الجامعة اللبنانية الأميركية
Lebanese American University

School of Arts and Sciences

- **Course name: Computer Organization**
- **Course number: CSC320**
- **Section 11**
- **Instructor: Dr. Sanaa Sharafeddine Dawy**
- **Date: 30/03/2022**

Design Phase 1 - CO

Presented by:

Hussien Ali-Ahmad	**ID: 202104969**
Bilal Delbani	**ID: 202104998**
Mohammad Al Fallah	**ID: 202105098**
Mohammad Alizzi	**ID: 202104792**

__Project Description__

In this project, you are expected to design and simulate a simple RISC processor that supports the following 28 instructions:

●- Arithmetic instructions support addition, subtraction, multiplication, division, AND, OR, and NOT. All arithmetic instructions take three registers: two source registers and one destination register. In the case of NOT, one of the source registers is not used.

●- Arithmetic Immediate Instructions.

●- Memory access instructions support loading and storing words from main memory.

●- Branch instructions support branch if equal, branch if not equal, branch if less than and branch if greater than

●- Register operation instructions support initializing a given register to a constant value, shifting the value of the register to the left by specific number of bits, and shifting the value of the register to the right by specific number of bits.

The processor has 8 general purpose 32-bit registers: R0, R1, R2, … R63, in addition to the program counter (PC) register. Two memories are used one for program memory, and other for data memory.

__Questions:__

1. Research RISC and CISC processors and discuss their main properties and differences between them.
2. Determine the types of instructions supported by your processor. For each type, design the instruction structure by specifying the instruction fields and their sizes (i.e. number of bits required for each field).
3. For each given instruction, define its syntax and corresponding opcode.
4. Specify unique identifiers for each of the registers.
5. For each type of instruction, give an example instruction and convert it to its corresponding machine language.
6. Specify the increment performed to the program counter.
7. Prepare your own green sheet like the MIPS green sheet to summarize your design.

1.

RISC and CISC:

- **RISC (Reduced Instruction Set Architecture):**
  The main purpose from this is to simplify the hardware by using a basic and few steps for loading, evaluating, and storing operations. Like loading data by load command.

- **CISC (Complex Instruction Set Architecture):**
  The main purpose from this is to load, evaluate, and store operations by a single instruction.
  Like loading, evaluating, and storing data by the multiplication command (complex instruction).

- **Both tend to increase the CPU performance. The RISC, at the cost of numbers of instructions in each program, reduces the cycles per instruction. The CISC, at the cost of increasing the number of cycles per instruction, reduces the number of instructions in each program**

RISC	**CISC**
Software focus	Hardware focus
Only Using Hardwired Control Unit	Using both hardwired and microprogrammed control unit
Using Transistors for more registers	Using Transistors for storing complex instructions
Fixed Sized Instructions	Dynamic Sized Instructions
Only Performing Arithmetic Operations of Register to Register	Performs Register to Register, Register to Memory, or ever Memory to Memory
Uses a greater number of registers	Uses a smaller number of registers
Large code	Small code
Instruction is executed in single clock cycle	Instruction takes longer than one clock cycle
Instruction can fit in one word	Instructions are larger than one word
For example: Subtracting two numbers will be performed by writing the programmer the first load command then using the correct operator and storing the result in wanted location.	**For example:** Subtracting two numbers will be performed by a single command like SUB.

2.

Deciding the format:

In all cases the processor must fetch the most significant bit of the opcode, if this bit is 0 then the instruction corresponds to the B – Format. Otherwise, if that bit is 1 then it corresponds to the M – Format.

For example:

If the opcode was '01011' then the instruction will be B – Format.

If the opcode was '11010' then the instruction will be M – Format.

There are two types of Formats:

1. B – Format
2. M – Format

B – Format:

31 26 20 14 0

Opcode	Var1	Var2	Constant/Address
5	6	6	15

The B – Format has 4 fields:

1. The first field corresponds to the opcode (5 bits).
2. The second field corresponds to the "Var1" register (6 bits).
3. The third field corresponds to the "Var2" register (6 bits).
4. The fourth field corresponds to the constant or the address (15 bits).

M – Format:

31 26 20 14 8 4 0

Opcode	Var1	Var2	Dest	Shamt	Unused
5	6	6	6	4	5

The M – Format has 6 fields:

1. The first field corresponds to the opcode (5 bits).
2. The second field corresponds to the "Dest" register (6 bits).
3. The third field correspond to the "Var1" register (6 bits).
4. The fourth field correspond to the "Var2" register (6 bits).
5. The fifth field corresponds to the Shamt (4 bits).
6. The sixth field corresponds to the Unused are added to this format so that all instructions have the same size of 32 – bits.

3.

Syntax and opcode for each Instruction:

M-format:

- **Add**: ++ Dest,var1,var2 // adds var1 and var2 then places the result in Dest

  The opcode: 10000

- **Subtract**:-- Dest,var1,var2 // subtracts var1 and var2 then places the result in Dest

  The opcode: 10001

- **Multiplication** : * Dest,var1,var2 // multiplies var1 and var2 then places the result in Dest

  The opcode: 10100

- **Division** / Dest,var1,var2 // divides var1 and var2 then places the result in Dest

  The opcode: 10101

- **And** : & Dest,var1,var2 // it ANDS var1 and var2 then places the result in Dest

  The opcode: 10110

- **OR**: || Dest,var1,var2 // computes (var1 or var2) and then places the result in Dest

  The opcode: 10111

- **Not**: ~ Dest, var1, unused // does the NOT of var1 and then places the result in Dest

  The opcode: 11000

- **Load Word**: LW Dest, unused, var2 //loads into Dest the content from the memory of the address that var2 carries.

  The opcode: 10010

- **Store Word**: SW Dest, unused, var2 // stores the content of Dest into the the memory at the address that var2 carries. .

  The opcode: 10011

- **Set Less Than** : SLT dest,var1,var2 // it checks if var1<var2 and sets Dest to the value 1 Otherwise it sets Dest to 0.
  The opcode: 11001

- **Shift Left Logical**: << Dest, var1, i // shifts the bits var1 to the left with i bits(the value of var1 is multiplied by 2^i)
  opcode: 11010

- **Shift Right Logical**: >> Dest, var1, i // shifts the bits of var1 to the right with i bits(the value of var1 is divided by 2^i)
  The opcode: 11011

- **Branch On Equal** : B= Dest,var1, var2, // if Dest =var1 is true then it branches to the specified address in var2.

  The opcode: 11100

- **Branch On Not Equal** B!= Dest,var1, var2, // if Dest !=var1 then it branches to specified address in var2
  The opcode: 11101

- **Branch less than** B< Dest,var1, var2, // if Dest <var1 then it branches to specified address in var2

  The opcode: 11110

- **Branch greater than** : B> Dest,var1, var2, // if Dest >var1 then it branches to specified address in var2

  The opcode: 11111

B-format:

- **Assign Immediate**: ASi var2, constant // it initializes the destination register to the value of the constant
  The opcode: 00000

- **Add Immediate**: i++ var2, var1, constant // adds value in var1 and a constant and places the result in var2

  The opcode: 00001

- **Subtract Immediate**: i-- var2, var1, constant // subtracts value in var1 and a constant and stores the result in var2

  The opcode: 01100

- **And Immediate**: i& var2, var1, constant // Does the AND of var1 and the constant and stores the result in var2

  The opcode: 00010

- **Or Immediate**: i|| var2, var1, constant // Does the Or of var1 and the constant and places the result in var2

  The opcode: 00011

- **Not Immediate**: ***i~*** var2, unused, constant // Does the Not of the constant and places the result in var2

  The opcode: 00100

- **Load Address:** **LA** var2 ,var1,constant /calculates the relative address (that is var1+constant) and then stores the result into the register var2
  The opcode: 01101

- **Division Immediate**: i/ var2, var1, constant // Does the division of var1 and the constant and places the result in var2

  The opcode: 00101

- **Multiplication Immediate**: i* var2, var1, constant // Does the multiplication of var1 and the constant and places the result in var2

  The opcode: 01011

4.

Registers:

The 64 different registers can be represented using 6 bits only since 6 bits can represent numbers ranging from 0 till 63 which is 111111.

Registers	Number	Use
!Z	0	The constant value 0
!S0 - !S20	1 -21	Saved Registers
!T0 - !T20	22 - 42	Temporary Registers
!A0 - !A16	43 - 59	Address Registers
!GP	60	Global Pointer
!SP	61	Stack Pointer
!FP	62	Frame Pointer
!RA	63	Register Address

5.

Examples: Here in our examples, we are going to convert a assembly representation into binary language using the B – Format and M - Format.

I. From assembly to binary representation: B – Format:

- ***Add Immediate instruction:***

In assembly: ***i++ !S0, !S0, 9***

i++	!S0	!S0	9
opcode	Var1	Var2	Constant
00001	000001	000001	000000000001001

In binary: 00001000001000001000000000001001

- ***And Immediate instruction*:**

In assembly: ***i& !S2, !S3, 16***

i&	**!S2**	**!S3**	**16**
opcode	**Var1**	**Var2**	**Constant**
00010	000011	000100	0000000000010000

In binary: 0001000000110001000000000000001000

- ***Or Immediate instruction*:**

In assembly: ***i|| !S4, !S3, 20***

i||	**!S4**	**!S3**	**20**
opcode	**Var1**	**Var2**	**Constant**
00011	000101	000100	0000000000010100

In binary: 000110001010001000000000000010100

- ***Not Immediate instruction*:**

In assembly: ***i~ !S5, 7***

i~		**!S5**	**7**
opcode	**Var1**	**Var2**	**Constant**
00100	000000	000110	0000000000000111

In binary: 001000000000000110000000000000111

- ***Division Immediate instruction:***

In assembly: ***i/ !S6, !S2, 17***

i/	**!S6**	**!S2**	**17**
opcode	**Var1**	**Var2**	**Constant**
00101	000111	000011	000000000010001

In binary: 00101000111000011000000000001000

- ***Multiplication Immediate instruction:***

In assembly: ***i* !T20, !S2, 30***

i*	**!T20**	**!S2**	**30**
opcode	**Var1**	**Var2**	**Constant**
01011	101010	000011	000000000011110

In binary: 01011101010010011000000000011110

- ***Assign Immediate instruction:***

In assembly***: ASi !S5,8***

ASi		**!S5,**	**8**
Opcode	**Var1**	**Var2**	**Constant/Address**
00000	000000	000110	000000000001000

In binary: 00000000000000110000000000001000

II. <u>From assembly to binary representation: M – Format:</u>

- ***<u>Add instruction:</u>***

<u>In assembly:</u> ++ ***!T1,!S0,!S1***

++	**!T1,**	**!S0,**	**!S1**		
Opcode	**Dest**	**Var1**	**Var2**	**Shamt**	**unused**
10000	010111	000001	000010	0000	00000

<u>In binary:</u> 10000010111000001000010000000000

- ***<u>Subtract instruction:</u>***

<u>In assembly:</u> -- ***!S6,!S0,!S1***

--	**!S6,**	**!S0,**	**!S1**		
Opcode	**Dest**	**Var1**	**Var2**	**Shamt**	**unused**
10001	000111	000001	000010	0000	00000

<u>In binary:</u> 10001000111000001000010000000000

- ***<u>And instruction:</u>***

<u>In assembly:</u> ***& !S3,!S0,!Z***

&	**!S3,**	**!S0,**	**!Z**		
Opcode	**Dest**	**Var1**	**Var2**	**Shamt**	**unused**
10110	000100	000001	000000	0000	00000

<u>In binary:</u> 10110000100000001000000000000000

- ***Or instruction:***

In assembly: **|| *!S10,!S0,!S1***

||	**!S10,**	**!S0,**	**!S1**		
Opcode	**Dest**	**Var1**	**Var2**	**Shamt**	**unused**
10111	001011	000001	000010	0000	00000

In binary: 10111001011000001000010000000000

- ***Multiplication instruction:***

In assembly: *** *!S20,!S11,!S12***

*****	**!S20,**	**!S11,**	**!S12**		
Opcode	**Dest**	**Var1**	**Var2**	**Shamt**	**unused**
10100	010101	001100	001101	0000	00000

In binary: 10100010101001100001101000000000

- ***Not instruction:***

In assembly: **~ *!T1,!S0***

~	**!T1,**		**!S0**		
Opcode	**Dest**	**Var1**	**Var2**	**Shamt**	**unused**
11000	010111	000000	000001	0000	00000

In binary: 11000010111000000000001000000000

- ***Load instruction:***

In assembly: ***LW !S2,!A1***

LW	**!S2,**		**!A1**		
Opcode	**Dest**	**Var1**	**Var2**	**Shamt**	**unused**
10010	000011	000000	000100	0000	00000

In binary: 10010000011000000000100000000000

- ***Branch Less Than instruction:***

In assembly: ***B< !S2, !S1, !A2***

B<	**!S2**	**!S1**	**!A2**		
opcode	**Dest**	**Var1**	**Var2**	**shamt**	**unused**
11110	000011	000010	101101	00000	00000

In binary: 111100000110000101011010000000000

#**Note**: !A2 contains the Target address = PC+ (offset * 4) => for example:
1000: B< !S2, !S1, !A2
!A2 is calculated before as the exact address for the Branch Instruction

1004: Instruction 1 …
1008: Instruction 2 …
1012: Instruction 3 …
1016: Instruction 4: ….
1020: branch ….

Target address = 1020
Target address=1020=address of branch

6.
PC:
The program counter must move to a new instruction every time it is incremented. Since this assembly language's instructions are composed of 32 bits each, the Program counter must be incremented by 4 Bytes (32 bits).

7.MIPS BLUE SHEET

Opcode(bin)	Opcode(hex)	Format	Function	Name
00000	00_{hex}	B	ASi	Assign Immediate
00001	01_{hex}	B	i++	Add Immediate
00010	02_{hex}	B	i&	And Immediate
00011	03_{hex}	B	i||	Or Immediate
00100	04_{hex}	B	i~	Not Immediate
00101	05_{hex}	B	i/	Division Immediate
01011	$0B_{hex}$	B	i*	Multiplication Immediate
01100	$0C_{hex}$	B	i--	Subtract Immediate
10000	10_{hex}	M	++	Add
10001	11_{hex}	M	--	Subtract
10010	12_{hex}	M	LW	Load word
10011	13_{hex}	M	SW	Store word
10100	14_{hex}	M	*	Multiplication
10101	15_{hex}	M	/	Division
10110	16_{hex}	M	&	And
10111	17_{hex}	M	||	Or
11000	18_{hex}	M	~	Not
11010	$1A_{hex}$	M	<<	Shift Left Logical
11011	$1B_{hex}$	M	>>	Shift Right Logical
11100	$1C_{hex}$	M	B=	Branch on equal
11101	$1D_{hex}$	M	B!=	Branch on not equal
11110	$1E_{hex}$	M	B<	Branch less than
11111	$1F_{hex}$	M	B>	Branch greater than

Registers	Number	Use
!Z	0	The constant value 0
!S0 - !S20	1 -21	Saved Registers
!T0 - !T20	22 - 42	Temporary Registers
!A0 - !A16	43 - 59	Address Registers
!GP	60	Global Pointer
!SP	61	Stack Pointer
!FP	62	Frame Pointer
!RA	63	Register Address

B – Format:

M – Format:

31 26 20 14 0

Opcode 26	20 Var1	Var2	Constant 4 Address
5	6	6	15

31 14 8 0

Opcode	Dest	Var1	Var2	Shamt	Unused
5	6	6	6	5	5