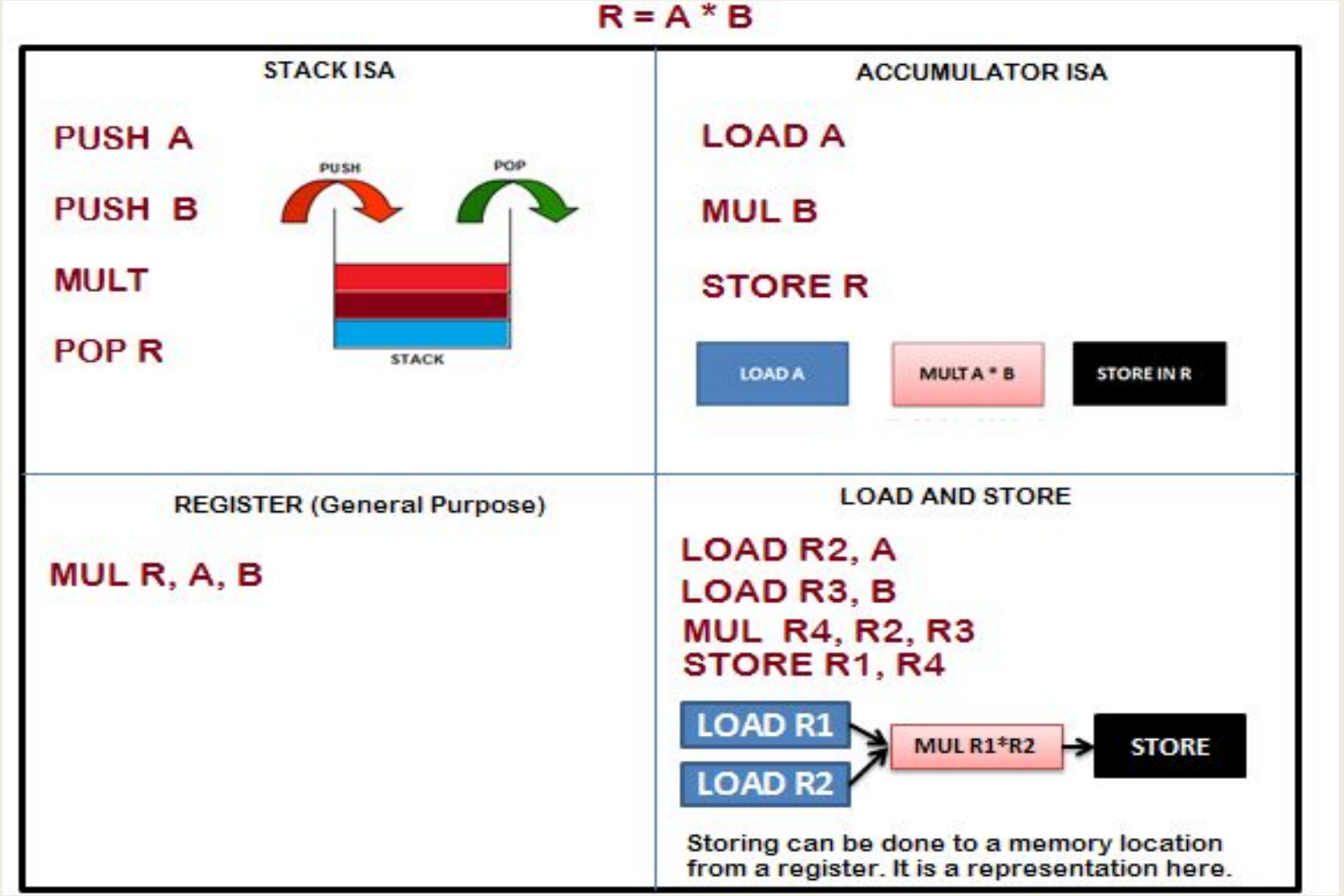
Contd.,

- Register (Load and Store)
- This ISA will demand the operands to be moved to the registers and only then the operation can be carried out.
- For an instance, if you want to add two numbers, first, the two numbers (operands) should be stored in the registers and then the ALU can be fed with the input.
- LOAD is used to move the data from the memory location to registers and STORE does it reverse.
- Anything and everything in this approach shall be done through the registers and no direct memory access is encouraged.
- This is what is referred as "RISC" architecture.
- It can be represented as 2-address or 3 - address instruction based on the requirement.

```
LOAD $R1, @Location 1000; Here, Register R1 is loaded with content from 1000.

LOAD $R2, @Location 1200; Here, Register R2 is loaded with content from 1200.

ADD R1, R2                    ; Multiply the content from R1 and R2. Store result in R1

STORE @Location 1300, R1; Store the result at 1300 from R1.
```

Contd.,

- **Expression to be implemented with all the four ISA types: R = A *B**



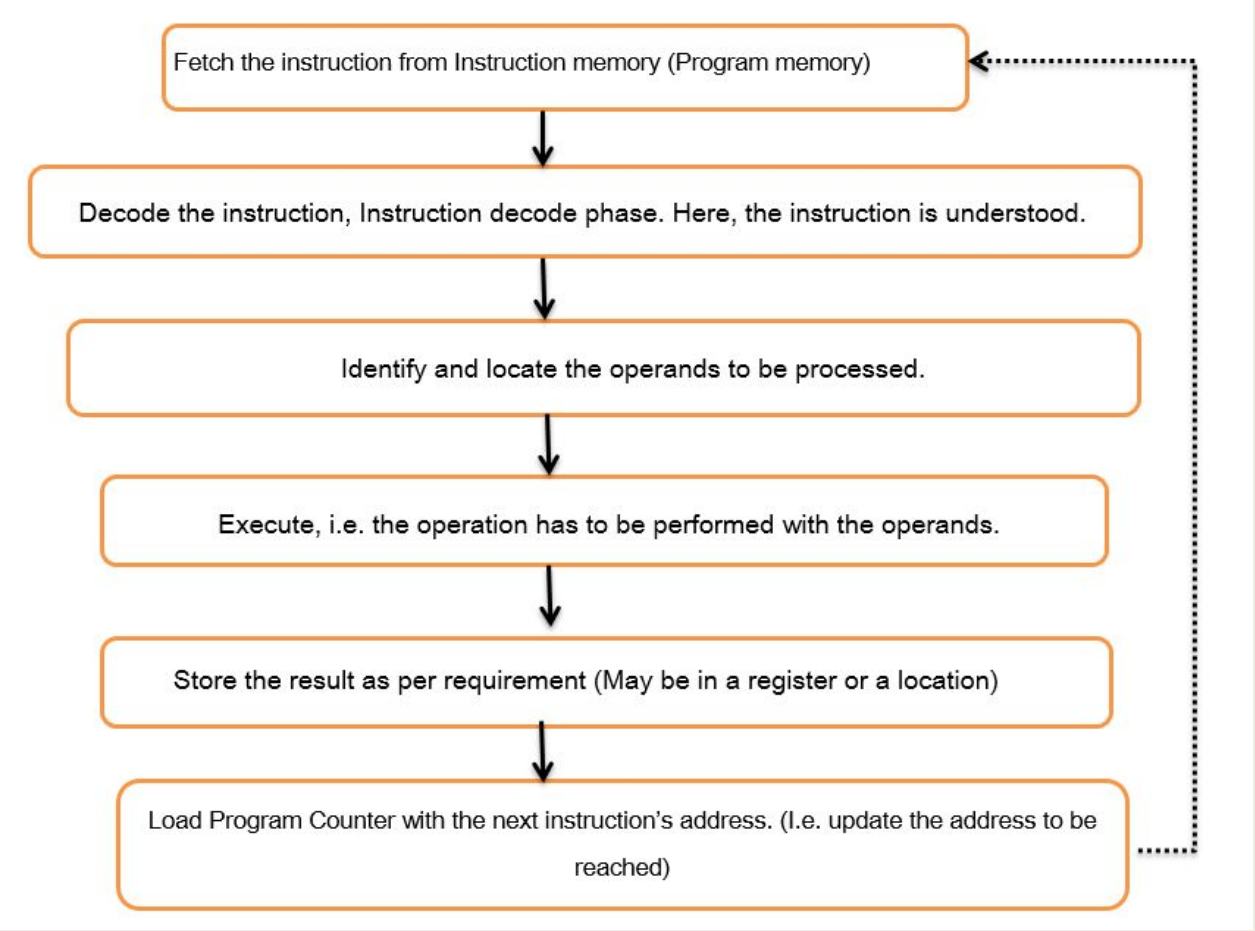
COMPUTER ORGANIZATION AND ARCHITECTURE

Shriram K Vasudevan

Session – 4

Instruction Execution Cycle and Terms Recap.

- A program is nothing but collection of instructions and all the instructions gets pulled one after another from the memory (don't worry which memory we are talking about, it is just memory).
- Then, the process of understanding what the instruction is all about begins.
- Followed by this, there would be computation and storing the result.
- One can simply call this as Fetch, Decode and Execute, which is more technical as well.
- But, to establish a better understanding it is needed to go with a deeper flow. (see RHS)



Some fundamental terms for recap…

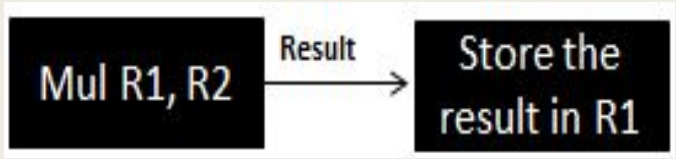
- **Program counter:** A register which holds the address of the next instruction (subsequent instruction) to be executed. This enables faster execution. Program counter is a register and it cannot be accessed by the programmer. Or, the programmer does not know the address of the program counter. It is purposefully hidden from the programmers view.
- **Stack and Stack Pointer**: Stack is a temporary storage area where the values can be stored. For an instance when a function is called like add (3,4), the arguments 3,4 would be kept in the stack (the operation is called pushing) and it is a temporary storage area. Once the function call is done and result is ready, the result will be again kept on the top of the stack. Once the execution is complete, the stack can be cleaned (the operation is referred to as popping). Stack pointer is a register used to point the stack.
- ***Bus:*** The channel through with the information flows. The data could be the address or the data from a particular location. If the bus carries the data, it is called data bus and in case addresses go through the bus, it is referred as address bus. There are control signals to be sent for controlling actions and in that case, the bus is referred as control bus.

CISC VS. RISC

SHRIRAM K VASUDEVAN

Recap from previous semester.

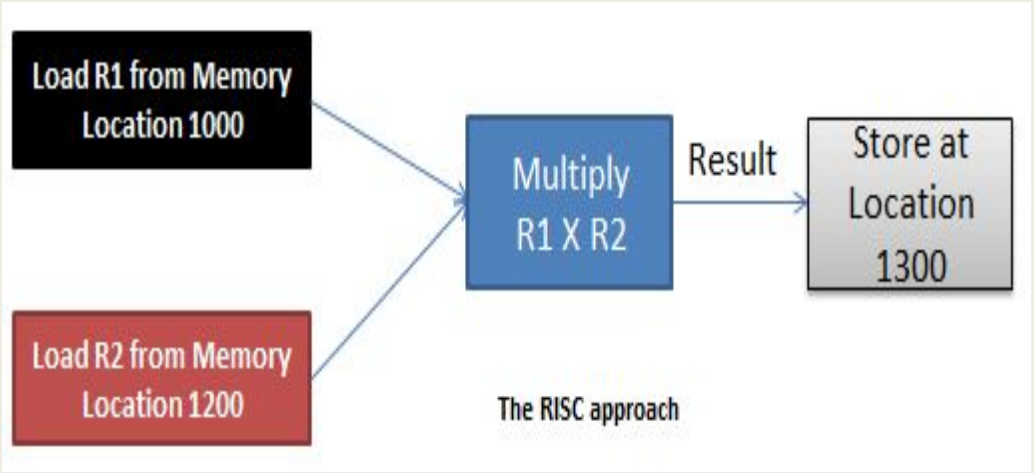
INTEL



The CISC Approach

MUL R1, R2 ; Multiply the content from R1 and R2. Store result in R1

APPLE COMPUTERS



LOAD $R1, @Location 1000; Here, Register R1 is loaded with content from 1000.
LOAD $R2, @Location 1200; Here, Register R2 is loaded with content from 1200.
MUL R1, R2 ; Multiply the content from R1 and R2. Store result in R1
STORE @Location 1300, R1; Store the result at 1300 from R1.

So…

- *Intel believes in hardware bearing more responsibility, making the coding simple. I.e. the developers shall be writing minimal lines of code. Apple believes the other way round. The philosophy is, hardware has to be simple and software has to lift most of the responsibility. Means, the developers will have a tough job of writing more lines of code with this architectural approach*

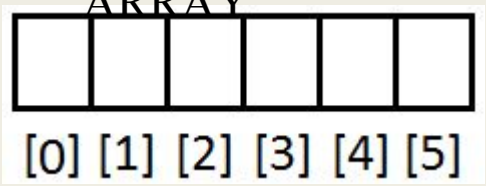
S.No	RISC	CISC
1	RISC is more oriented towards complexity in software. Programmer will write more lines of code to accomplish a task.	CISC is oriented more towards hardware complexity, where programmer will write limited lines of code. Load the hardware not the programmer is the logic.
2	One clock cycle is based instructions	Clock cycle time is a variable with CISC instructions.
3	LOAD and STORE approach is mandatory.	LOAD and STORE not explicit. Instructions are capable to fetch the content from the location to registers.
4	Since LOAD and STORE requires lot of registers to be used in the code, more registers are present in this architecture.	Limited registers. That too, they could be utilized as general purpose registers.
5	Pipelining is easier as an operation would be accomplished with many instructions. For Example, Multiplication is done with 4 instructions.	Not easy with respect to pipelining.
6	Not many formats of instructions. Uniform instruction sizes.	Multiple formats are supported.
7	Limited addressing modes	Many addressing modes supported as instructions are different formats.

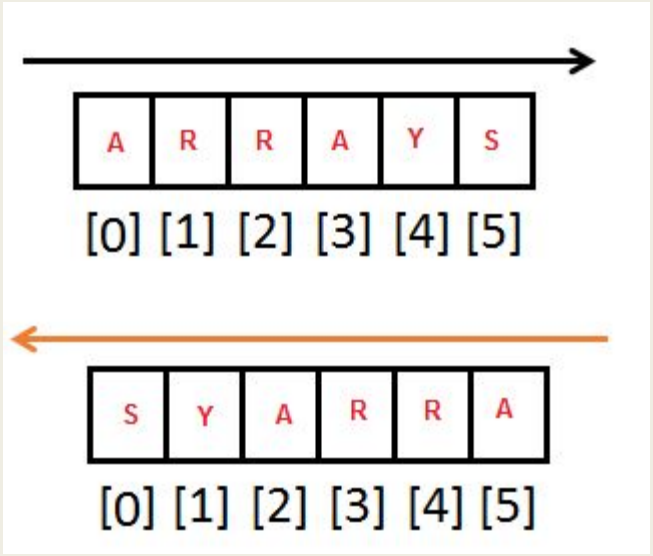
LITTLE ENDIAN AND BIG ENDIAN FORMATS – A CLEAR UNDERSTANDING.

Shriram K Vasudevan

Recap from previous semester.

AN EMPTY ARRAY

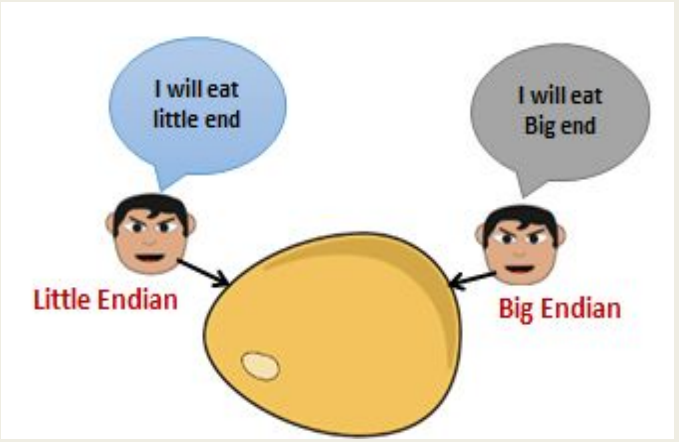




Ordering (Endianness)	Base Address + 0	Base Address + 1	Base Address + 2	Base Address + 3	Base Address + 4	Base Address + 5	Base Address + 6
Big – Endian	A	R	R	A	N	G	E
Little – Endian	E	G	N	A	R	R	A

The advantage!

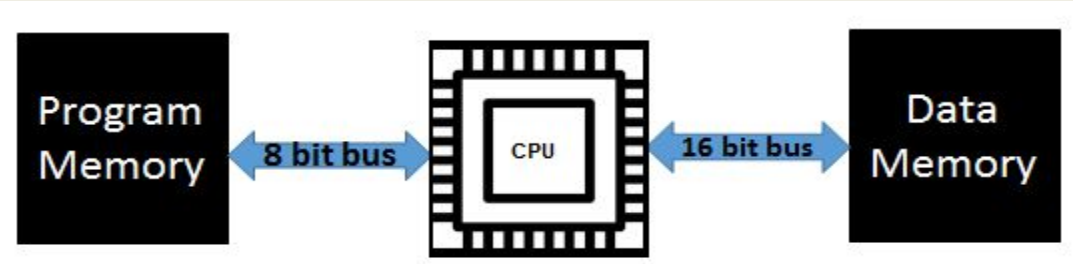
S.NO	Little Endian	Big Endian
1	MS Paint	JPEG
2	RTF	Adobe Photoshop
3	BMP	Mac Paint



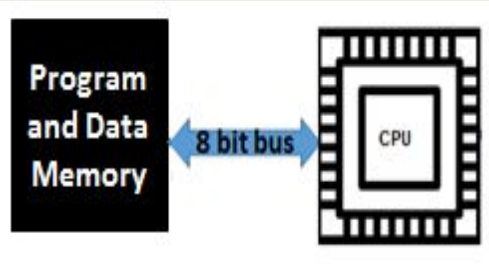
HARVARD AND VON-NEUMANN ARCHITECTURE – LET'S UNCOVER THE JARGONS

Shriram K Vasudevan

Recap from previous semester.



Harvard Architecture – A diagrammatic representation



Von Neumann Architecture – A diagrammatic representati

S.No.	Harvard Architecture	Von Neumann Architecture
1	Separate program and data memory, which eventually means separate buses available for data and memory.	The data and program (instruction) are stored in the same memory which implies that there is only one bus being used.
2	Since the buses are separate, the bus width can be different for data and instruction.	Not possible to have different bus widths.
3	Operations with both the memories (data and instruction) can be carried out simultaneously as there are no dependencies.	Not possible to have simultaneous processing
4	No need to think much about scheduling and stuff as the memories and buses are independent.	Scheduling considerations are to be given as the resources are limited from the bus and memory view.
5	Not much of bandwidth related constraints or challenges.	Bandwidth related constraints could rise as the resources are limited.
6	ARM9 follows Harvard architecture	Pentium processors follow Von-Neumann architecture.

Harvard vs. Von-Neumann Architecture

THANKS

Shriram K Vasudevan

ADDRESSING MODES

Shriram K Vasudevan

Remember

- *Addressing mode*: The rule for interpreting or modifying the address field of an instruction.
- *Effective address*: The address of operand produced by the application of the rule for interpreting or modifying the address field of the instruction before the operand is actually referenced.

Addressing Modes

How do I travel to the destiny?

- Common addressing modes are:
  - Immediate
  - Direct
  - Indirect
  - Register
  - Register Indirect
  - Displacement (Indexed)
  - Stack

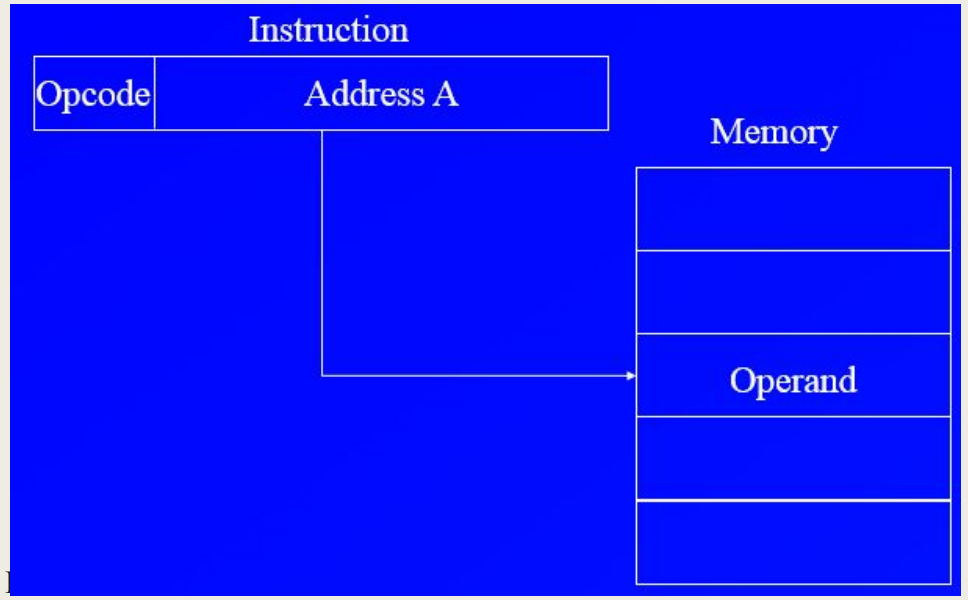
Immediate Addressing

Mode	Description	Example
Immediate addressing mode	Simplest mode, where the instruction itself will have the information to be moved or to be processed. The data to be processed can be moved to any register or memory location.	◆ **MVI A,05H** - 05H is the data to be moved to Accumulator with MVI instruction which is abbreviation of Move Immediate. Similarly immediate addressing mode can be used in arithmetic operations as well. Simple example would be: ◆ ADI 01H, where 1 is added to the content already available in the accumulator.

Instruction

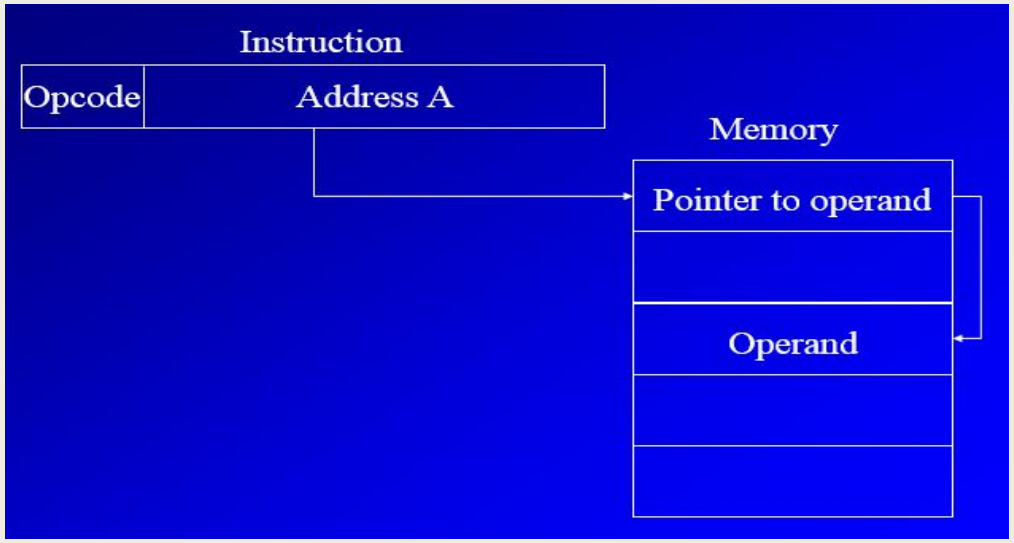
Direct Addressing Mode.

Direct addressing mode	In this mode one of the operands will be the address itself where the information is available	• **LDA 2000H** – Where, 2000H is the address of the source and LDA is the instruction which is expanded as load accumulator. The contents available in the memory location 2000H will be stored in accumulator once this instruction is executed. • **STA 2000H** – This is yet another instruction which falls in the category of direct addressing more. This instruction prompts the microprocessor to store the content of the accumulator into the specified address location 2000H.



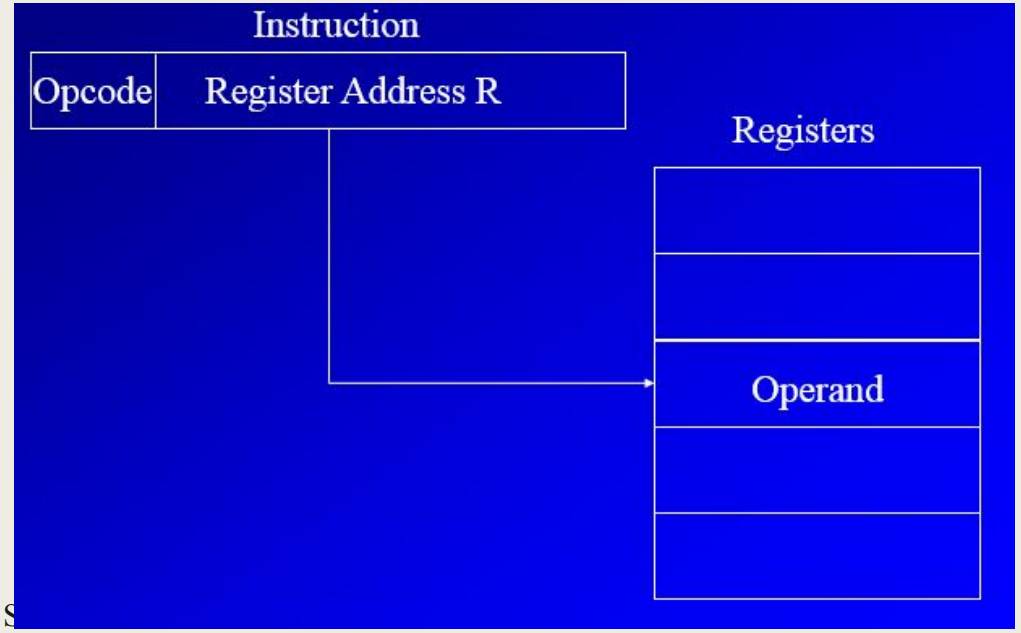
Indirect Addressing mode..

Indirect addressing mode	In the previous mode, address was specified right away, but here the place where the address is available will be specified.	**LDAX D** - Load the accumulator with the contents of the memory location whose address is stored in the register pair DE



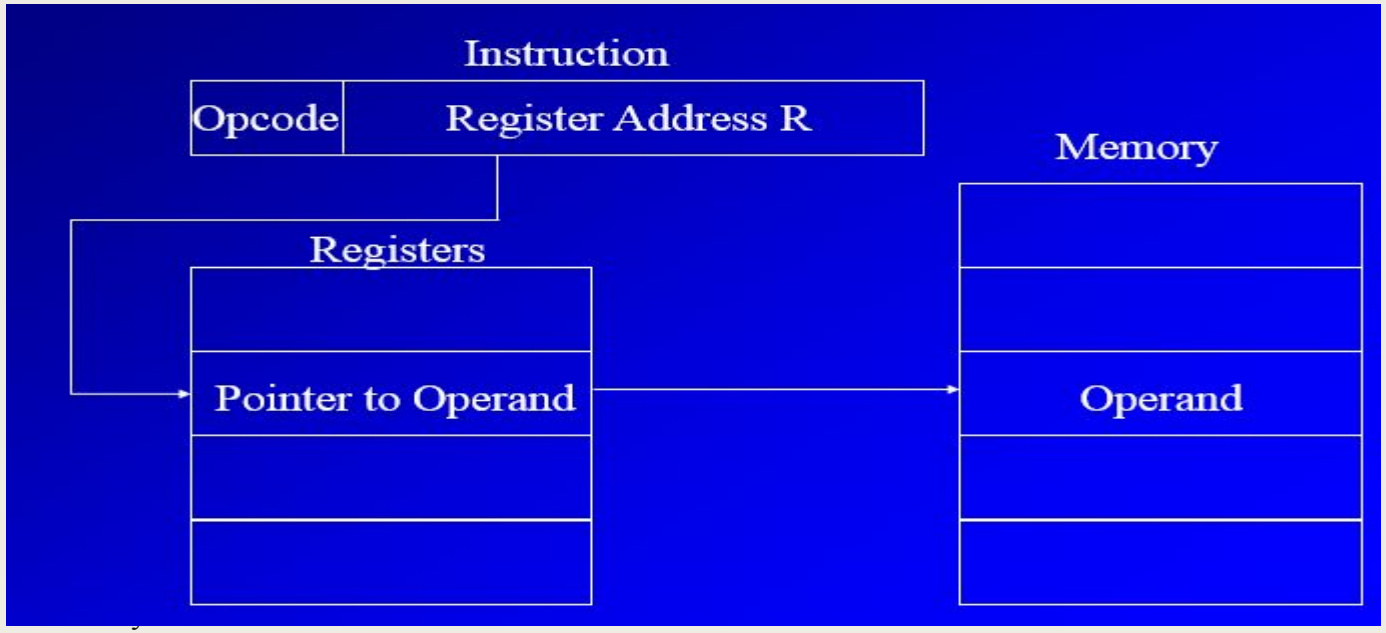
Register Addressing Mode.

Register addressing mode	Here the instruction will have the names of the register in which the data is available.	**MOV Rd, Rs** – **Rs** is the source register and Rd is the destination register. Data will be moved from source to destination with this instruction. One simple instance would be MOV C, D where the content of D is moved to the register C. Meanwhile the same addressing mode is applicable for arithmetic operations. **ADD C** will add the content of register C to accumulator and will store the result again in accumulator.

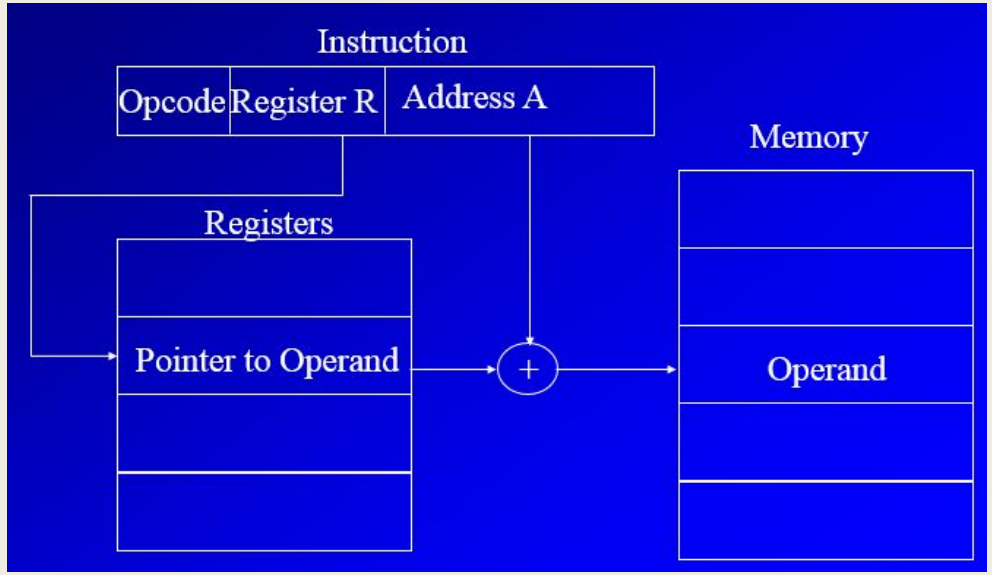


Register Indirect Addressing

- EA = (R)
- Register indirect addressing is similar to indirect addressing, except that the address field refers to a register instead of a memory location.
- Operand is in memory cell pointed to by contents of register R



Displacement Addressing (indexed)



- EA = A + (R)
- Address field hold two values
  - *A = base value*
  - *R = register that holds displacement*
  - *or vice versa*
- A very powerful mode of addressing combines the capabilities of direct addressing and register indirect addressing, which is broadly categorized as displacement addressing

Stack addressing mode..

- Operand is (implicitly) on top of stack
- e.g.
  - *ADD Pop top two items from stack and add*

LET US GO TO MIPS

Shriram K Vasudevan

Session 5

What is MIPS?

- It is not Million Instructions Per Second, here! ☺
- Expanded as Microprocessor Without Interlocked Pipeline Stages.
- It is an ISA with 32 bit instructions. (Means, all instructions are 32 bits wide)
- It is completely RISC architecture. (Means, it is all about load and store)
- Any RISC architecture would eventually support FIXED WIDTH instructions and it is actually beneficial (You will understand this soon)
- A very important point to note is "All these instructions must be stored at "WORD-ALIGNED address".
  - *What is that?*
  - *Word aligned address means, the addresses are divisible by 4. (Must be, we shall learn more about this, shortly)*

MIPS – How it all happened? From where did we reach MIPS?

- How did it grow? It started with Motorola and Intel.
  - ***Motorola 6800 / Intel 8085 (1970s)***
    - 1-address architecture: `ADDA` <*addr*>
    - (A) = (A) + (addr)
  - ***Intel x86 (Early 80s)***
    - 2-address architecture: `ADD EAX, EBX`
    - (A) = (A) + (B)
  - ***MIPS architecture (Early 90s)***
    - 3-address architecture: `ADD $2, $3, $4`
    - ($2) = ($3) + ($4)

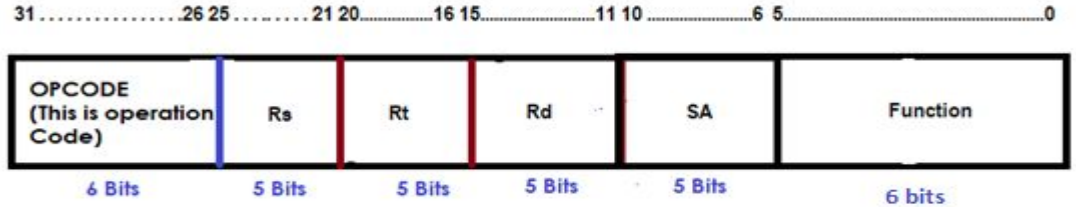
Let’s understand this as well.

- MIPS has the following design considerations.
  - It is designed with having high level languages in mind. Means, usage of high level languages is much easier with MIPS.
  - MIPS does not demand the programmer to learn many addressing modes or instructions. It is all simple here and one could learn MIPS with ease.
  - Increased parallelism and also classy pipelining supported in MIPS. (Remember the production lines)
  - 100% Load and Store architecture.
  - Increased flexibility with providing more number of general purpose registers (Compare ARM with 8085 folks)
  - All instructions are 32 bits wide. Hence, there is a uniformity internally enabled.
    - Hardware design considerations are easier when it is all uniform.

MIPS Instructions

- MIPS instructions are of the following types:
  - *Arithmetic/logical/shift/comparison*
  - *Control instructions (branch and jump)*
  - *Load/store*
  - *Other (exception, register movement to/from GP registers, etc.)*
- Let us learn this one after another.
- Again, classified as I Type, R Type, J Type instructions.
  - *All the instructions shall fall under any of these instruction types.*

Let us understand how it looks like…



MIPS supports 3 address instructions as we know. **(2 Source Registers and 1 destination register)**

Simple Example: Add instruction to be referred.

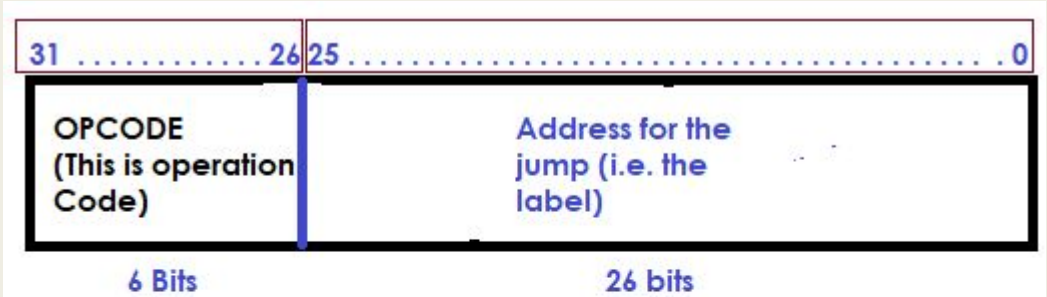
- **add $s1, $s1, $s2**

  rd rs1 rs2

Opcode - Operation Code
Rs - Source Register
Rt - Can be used as Source / Destination.
Rd - Destination
Function code - 6 bits. Specifies function.
SA - Shift amount.

Immediate (I-Type)
Opcode - 6 bits code specifying what operation is being carried out.
Rs - Source Register - 5 bits wide.
Rt - Targer Register - 5 bits wide.

R Type



J is jump type instruction.
Opcode + Address (6 bits + 26 Bits)
Example:
add a + b;
jmp label; // On seeing this instruction control will be transferred to the label.
sub a – b;

DESIGN RULES – PART 1

Shriram K Vasudevan

Session 6

Let us learn the instructions. But, with some design rules in place.

- A question that anyone can answer is to be asked.
- Can I speak with you in Latin and will you understand (Assuming you do not know Latin)?
- Answer is overt. And yes, you wont understand.
- Hence, it is important to communicate to someone in the language he/she understands.
- What is computer's language?
  - *Undoubtedly, it is instructions.*
  - *All instructions are grouped under one housing unit called Instruction Set. What we will see in this chapter?*
  - *So, what next?*
  - *We should learn how to talk to a computer and how computer could interpret it as well.*
  - *We can cite this as "Secret of Computing".*

MIPS writing / representation notations (This is like the coding guidelines)

- We can take / cite some addition operation in this section to get the understanding constructed, solid.
- Addition is the operation which we shall talk about as it is much easier and straight forward than anything else.
- The instruction could be – add x, y, z
  - *What this does?*
  - *Simple, add y and z. Store the result in X.*
  - *This is called MIPS representation.*
- Can we have any other notation to represent MIPS?
  - *Come on. Not possible.*
  - *Extremely inflexible when it comes to MIPS.*
  - *Each operation will have 3 operands for sure. (means X, Y, Z)*
  - *Also, each operation could only perform one operation.*
    - Means, it is addition. Only addition.
    - ***MSD is a batsman. No bowler. One action!***

Lets add some fun…

- Challenge – add the values at Q R S T registers and place the result in P.
- How MIPS could do this?
- Considering the restrictions thrown in the previous slide, we should get a solution.

```
ADD P, Q, R     # Here, Q and R gets added, result goes to P.
ADD P, P, S     # Previous result gets added with S. Result stored in P.
ADD P, P, T     # Previous result gets added with T. Result stored in P.
```

- Thus, we require 3 instructions to add 4 numbers (Costly, huh??)
- MIPS guidelines say, each line can have one instruction, the max.
- Comments always terminate at end of the line.

Just, MIPS things.

Fantastic Four – Design Rules

- Rule 1 – Keep it simple. (Referred as Simplicity offers Regularity)
- It is all simple with MIPS. We can now do something interesting.

```
a = b + c;
d = a - e;
```

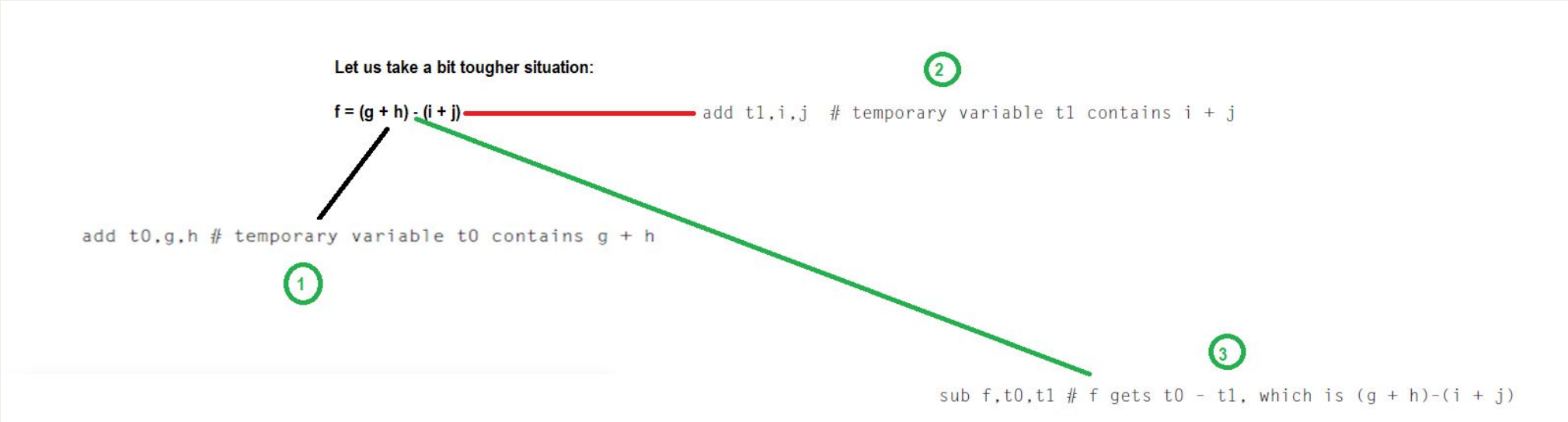
This is C Programming approach…

```
add a, b, c
sub d, a, e
```

The MIPS way..

Inference: Directly we mapped!

Contd.,



So it is easy with MIPS. Simplicity rendered to perform operations as good as it is in high level languages is an added advantage.

Arithmetic	add	add a,b,c	a = b + c	Always three operands
	subtract	sub a,b,c	a = b - c	Always three operands

Lets go a bit deeper…

- C or CPP has a facility which MIPS does not have. That is, storage luxury.
  - *There are very limited storage options and register's usage is inevitable.*
- Registers are going to be the heart, kidney and liver.
- They are termed even as bricks of computer organization.
- As discussed earlier, the size of MIPS registers is 32 bits.
- In MIPS architecture, 32 bits together is termed as WORD.
  - *A register is capable of storing a word is the point to be taken.*

Inference

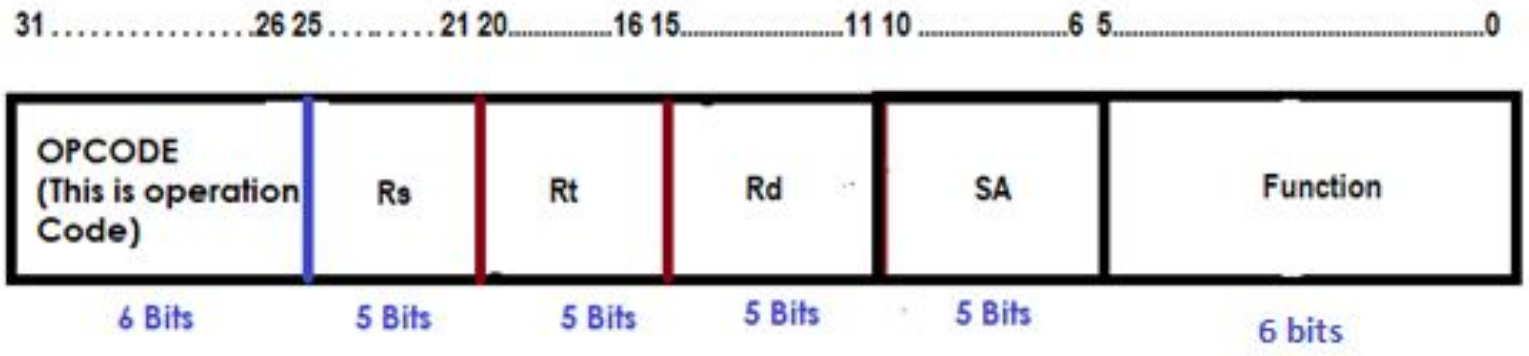
- MIPS has 32 registers. (Everything is 32 here!)
- So what?
- **We have added the restriction that the three operands of MIPS arithmetic instructions must each be chosen from one of the 32 32-bit registers.**

Rule – 2
Make it smaller (As it would be faster)

- If more number of registers are there, it would need more clock cycle time.
  - *Why?*
  - *Simple. It will take electronic signals longer when they have to travel farther.*
- Also, 31 Registers may not be faster than 32 ☺
- There are notations to be followed in MIPS architecture.
- $ is used to represent a register.
- **$s0, $s1 For Registers.**
- **$t0, $t1 Temp**

clock cycle is the time between two adjacent pulses of the oscillator that sets the tempo of the computer processor.

Let us learn what R Type instructions are all about!!! Fun Ride!!!!



MIPS supports 3 address instructions as we know. **(2 Source Registers and 1 destination register)**

Simple Example: Add instruction to be referred.

- **add $s1, $s1, $s2**

  rd rs1 rs2

Opcode - Operation Code
Rs - Source Register
Rt - Can be used as Source / Destination.
Rd - Destination
Function code - 6 bits. Specifies function.
SA - Shift amount.

For R Type Instruction Remember the following stuff:

- **OPCODE IS ALWAYS 0.**
- **"add" and "sub" both are R type instructions. Opcode for both of them is 0.**
- **Compiler would differentiate the operations based on the function field (Operation code).**

Contd.,

- A Simple Home Work!! (Not really folks)

$s0, $s1 → For Registers.
$t0, $t1 → Temp
$s0 to $s7 map onto registers 16 to 23, and registers $t0 to $t7 map onto registers 8 to 15

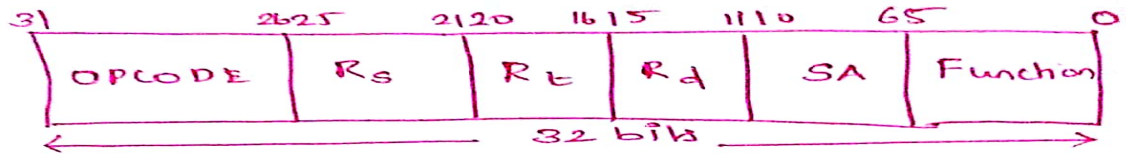
AN EXAMPLE :

REMEMBER THIS :
$s0 – $s7 = 16 to 23
$t0 – $t7 = 8 to 15

MIPS ASSEMBLY INSTRUTION TO MACHINE LANGUAGE

ADD $t0, $s1, $s2

R – Type Format :



Add $t0, $s1, $s2

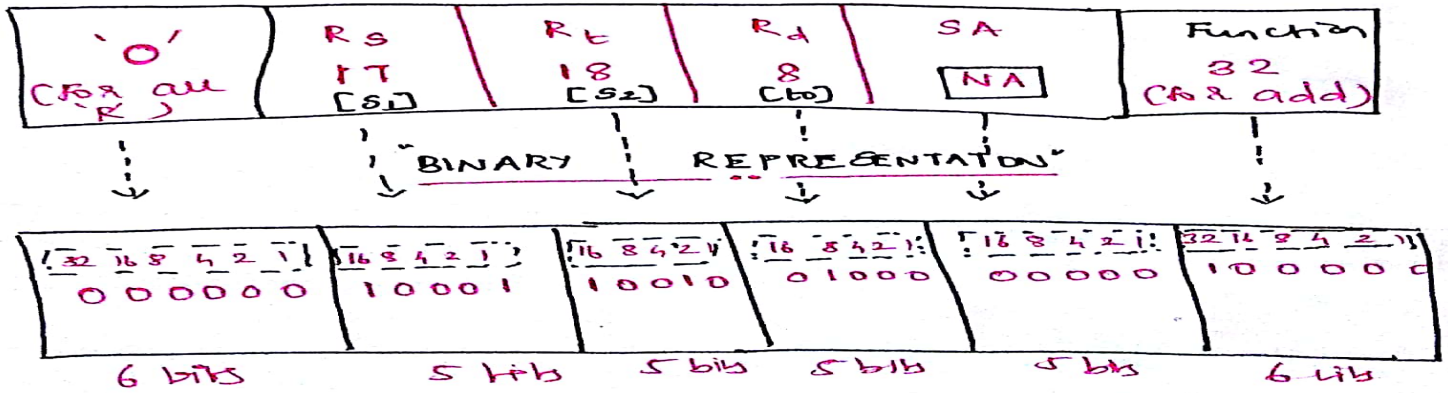
$t0 = $s1 + $s2

↓

Rd = Rs + Rt

So,

"DECIMAL REPRESENTATION"

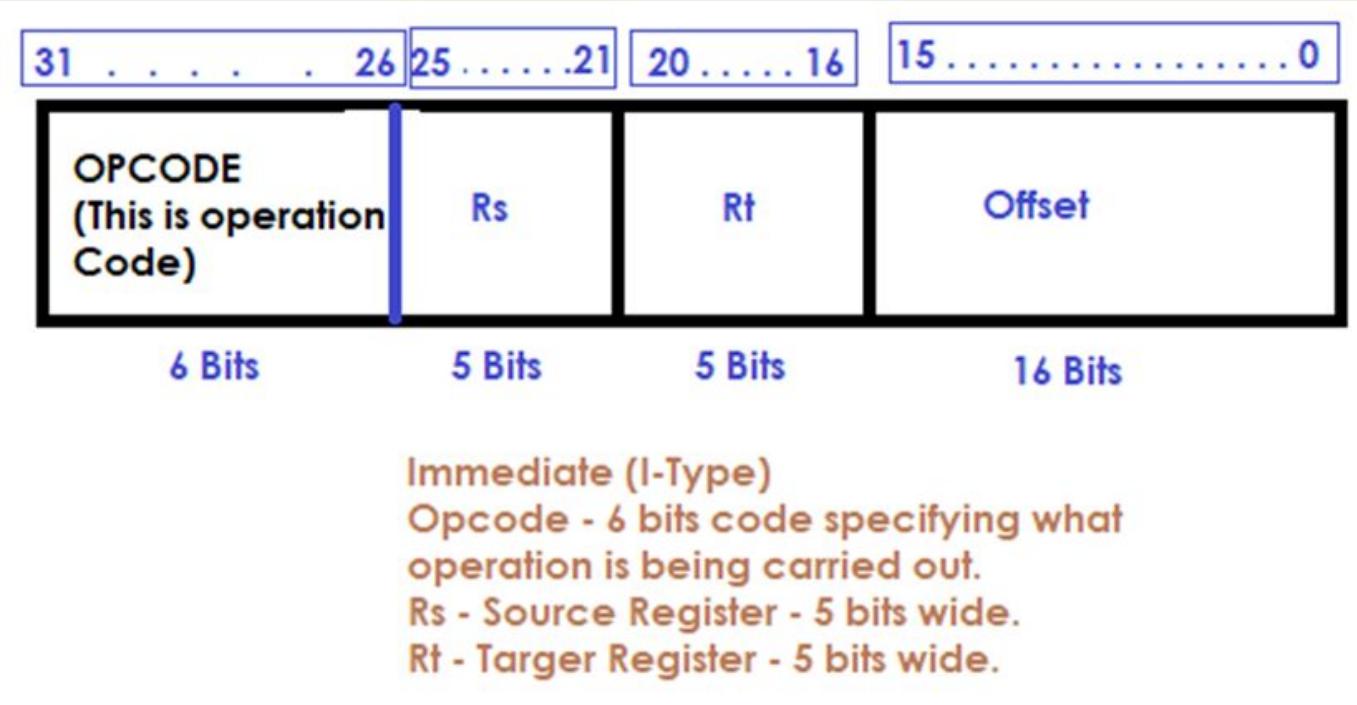


Understand this as well…

- Are we and computer the same when we look at a number?
  - *No, Not at all. We are unique!*
- Humans are capable of thinking everything as base 10 (Decimal).
- Computers know only binary (0s and 1s)
- 25 base 10 = = 11001 base 2. (16 8 4 2 1)

registers $s0 to $s7 map onto registers 16 to 23, and **registers** $t0 to $t7 map onto registers 8 to 15.

Lets learn I Type Instructions (Once this is done, we are half done)



- **Load Word and Store Word are the two instructions to be referred under this category.**
- Load -> Loads the data from the memory location specified to the register.
- Store -> Takes the value from the register and stores in the specified memory location.

$s0, $s1 → For Registers.
$t0, $t1 → Temp

$s0 to $s7 map onto registers 16 to 23, and registers $t0 to $t7 map onto registers 8 to 15

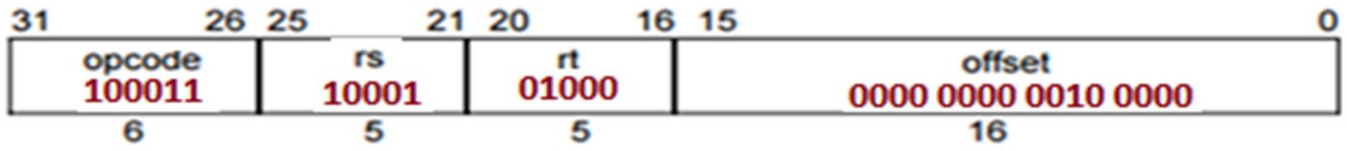
An instance:

- **lw $to, 32 ($s1)**
  - ***32 is referred to be immediate value.*** *(value being part of instruction is* ***immediately enforced)***
  - ***Assuming $s1 has 500, 500 + 32 = 532 will be the address to look for data. Content of 532 will be retrieved and stored at $t0***

rt rs

- lw $to, 32 ($s1)
  - 32 is referred to be immediate value. (value being part of instruction is immediately enforced)
  - Assuming $s1 has 500, 500 + 32 = 532 will be the address to look for data. Content of 532 will be retrieved and stored at $t0

I-Type (Immediate).



The lw instruction is identified by 35

Lets also know this…
Why we need load and store?

- C or C P P for an instance support single data elements and complex data elements as well, which include array, structure and union.
- How can a computer represent and access such large structures? (Worthy question right??) From COA perspective?
- There are limited registers only made available in the processor.
  - *Hence, naturally, very limited information / data can be stored in the registers.*
  - *But, a computer may have to handle so much data. Hence, data structures (arrays and structures) are kept in memory.*
- **Arithmetic operations occur only on registers in MIPS instructions;**
- Thus, MIPS must include instructions that transfer data between memory and registers. Such instructions are called **data transfer instructions**.
- To access a word in memory, the instruction must supply the memory **address.**
- The data transfer instruction that copies **data from memory to a register is traditionally called load** and vice-versa is referred as store. Hence, we need load and store.

How much important is I TYPE Instruction (addi)

- Many times a program will use a constant in an operation—for example, **incrementing an index to point to the next element of an array.**
- **An example is precise in this situation:**

Problem Statement: (Condition - Do not use ADDi)

Add constant 5 to Register $S4

How to do this ???

```
LW $t1, Constant_move_tempregister($S2)  # Here, $t1 = 5
ADD $S4, $S4, $t1                        # $S4 = $S4 + $t1
```

Is this not complex? Yes. It is.

So how do we sort this out?

```
ADDI $S4, $S4, 5                         #$S4 = $S4 + 5;
```

Contd.,
An Example Again…

Compiling an Assignment When an Operand Is in Memory

Let's assume that `A` is an array of 100 words and that the compiler has associated the variables `g` and `h` with the registers `$s1` and `$s2` as before. Let's also assume that the starting address, or *base address*, of the array is in `$s3`. Compile this C assignment statement:

```
g = h + A[8];
```

The following instruction can operate on
the value in $t0 (which equals A[8]) since it is in a register. The instruction must add h (contained in $s2) to A[8] ($t0) and put the sum in the register corresponding to g (associated with $s1):

```
add  $s1,$s2,$t0 # g = h + A[8]
```

The constant in a data transfer instruction is called the *offset*, and the register added to form the address is called the *base register*.

Although there is a single operation in this assignment statement, one of the operands is in memory, so we must first transfer A[8] to a register. The address of this array element is the sum of the base of the array A, found in register $s3, plus the number to select element 8. The data should be placed in a temporary register for use in the next instruction.
first compiled instruction is

```
lw   $t0,32($s3) # Temporary reg $t0 gets A[8]
```

8 * 4 = 32

Compiling Using Load and Store

Assume variable h is associated with register $s2 and the base address of the array A is in $s3. What is the MIPS assembly code for the C assignment statement below?

```
A[12] = h + A[8];
```

- Although there is a single operation in the C statement, now two of the operands are in memory, so we need even more MIPS instructions.
- The first two instructions are the same as the prior example, except this time we use the proper offset for byte addressing in the load word instruction to select A[8], and the add instruction places the sum in $t0:

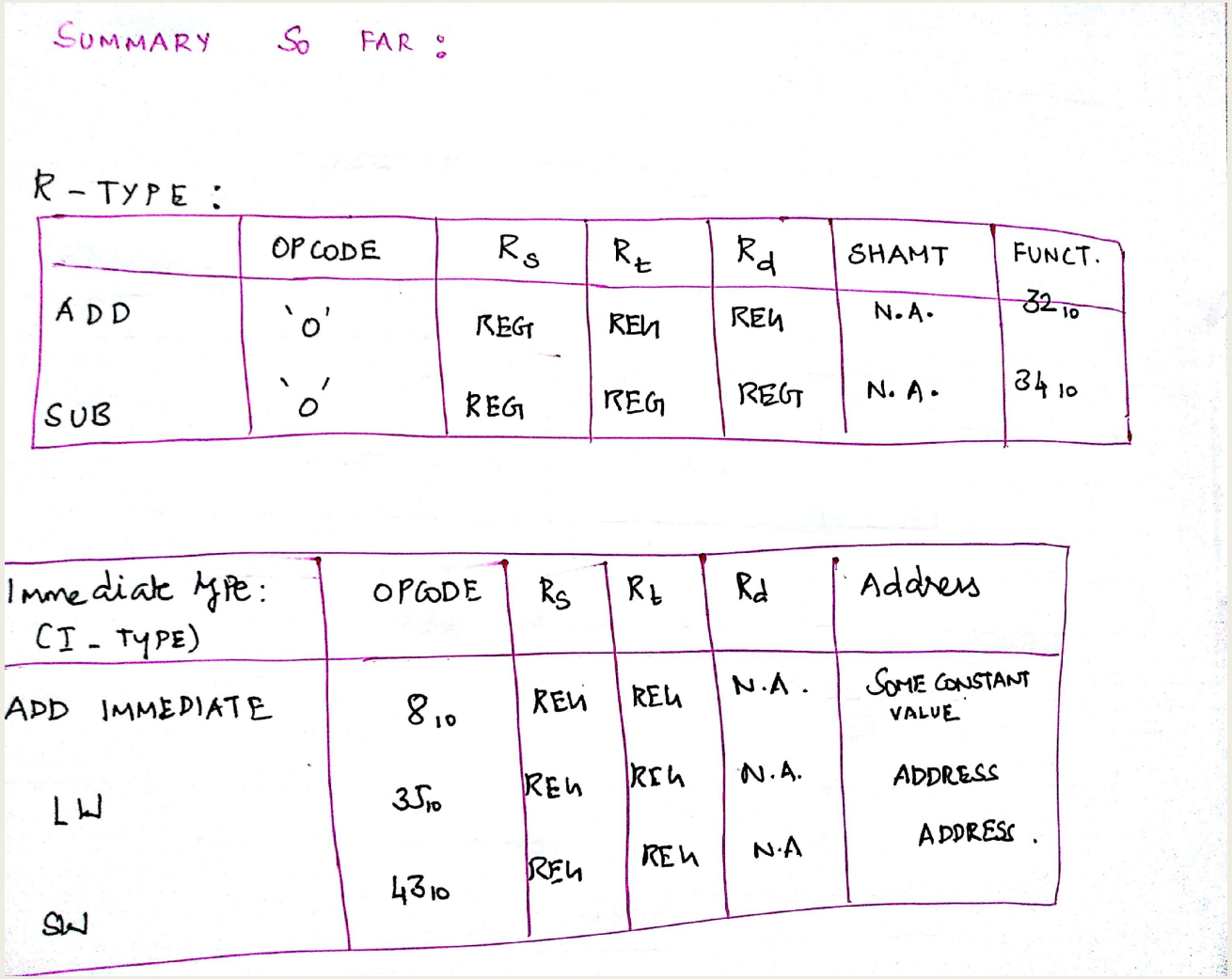
```
lw   $t0,32($s3)   # Temporary reg $t0 gets A[8]
add  $t0,$s2,$t0   # Temporary reg $t0 gets h + A[8]
```

The final instruction stores the sum into A[12], using 48 as the offset and register $s3 as the base register.

```
sw   $t0,48($s3)   # Stores h + A[8] back into A[12]
```

Write the equivalent binary for this.. Work out now.

So far.. So good.


SUMMARY SO FAR :

R-TYPE :

	OPCODE	R_s	R_t	R_d	SHAMT	FUNCT.
ADD	'0'	REG	REG	REG	N.A.	32_{10}
SUB	'0'	REG	REG	REG	N.A.	34_{10}

Immediate Type: (I-TYPE)	OPCODE	R_s	R_t	R_d	Address
ADD IMMEDIATE	8_{10}	REG	REG	N.A.	SOME CONSTANT VALUE
LW	35_{10}	REG	REG	N.A.	ADDRESS
SW	43_{10}	REG	REG	N.A	ADDRESS.

- Add, Sub and Addi are arithmetic.
- LW and SW are Data transfer Instructions.
- For add, sub = R Type = Data in registers.
- For Addi = Constant addition, to save time.
- For Load = Data moved to register from memory
- For Store = Data moved to memory from register.

DESIGN RULES – PART 2

Shriram K Vasudevan

Session 7

Rule : 3
Make the common case fast.

- We can never avoid constants in the instructions.
- Particularly, when it comes to arithmetic instructions, constants remain inevitable.
- Hence, use it as we used with Immediate Instructions as it would certainly be faster than if is getting loaded from memory.

Problem Statement: (Condition - Do not use ADDi)

Add constant 5 to Register $S4

How to do this ???

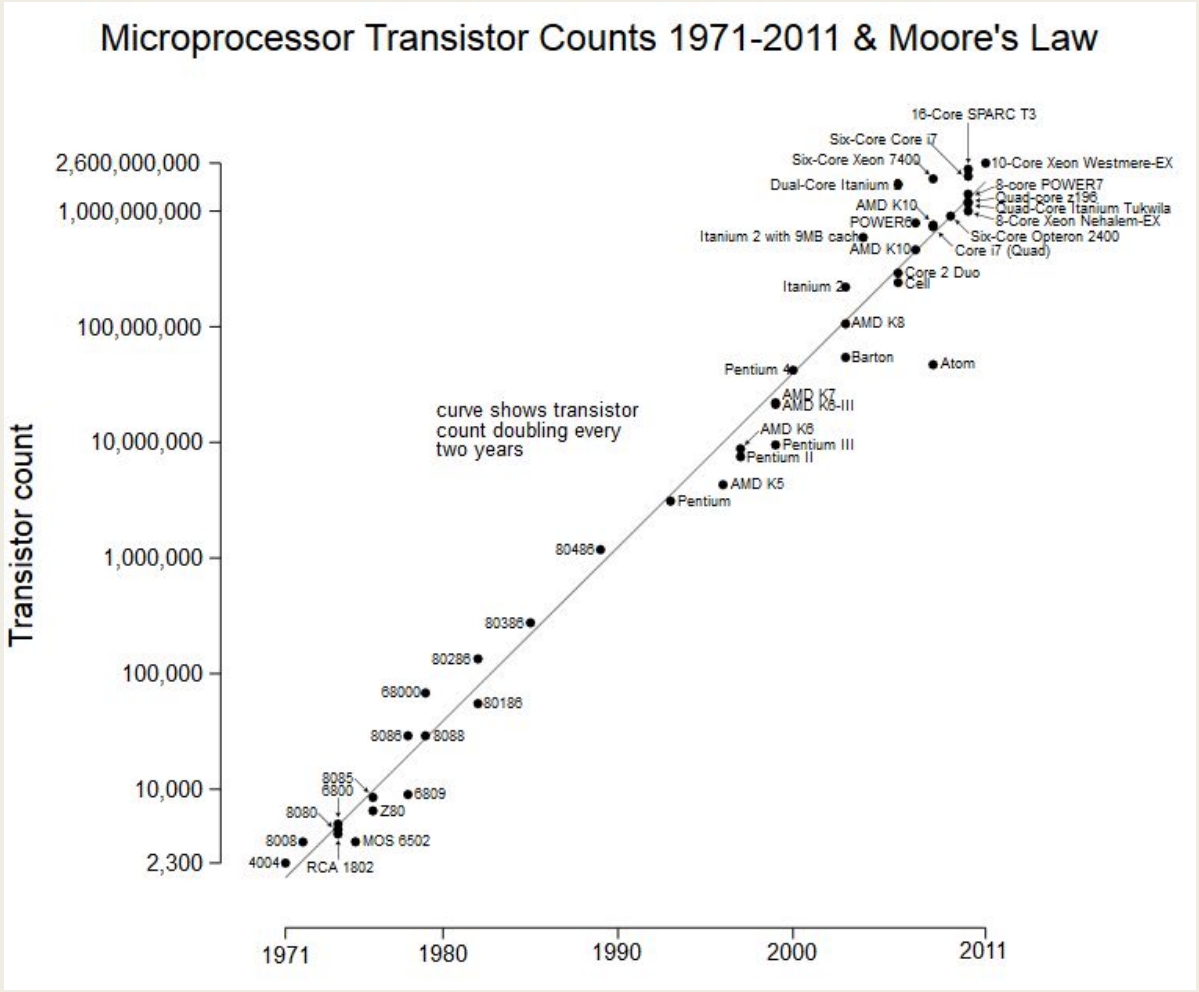
```
LW $t1, Constant_move_tempregister($S2)  # Here, $t1 = 5
ADD $S4, $S4, $t1                        # $S4 = $S4 + $t1
```

Is this not complex? Yes. It is.

So how do we sort this out?

```
ADDI $S4, $S4, 5                         #$S4 = $S4 + 5;
```

Remember this – This is for good!



- The observation made in 1965 by Gordon **Moore**, co-founder of Intel, that the number of transistors per square inch on integrated circuits had doubled every year since the integrated circuit was invented. **Moore** predicted that this trend would continue for the foreseeable future.
- MOORE's law is simply superb! You can't ignore!

Rule : 4
We have to accept some compromises to get good design in place.

- By now, it would have been clearer for the readers that, all MIPS instructions are of same size. I.e. same instruction length. 32 bits.
- This has forced the designers to provide 3 different formats – I J and R type to meet the requirements.
- This is a compromise. Undoubtedly.

- **What is the MIPS machine language code for these three instructions?**

Translating MIPS Assembly Language into Machine Language

We can now take an example all the way from what the programmer writes to what the computer executes. If `$t1` has the base of the array `A` and `$s2` corresponds to `h`, the assignment statement

```
A[300] = h + A[300];
```

is compiled into

```
lw   $t0,1200($t1) # Temporary reg $t0 gets A[300]
add  $t0,$s2,$t0   # Temporary reg $t0 gets h + A[300]
sw   $t0,1200($t1) # Stores h + A[300] back into A[300]
```

The lw instruction is identified by 35

The base register 9 ($t1) is specified in the second field (rs)

Destination register 8 ($t0) is specified in the third field (rt).

The offset to select A[300] is found in the final field (address).

- *let's first represent the machine language instructions using decimal numbers . . .*

The binary equivalent to the decimal form is the following (1200 in base 10 is 0000 0100 1011 0000 base 2):

100011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000 0100 1011 0000		

NOTE: ...2048 1024 512 256 128 64 32 16 8 4 2 1

Summary for you…

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add $s1,$s2,$s3
sub	R	0	18	19	17	0	34	sub $s1,$s2,$s3
addi	I	8	18	17	100			addi $s1,$s2,100
lw	I	35	18	17	100			lw $s1,100($s2)
sw	I	43	18	17	100			sw $s1,100($s2)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer format

Courtesy: Patterson, David A & J L Hennessy, *Computer Organization & Design* 3rd Ed.

LET'S LEARN LOGICAL INSTRUCTIONS

Session – 8

Shriram K Vasudevan

Logical Move!

- Operations supported:
  - *Shift*
    - Shift Left Instruction sll
    - Shift Right Instruction srl
  - *AND (bitwise)*
    - and / andi
  - *OR (bitwise)*
    - or / ori
  - *NOT (bitwise)*
    - nor (no nor immediate)

Let us learn one by one, logically! SLL and SRL:

- First, let us take Shift operations.
- The shift can be done to the left or right based on the instruction.
- Also, the emptied bits shall be filled with 0s just following the shift tradition.
- ***Let us meet a situation to understand the shifting!***
- ***Assume that $S2 has got following content: (32 Bits, don't forget)***

  11_{ten} = 0000 0000 0000 0000 0000 0000 0000 0000 1011_{two}

- Let us do a shift left by 4 with the above input. After shifting, the result what we get would be:

  176 $_{ten}$ = *0000 0000 0000 0000 0000 0000 1011 0000 two*

- Let us also do a shift right by 4 with the above input (176 $_{ten}$)

  176 $_{ten}$ = *0000 0000 0000 0000 0000 0000 1011 0000 two* *after shift right by 4,*

  11_{ten} = 0000 0000 0000 0000 0000 0000 0000 0000 1011_{two}

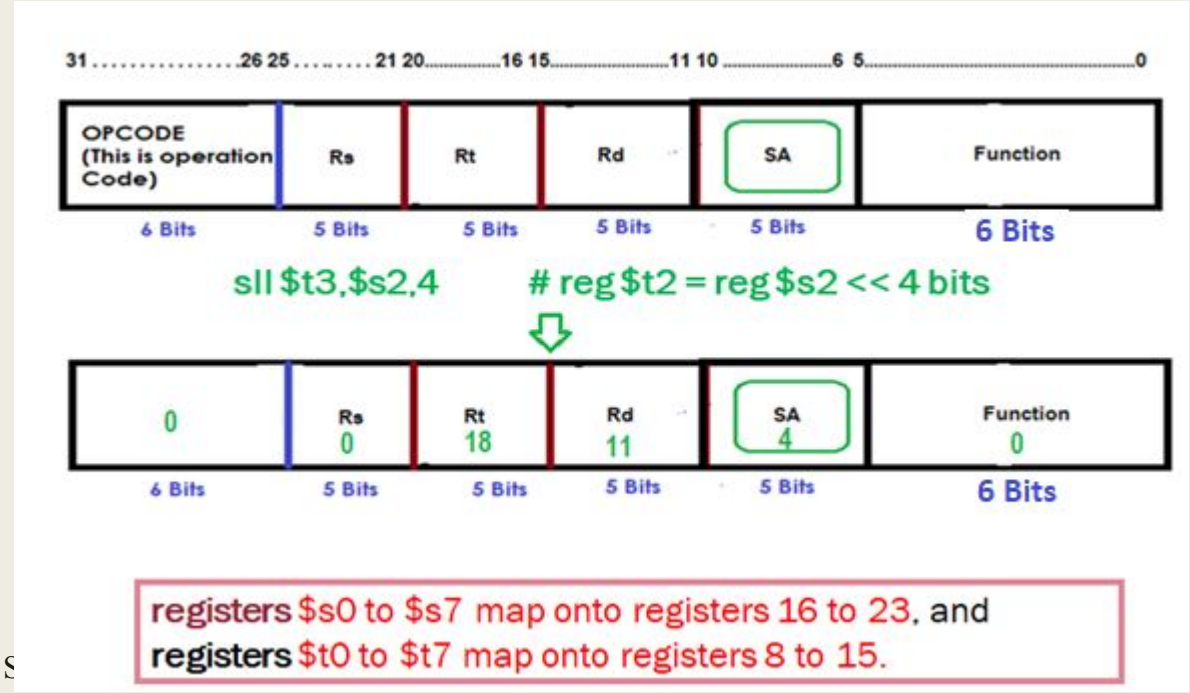
Moral of the story – SLL is complement for SRL

Contd.,

- The instruction to perform the SLL is presented below. The result is expected to be moved to $t3. The content to be shifted is present in $s2.

  sll $t3,$s2,4 # reg $t2 = reg $s2 << 4 bits

- Time has come to use the SHIFT field in the R – Type Instruction. (Remember??, we marked it not applicable, earlier)

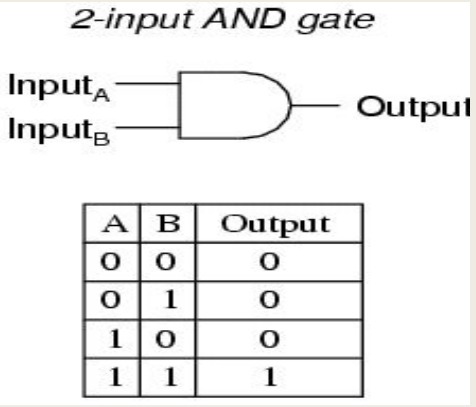


Additional Responsibility of SLL

- Shifting left by *n* bits gives the same result as multiplying by 2^*n*.
- How this is possible? Simple, see below.

Decimal	Binary
1	0 0 0 1
2	0 0 1 0
4	0 1 0 0
8	1 0 0 0

and it is AND!

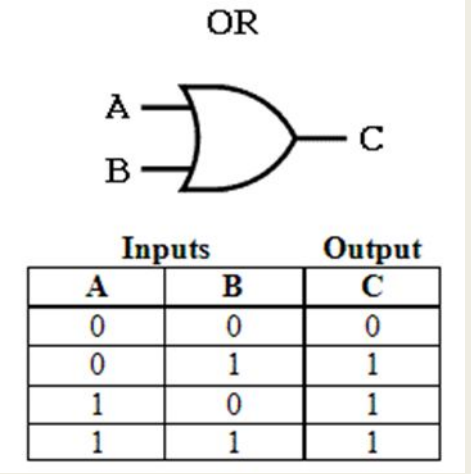


A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

- AND needs both the bits to be 1. You remember AND truth table?
- Let us take a situation:
- Register $t2 has the following content:
  - *0000 0000 0000 0000 0000 1111 0000 0000two*
- Register $t1 has the following content:
  - *0000 0000 0000 0000 1111 1100 0000 0000two*
- We got to AND these two. Result to be stored in $S1
  - *and $s1,$t1,$t2* *# reg $s1 = reg $t1 & reg $t2*
  - *0000 0000 0000 0000 0000 1100 0000 0000two (Hope you understand) . . .*

AND Can be used for BIT MASKING! WHAT WE DID HERE IS "MASKING" and nothing else!

now it is OR, or we quit!

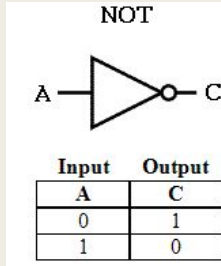


- When there is AND, there is OR as well.
- Again, refer the truth table. If 1 is there at A or B, result is 1.
- Register $t2 has the following content:
  - *0000 0000 0000 0000 0000 0011 0000 0000two*
- Register $t1 has the following content:
  - *0000 0000 0000 0000 0011 1110 0000 0000two*
- We got to OR these two. Result to be stored in $S1
  - *OR $s1,$t1,$t2 # reg $s1 = reg $t1 | reg $t2*
    - 0000 0000 0000 0000 0011 1111 0000 0000two

NOT through NOR!

NOT A logical bit-by-bit operation with one operand that inverts the bits; that is, it replaces every 1 with a 0, and every 0 with a 1.

NOR A logical bit-by-bit operation with two operands that calculates the NOT of the OR of the two operands.



- NOT is inversion operation and as you know, '0' becomes '1' and '1' becomes '0'.
- Let us assume:

– *$t0 has the following content:*

0000 0000 0000 0000 0000 1100 0000 0000two

And Register $t1

has the value "0"

Now,

A NOR 0 = NOT (A OR 0) = NOT (A)

- The instruction is:

nor $S1, $t0, $t1 # here, it is equal to $S1 = ~ ($to | $t1)

1111 1111 1111 1111 1111 0011 1111 1111two (We got the NOT of the INPUT)

Your do it yourself.

- andi
- ori

A Summary!

and	R	0	18	19	17	0	36	and $s1,$s2,$s3
or	R	0	18	19	17	0	37	or $s1,$s2,$s3
nor	R	0	18	19	17	0	39	nor $s1,$s2,$s3
andi	I	12	18	17	100			andi $s1,$s2,100
ori	I	13	18	17	100			ori $s1,$s2,100

R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	address		

DECISION MAKING INSTRUCTIONS

Session – 9

Shriram K Vasudevan

Decision Making

- The best part of a computer (i.e. Processor) is the ability towards making decisions. A calculator may not be that intelligent as a computer, from this particular aspect. N
- When the input data changes dynamically during runtime, there is a certain need to pull out different set of instructions. (if A, Call X. if B, Call Y)
- We are familiar with this in our high level programming languages through if, if else, while, do while etc. History has used goto as an option as well, which slowly has disappeared in the modern programming world.
- MIPS is supporting 2 such instructions BEQ and BNE.

Contd.,

- The first instruction to be learnt is “BEQ” which can be expanded as Branch if Equal.
  - BEQ Reg1, Reg2, Label
- What does this instruction imply? Simple. It says that the value in the reg1 shall be compared with value in reg2. if found equal, the branching shall happen to the label.
- The second instruction is “BNE” which gets expanded as Branch if Not Equal.
  - BNE Reg1, Reg2, Label
- It means go to the statement labeled L1 if the value in register1 does *not* equal the value in register2.
- The mnemonic bne stands for *branch if not equal*.
- These two instructions are traditionally called **conditional branches**
- **Slt is homework. Try it out yourself.**

IF - ELSE to CONDITIONAL BRANCHING MAPPING:

A, B, C, D, E = VARIABLES ASSUMED

$S0 $S1 $S2 $S3 $S4

A - E = $S0 - $S4

*** Remember

"FOR THIS 'C' IF STATEMENT WHAT IS MIPS EQUIVALENT"???

```
IF (D == E)
    A = B+C ;
ELSE
    A = B-C;
```

To CHECK:

* IF D == E

```
BNE   $S3, $S4, LABEL1    # IF D# E, LABEL1
ADD   $S0, $S1, $2        # IF D=E, THIS WORKS
J     END                 # WORK OVER

LABEL1: SUB  $S0, $S1, $S2    # SUB, STORE RESULT IN $S0, 'A'
END:
```

So,

```
BEQ   $S1, $S2, LABEL
        IF $S1 == $S2, GO TO LABEL
BNE   $S1, $S2, LABEL
        IF $S1 ≠ $S2, GO TO LABEL
```

BEQ, BNE — I TYPE INSTRUCTION.

	TYPE	OP	REG	REG	Add
BEQ	I	4	Rs	Rt	Address
BNE	I	5	Rs	Rt	Address

BEQ $S1, $S2, 150 ← "SEE THIS"

BEQ = 4

4	17	18	40
OP	Rs	Rt	Add

40 X 4 = 160

While – let’s learn it for a while.

- Why important?
  - *Iterative operations are carried out with Loops (after checking for conditions).*
  - *Recollect what do we do with traditional while loop in the C Programming.*
  - *An instance shall be handy.*
  - *Let us write code for while in MIPS with having C code as reference.*

```
While (save [P] = Q)
    P + = 1 ;
```

"CONDITION TO BE CHECKED"

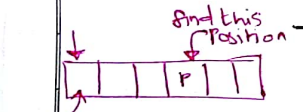
Assume :-

```
P = $S1 ;    Q = $S2
base of array save = $S3
```

Let's write equivalent MIPS :

sequence :

- Save [P] to be moved to some temp. register. [Let us take $to]
- for this above step to be done, one needs the address. [Makes sense huh!!]
- index P * '4' to be done
  - to handle addressing scheme in 'MIPS'
- How do you multiply by 4??
  - SLL by '2'



CODE WITH MIPS :

```
LOOP_FOREVER: SLL  $to, $S1, 2   # $to = 4*P
```

To GET ADDRESS OF SAVE [P], add $to & base of SAVE. [$S3]

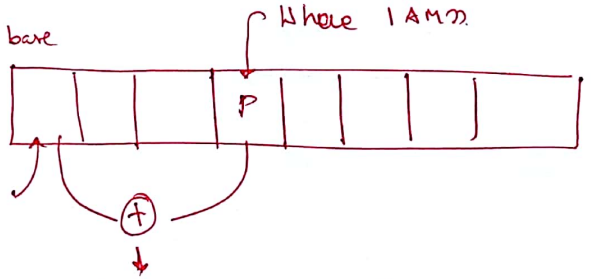
```
ADD   $to, $to, $S3  # ADDRESS FOUND
LW    $t1, 0($to)    # LOAD THAT TO $t1
```

NOW,

WE HAVE CONTENT. LETS DO [SAVE[P] == Q] CHECK.

```
BNE   $t1, $S2, END.   # If not equal, Exit.
ADD   $S3, $S3, 1
J     LOOP_FOREVER     # Loop.
END:
```

Note :



Let us know SWITCH - CASE

- You know what is switch-case. It provides comfort for the programmer to select one of alternatives.
- Many languages including scripting languages support switch-case.
- We do not have switch case supported directly through instructions in MIPS.
- But, we have other ideas.
  - *Implement switch case through if-then-else sequence.*
  - *Try this out yourself! A simple home work, folks!*

A final summary!

MIPS machine language

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add $s1,$s2,$s3
sub	R	0	18	19	17	0	34	sub $s1,$s2,$s3
lw	I	35	18	17	100			lw $s1,100($s2)
sw	I	43	18	17	100			sw $s1,100($s2)
and	R	0	18	19	17	0	36	and $s1,$s2,$s3
or	R	0	18	19	17	0	37	or $s1,$s2,$s3
nor	R	0	18	19	17	0	39	nor $s1,$s2,$s3
andi	I	12	18	17	100			andi $s1,$s2,100
ori	I	13	18	17	100			ori $s1,$s2,100
sll	R	0	0	18	17	10	0	sll $s1,$s2,10
srl	R	0	0	18	17	10	2	srl $s1,$s2,10
beq	I	4	17	18	25			beq $s1,$s2,100
bne	I	5	17	18	25			bne $s1,$s2,100
slt	R	0	18	19	17	0	42	slt $s1,$s2,$s3
j	J	2	2500					j 10000 (see Section 2.9)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer, branch format

Contd.,

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three operands; data in registers
	subtract	sub $s1,$s2,$s3	$s1 = $s2 – $s3	Three operands; data in registers
Data transfer	load word	lw $s1,100($s2)	$s1 = Memory[$s2 + 100]	Data from memory to register
	store word	sw $s1,100($s2)	Memory[$s2 + 100] = $s1	Data from register to memory
Logical	and	and $s1,$s2,$s3	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	or $s1,$s2,$s3	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	nor $s1,$s2,$s3	$s1 = ~ ($s2 |$s3)	Three reg. operands; bit-by-bit NOR
	and immediate	andi $s1,$s2,100	$s1 = $s2 & 100	Bit-by-bit AND reg with constant
	or immediate	ori $s1,$s2,100	$s1 = $s2 | 100	Bit-by-bit OR reg with constant
	shift left logical	sll $s1,$s2,10	$s1 = $s2 << 10	Shift left by constant
	shift right logical	srl $$s1,$s2,10	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	beq $s1,$s2,L	if ($s1 == $s2) go to L	Equal test and branch
	branch on not equal	bne $s1,$s2,L	if ($s1 != $s2) go to L	Not equal test and branch
	set on less than	slt $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; used with beq, bne
	set on less than immediate	slt $s1,$s2,100	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare less than immediate; used with beq, bne
Unconditional jump	jump	j L	go to L	Jump to target address

MIPS architecture revealed

THANKS

Shriram K Vasudevan
YouTube:
https://www.youtube.com/playlist?list=PL3uLubnzL2TnsFH20zsKOcaViV0Jl_b3h