Practical 1

- **Aim :** Implement Booth’s algorithm

Booth’s multiplication algorithm:

Booth’s algorithm is a multiplication algorithm that multiplies two signed binary numbers in 2nd’s compliment notation.

Algorithm:

Put multiplicand in BR and multiplier in QR, and then the algorithm works as per the following conditions:

- 1: If Qn and Qn+1 are same i.e. 00 or 11 perform arithmetic shift by 1 bit.
- 2: If QnQn+1 = 10; do A = A+ BR and perform arithmetic shift by 1 bit.
- 3: If QnQn+1 = 01; do A = A – BR and perform arithmetic shift by 1 bit.

Example:-

Input : 0110,0010

Output:

Qn Qn+1		AC	QR	step count (SC)
	Initial.	0000	0010	4
0 0.	Right shift.	0000	0001	3
1 0	A = A – BR	1010		
	Right shift.	1101	0000	2
0 1.	A = A + BR	0011		
	Right shift.	0001	1000	1
0 0	right shift.	0000	1100.	0

Result = 1100.

// Java code to implement Booth’s algorith

Practical 2

- **Aim :** Write the working of 8085 simulator GNUsim8085 and basic architecture of 8085 along with small introduction.

❖ **Introduction:**

- The GNUsim8085 is an 8 bit micro processor and it is the updated version of the micro processor (with the help of NMOS technology).
- The configuration of 8085 micro processor mainly include data bus 8-bit , address bus 16- bit, program counter 16-bit, stack pointer 16-bit, registers 8-bit, +5v voltage supply and works at 3.2MHz single segment CLK.
- The application of 8085 micro processor include microwave ovens, washing Machines, gadgets, etc.

❖ **Architecture:**

The architecture of 8085 micro processor mainly includes the timing and control unit, Arithmetic and logic unit, decoder, instruction register, interrupt control,a register array, serial input/output control.

The most important part of the micro processor is the central processing unit.

Operations of 8085 Microprocessor:

The main operation operation of ALU is arithmetic as well as logical which includes addition, increment, subtraction, decrement, logical operations like OR, AND, EX-OR , complement, evaluation, left shift or right shift.

Both the temporary registers as well as accumulators are utilised for hiding the information throughout in the operations then the outcome will be stored within the accumulator.

The different flags are arranged or rearrange based on the outcome of the operation.

- **Flag registers :**

The flag registers of micro processor 8085 are classified into five types namely sign, zero, auxiliary carry, parity & carry. The positions of bit set aside for these types of flags.

After the operation of an ALU, when the result of the most significant bit (D7) is one, then the sign flag will be arranged.

When the operation of the ALU outcome is zero, then the zero flags will be set.

- **Control and timing unit:**

The control and timing unit co-ordinates with all the actions of the micro processor by the clock & gives the control signals which are required for communication among the micro processor as well as peripherals.

- **Decoder and instruction register:**

As an order is obtained from memory after that it is located in the instruction register and encoded and decoded into different device Cycles.

- **Register array:**

The general purpose programmable registers are classified into several types apart from the accumulator such as B,C,D,E,H and L. These type utilised as 8 bit registers otherwise coupled to stock up the 16-bit of data.

The permitted couples are used in the processors and it cannot be utilised with the developer.

The permitted couples are BC,DE and HL, and the short term W&Z registers are used in the processor and it cannot be utilised with the developer.

- **Special purpose registers :**

These registers are classified into four types namely program counter, stack pointer, increment or decrement register, address buffer or data buffer.

1. **Program counter**

   This is the first type of special purpose register and considers that the instruction is being performed by the micro processor.
   When the ALU completed performing the instruction, then the micro processor searches for other instruction to be performed.
   Thus, there will be a requirement of holding the next instruction address to be performed in order to conserve time.
   Micro processor increases the program when an instruction is being performed, therefore that the program counter position to the next instruction memory address is going to be performed.
2. **Stack pointer In 8085**
   The stack pointer is a 16-bit register and functions similar to a stack, which is constantly increased or decreased with two throughout push and pop processes.
3. **Increment or decrement register**

   The 8-bit register contents or else a memory position can be increased or decreased with one.

   The 16-bit register it useful for incrementing or decrementing program counters as well as stack pointer register content with one.

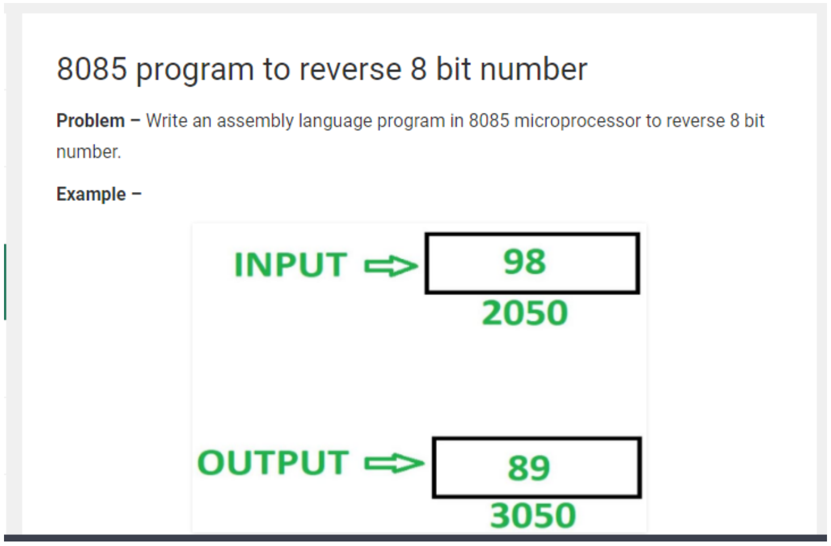
   This operation can be performed on any memory position or any kind of register.
4. **Address buffer & address data buffer**

   Address buffer stores the copied information from the memory for the execution.

   The memory and I/O chips are associated with these buses; then the CPU can replace the preferred data by I/O chips and the memory.

Practical 3

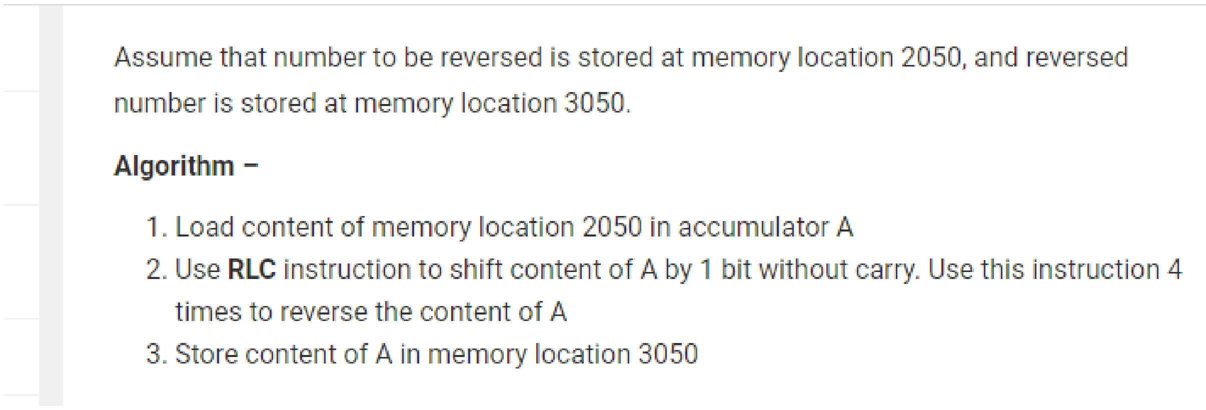
- **Aim :** write an assembly language code in GNUsim8085 to store numbers in reverse order in memory location.



8085 program to reverse 8 bit number

Problem – Write an assembly language program in 8085 microprocessor to reverse 8 bit number.

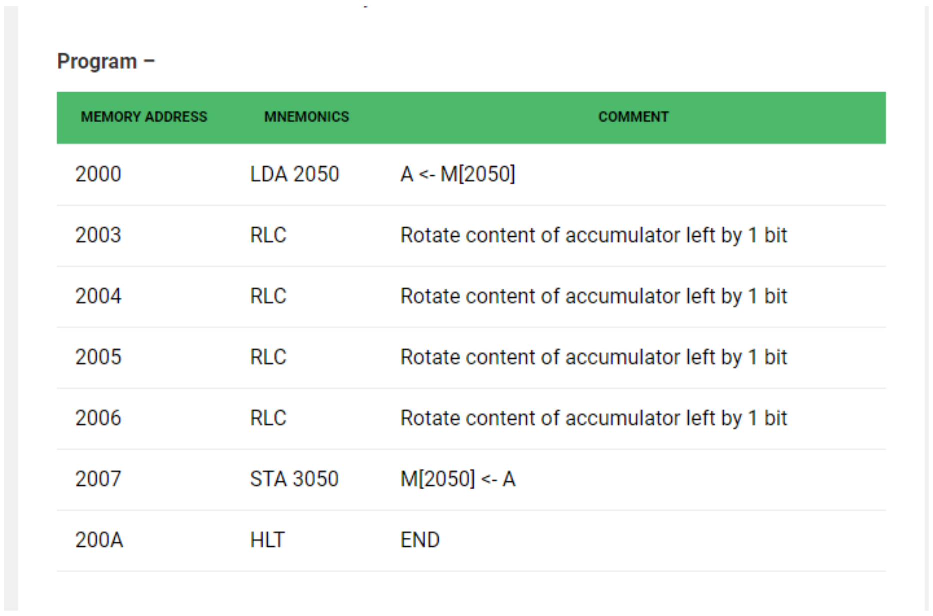
Example –



Assume that number to be reversed is stored at memory location 2050, and reversed number is stored at memory location 3050.

Algorithm –

1. Load content of memory location 2050 in accumulator A
2. Use **RLC** instruction to shift content of A by 1 bit without carry. Use this instruction 4 times to reverse the content of A
3. Store content of A in memory location 3050



Program –

MEMORY ADDRESS	MNEMONICS	COMMENT
2000	LDA 2050	A <- M[2050]
2003	RLC	Rotate content of accumulator left by 1 bit
2004	RLC	Rotate content of accumulator left by 1 bit
2005	RLC	Rotate content of accumulator left by 1 bit
2006	RLC	Rotate content of accumulator left by 1 bit
2007	STA 3050	M[2050] <- A
200A	HLT	END

-

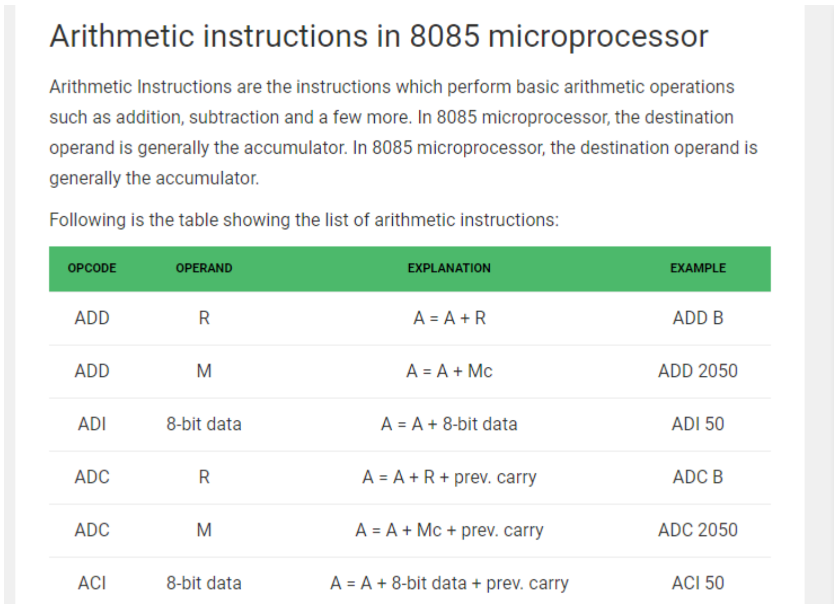
Explanation:
Register A used.

1. LDA 2050: load value of memory location 2050 in accumulator A.
2. RLC : shift content of accumulator left by 1 bit without carry.
3. RLC : shift content of accumulator left by 1 bit without carry.
4. RLC : shift content of accumulator left by 1 bit without carry.
5. RLC : shift content of accumulator left by 1 bit without carry.
6. STA 3050: store content of accumulator left by 1 bit without carry.
7. HLT: Stops executing the program and halts any further execution.

Practical 4

- **Aim:** Write an assembly language code in GNUSIM8085 to implement arithmetic instruction.

Arithmetic instructions are the instructions which perform basic arithmetic operations such as addition, subtraction and a few more. In 8085 microprocessor, the destination operand is generally the accumulator.

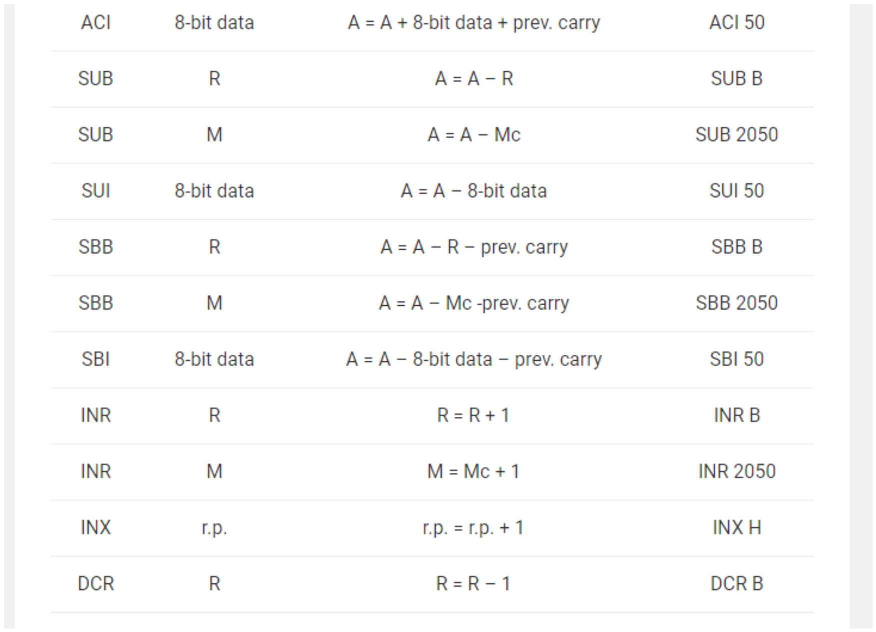


Arithmetic instructions in 8085 microprocessor

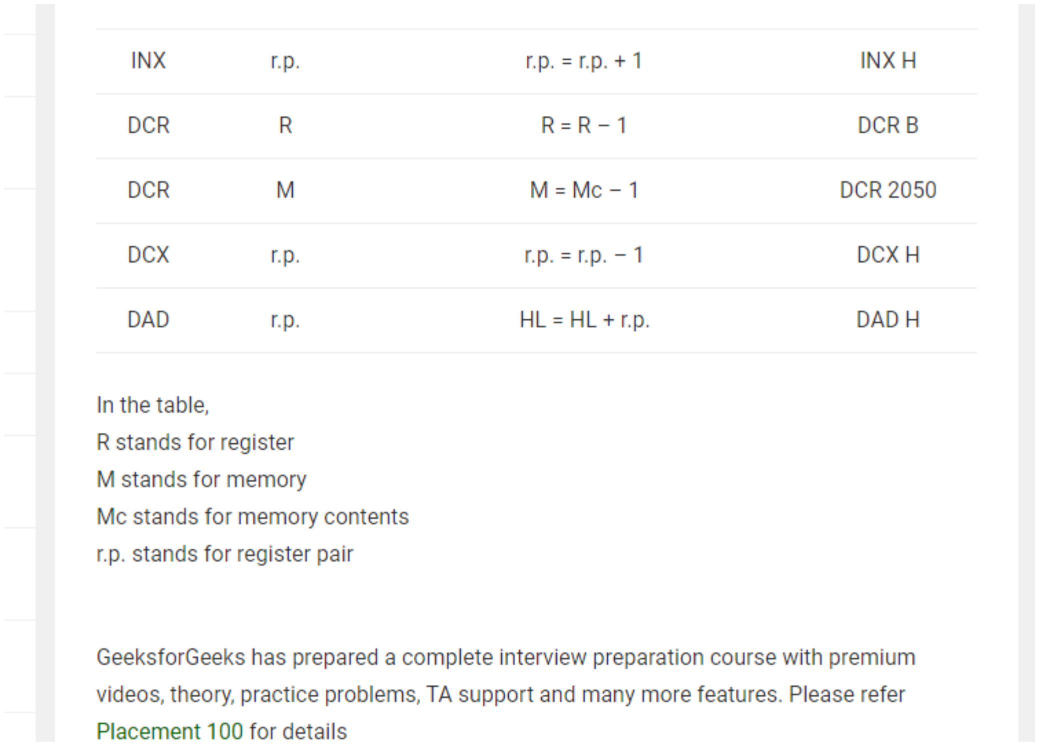
Arithmetic Instructions are the instructions which perform basic arithmetic operations such as addition, subtraction and a few more. In 8085 microprocessor, the destination operand is generally the accumulator. In 8085 microprocessor, the destination operand is generally the accumulator.

Following is the table showing the list of arithmetic instructions:

OPCODE	OPERAND	EXPLANATION	EXAMPLE
ADD	R	A = A + R	ADD B
ADD	M	A = A + Mc	ADD 2050
ADI	8-bit data	A = A + 8-bit data	ADI 50
ADC	R	A = A + R + prev. carry	ADC B
ADC	M	A = A + Mc + prev. carry	ADC 2050
ACI	8-bit data	A = A + 8-bit data + prev. carry	ACI 50



ACI	8-bit data	A = A + 8-bit data + prev. carry	ACI 50
SUB	R	A = A – R	SUB B
SUB	M	A = A – Mc	SUB 2050
SUI	8-bit data	A = A – 8-bit data	SUI 50
SBB	R	A = A – R – prev. carry	SBB B
SBB	M	A = A – Mc -prev. carry	SBB 2050
SBI	8-bit data	A = A – 8-bit data – prev. carry	SBI 50
INR	R	R = R + 1	INR B
INR	M	M = Mc + 1	INR 2050
INX	r.p.	r.p. = r.p. + 1	INX H
DCR	R	R = R – 1	DCR B



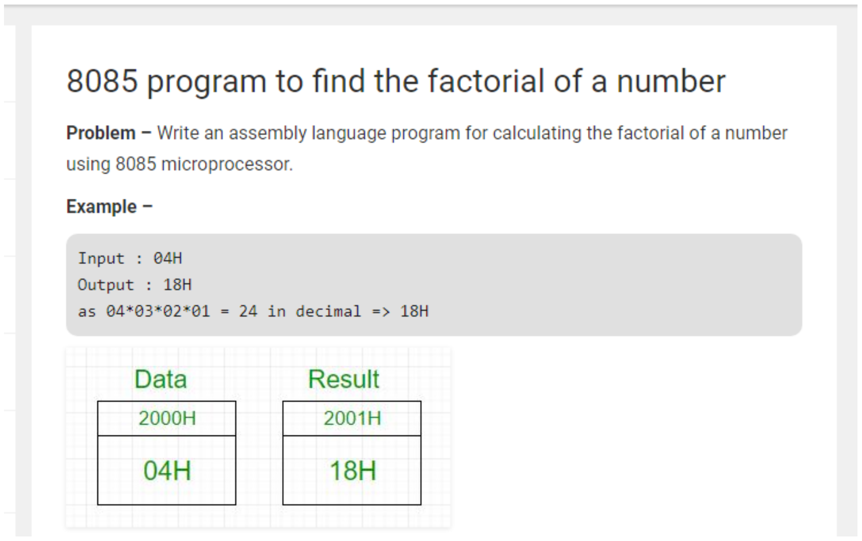
INX	r.p.	r.p. = r.p. + 1	INX H
DCR	R	R = R – 1	DCR B
DCR	M	M = Mc – 1	DCR 2050
DCX	r.p.	r.p. = r.p. – 1	DCX H
DAD	r.p.	HL = HL + r.p.	DAD H

In the table,
R stands for register
M stands for memory
Mc stands for memory contents
r.p. stands for register pair

GeeksforGeeks has prepared a complete interview preparation course with premium videos, theory, practice problems, TA support and many more features. Please refer Placement 100 for details

Practical 5

- **Aim**: Write an assembly language code in GNUsim8085 to find factorial of a number.

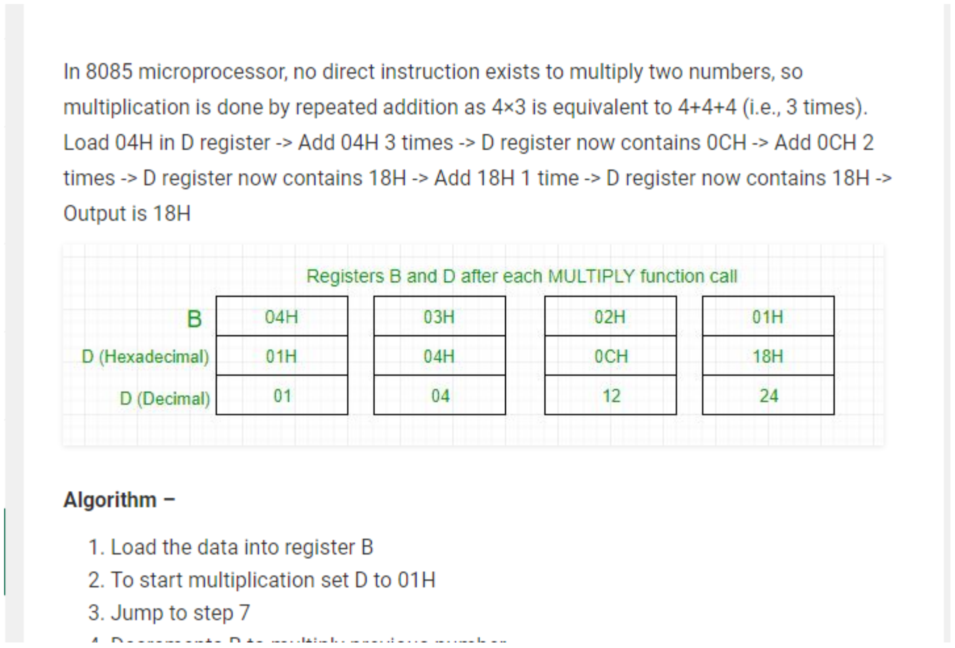


8085 program to find the factorial of a number

Problem – Write an assembly language program for calculating the factorial of a number using 8085 microprocessor.

Example –

```
Input : 04H
Output : 18H
as 04*03*02*01 = 24 in decimal => 18H
```



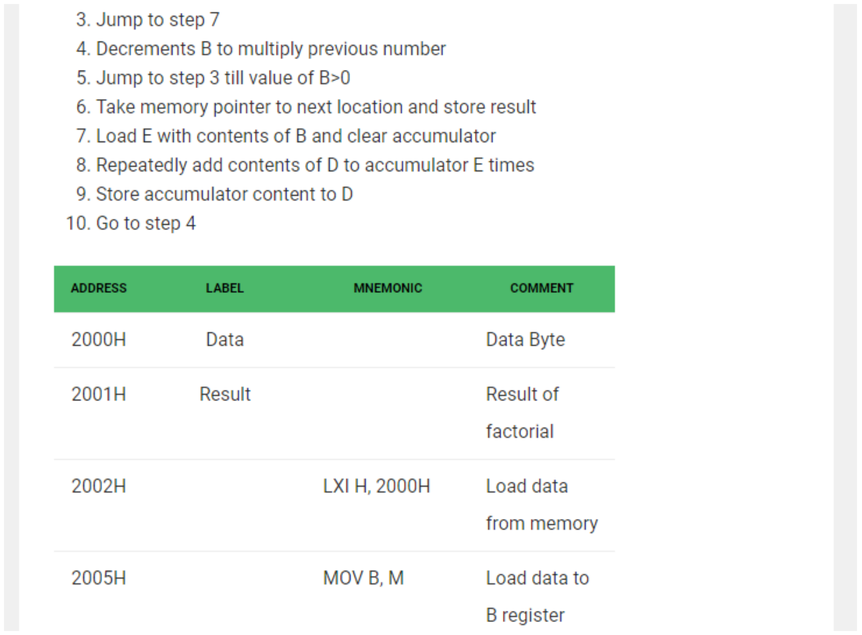
In 8085 microprocessor, no direct instruction exists to multiply two numbers, so multiplication is done by repeated addition as 4×3 is equivalent to 4+4+4 (i.e., 3 times). Load 04H in D register -> Add 04H 3 times -> D register now contains 0CH -> Add 0CH 2 times -> D register now contains 18H -> Add 18H 1 time -> D register now contains 18H -> Output is 18H

Registers B and D after each MULTIPLY function call

B	04H	03H	02H	01H
D (Hexadecimal)	01H	04H	0CH	18H
D (Decimal)	01	04	12	24

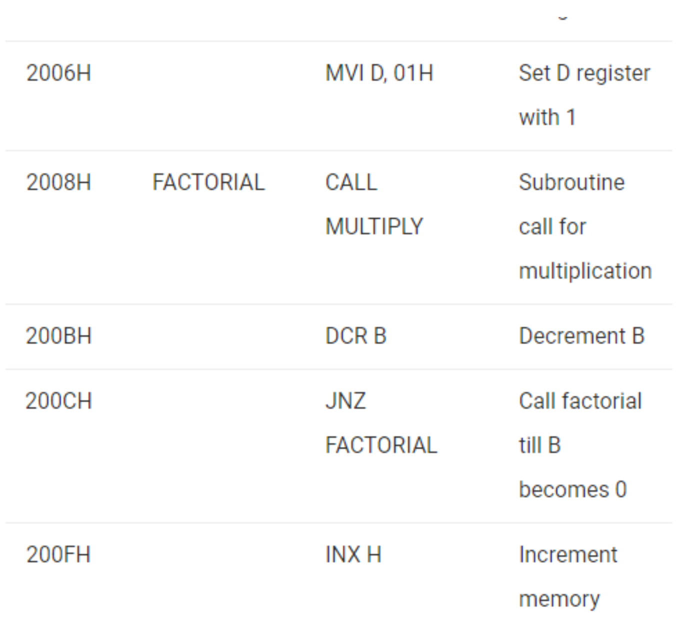
Algorithm –

1. Load the data into register B
2. To start multiplication set D to 01H
3. Jump to step 7

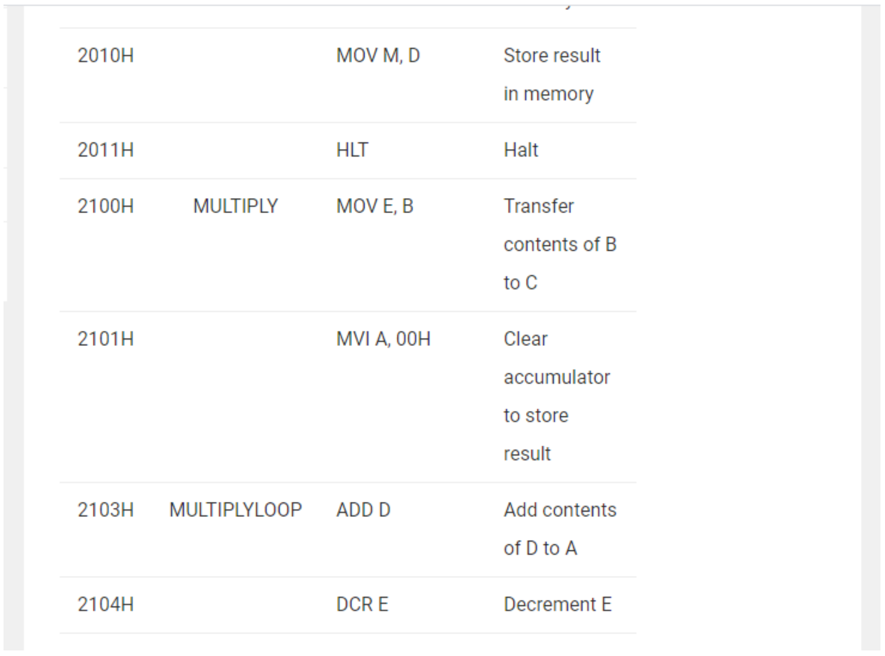


3. Jump to step 7
4. Decrements B to multiply previous number
5. Jump to step 3 till value of B>0
6. Take memory pointer to next location and store result
7. Load E with contents of B and clear accumulator
8. Repeatedly add contents of D to accumulator E times
9. Store accumulator content to D
10. Go to step 4

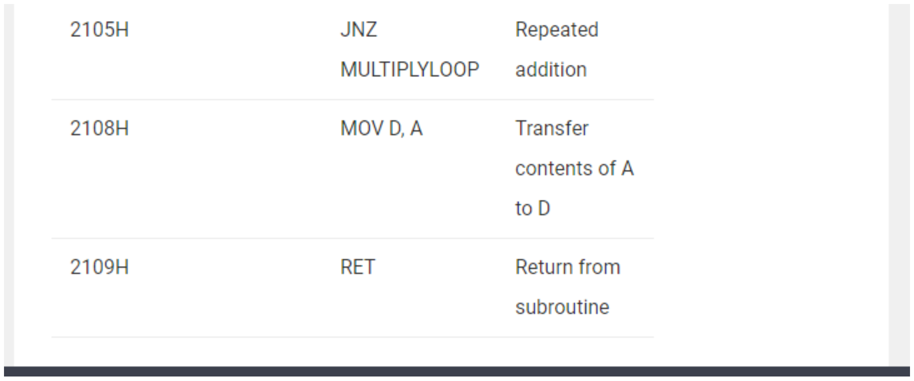
ADDRESS	LABEL	MNEMONIC	COMMENT
2000H	Data		Data Byte
2001H	Result		Result of factorial
2002H		LXI H, 2000H	Load data from memory
2005H		MOV B, M	Load data to B register



2006H		MVI D, 01H	Set D register with 1
2008H	FACTORIAL	CALL MULTIPLY	Subroutine call for multiplication
200BH		DCR B	Decrement B
200CH		JNZ FACTORIAL	Call factorial till B becomes 0
200FH		INX H	Increment memory



2010H		MOV M, D	Store result in memory
2011H		HLT	Halt
2100H	MULTIPLY	MOV E, B	Transfer contents of B to C
2101H		MVI A, 00H	Clear accumulator to store result
2103H	MULTIPLYLOOP	ADD D	Add contents of D to A
2104H		DCR E	Decrement E



2105H	JNZ MULTIPLYLOOP	Repeated addition
2108H	MOV D, A	Transfer contents of A to D
2109H	RET	Return from subroutine

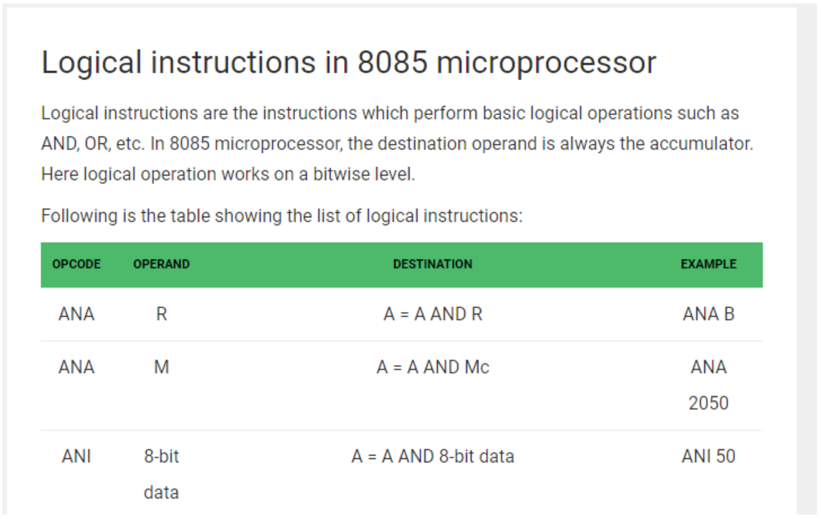
Explanation:

Step 1: First set register B with data.
Step 2: Set register D with data by calling MULTIPLY subroutine one time.
Step 3: Decrement B and add D to itself B times by calling MULTIPLY subroutine as 4*3 is equivalent to 4+4+4 (i.e., 3 times).
Step 4: Repeat the above step till B reaches 0 and then exit the program.
Step 5: The result is obtained in D register which is stored in memory.

Practical 6

- **Aim:** Write an assembly language code in GNUsim8085 to implement logical instructions.

The following table shows the list of logical instructions with their meanings.

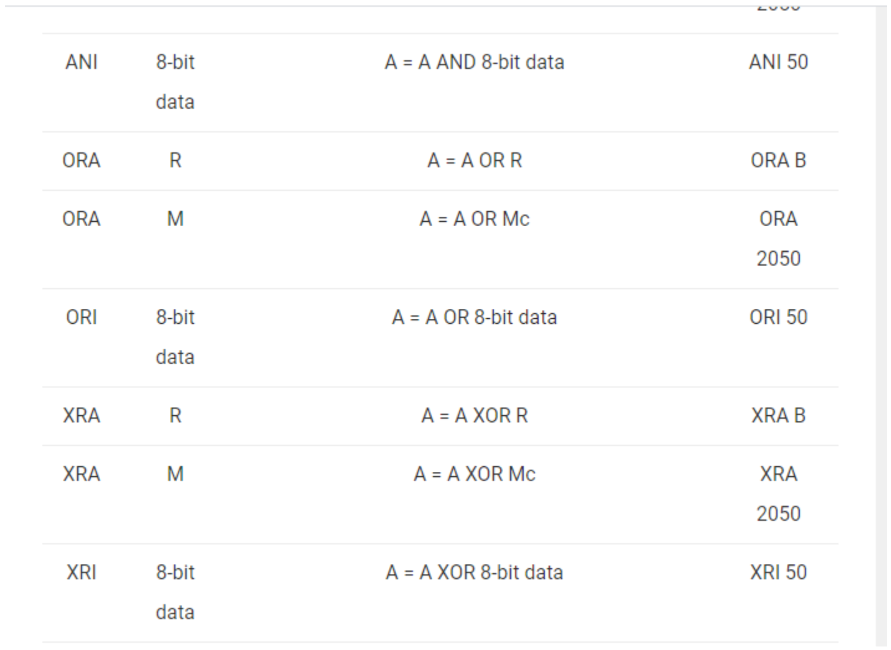


Logical instructions in 8085 microprocessor

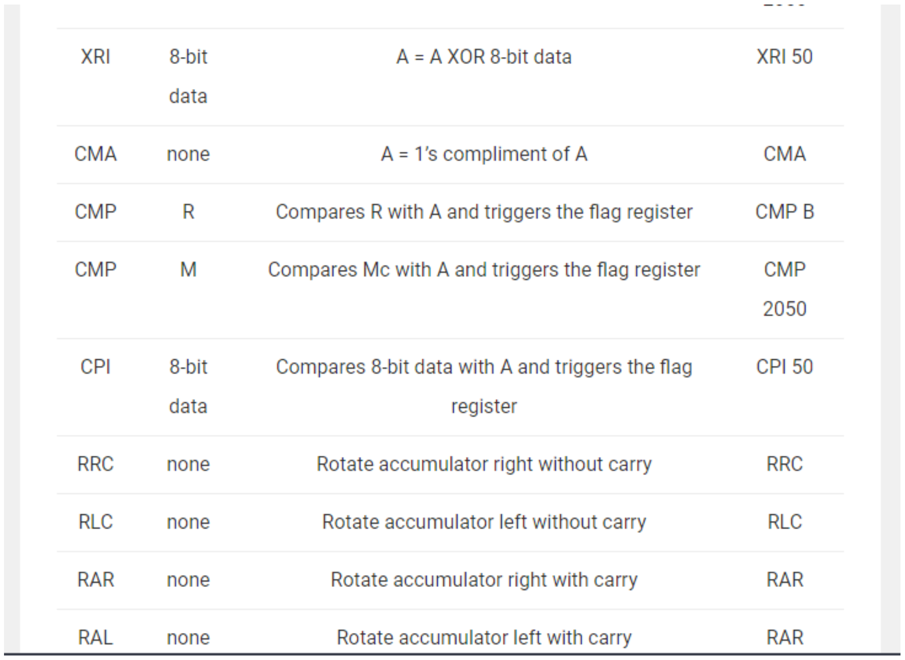
Logical instructions are the instructions which perform basic logical operations such as AND, OR, etc. In 8085 microprocessor, the destination operand is always the accumulator. Here logical operation works on a bitwise level.

Following is the table showing the list of logical instructions:

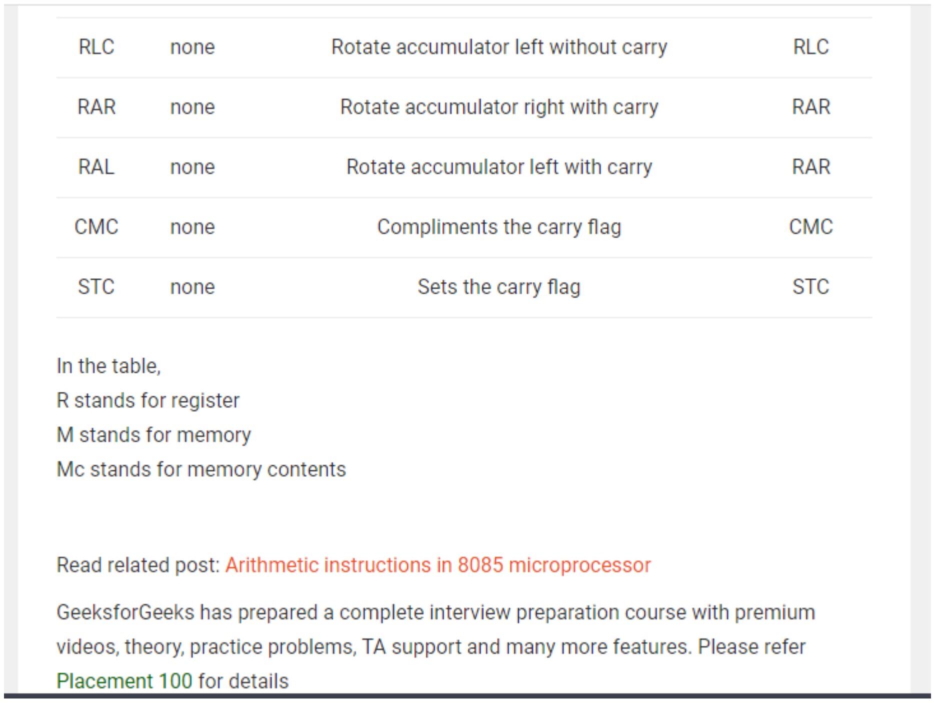
OPCODE	OPERAND	DESTINATION	EXAMPLE
ANA	R	A = A AND R	ANA B
ANA	M	A = A AND Mc	ANA 2050
ANI	8-bit data	A = A AND 8-bit data	ANI 50



ANI	8-bit data	A = A AND 8-bit data	ANI 50
ORA	R	A = A OR R	ORA B
ORA	M	A = A OR Mc	ORA 2050
ORI	8-bit data	A = A OR 8-bit data	ORI 50
XRA	R	A = A XOR R	XRA B
XRA	M	A = A XOR Mc	XRA 2050
XRI	8-bit data	A = A XOR 8-bit data	XRI 50



XRI	8-bit data	A = A XOR 8-bit data	XRI 50
CMA	none	A = 1's compliment of A	CMA
CMP	R	Compares R with A and triggers the flag register	CMP B
CMP	M	Compares Mc with A and triggers the flag register	CMP 2050
CPI	8-bit data	Compares 8-bit data with A and triggers the flag register	CPI 50
RRC	none	Rotate accumulator right without carry	RRC
RLC	none	Rotate accumulator left without carry	RLC
RAR	none	Rotate accumulator right with carry	RAR
RAL	none	Rotate accumulator left with carry	RAR



RLC	none	Rotate accumulator left without carry	RLC
RAR	none	Rotate accumulator right with carry	RAR
RAL	none	Rotate accumulator left with carry	RAR
CMC	none	Compliments the carry flag	CMC
STC	none	Sets the carry flag	STC

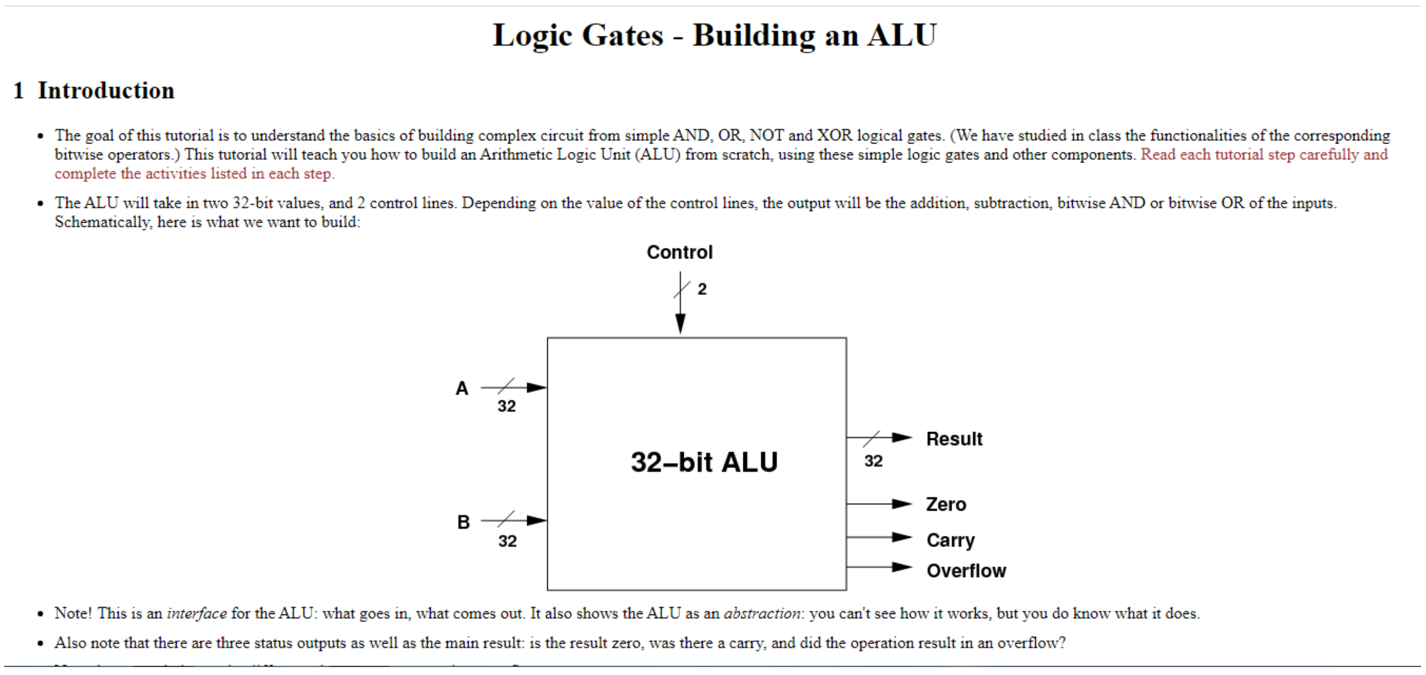
In the table,
R stands for register
M stands for memory
Mc stands for memory contents

Read related post: Arithmetic instructions in 8085 microprocessor

GeeksforGeeks has prepared a complete interview preparation course with premium videos, theory, practice problems, TA support and many more features. Please refer Placement 100 for details

Practical 7

- **Aim** : Design ALU using Logism.



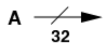
Logic Gates - Building an ALU

1 Introduction

- The goal of this tutorial is to understand the basics of building complex circuit from simple AND, OR, NOT and XOR logical gates. (We have studied in class the functionalities of the corresponding bitwise operators.) This tutorial will teach you how to build an Arithmetic Logic Unit (ALU) from scratch, using these simple logic gates and other components. Read each tutorial step carefully and complete the activities listed in each step.
- The ALU will take in two 32-bit values, and 2 control lines. Depending on the value of the control lines, the output will be the addition, subtraction, bitwise AND or bitwise OR of the inputs. Schematically, here is what we want to build:

- Note! This is an *interface* for the ALU: what goes in, what comes out. It also shows the ALU as an *abstraction*: you can't see how it works, but you do know what it does.
- Also note that there are three status outputs as well as the main result: is the result zero, was there a carry, and did the operation result in an overflow?

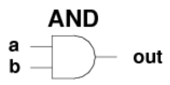
- Also note that there are three status outputs as well as the main result: is the result zero, was there a carry, and did the operation result in an overflow?
- Note: just a reminder on the difference between a carry and an overflow:
  - **Carry**: was there a carry in the most-significant bit which we could not output in the given number of bits (32 bits, above)?
  - **Overflow**: does the sign of the output differ from the inputs, indicating (for example) that a sum of two positive numbers has overflowed and is now a ne
- Some of the data and control lines are shown with a slash and a number like 32.



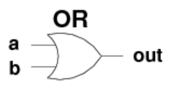
- This indicates that the line is actually 32 lines in parallel, e.g. the result is 32-bits wide. If you don't see a slash in the diagrams below, you can assume that the l

1.1 Basic Components

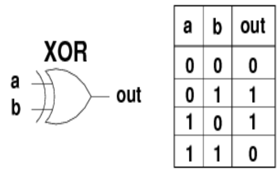
- We are going to use four logic gates: AND, OR, NOT and XOR. You should Below are the icons for each, and their truth table. (We have already seen the truth



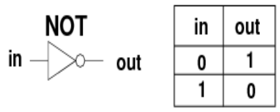
a	b	out
0	0	0
0	1	0
1	0	0
1	1	1



a	b	out
0	0	0
0	1	1
1	0	1
1	1	1

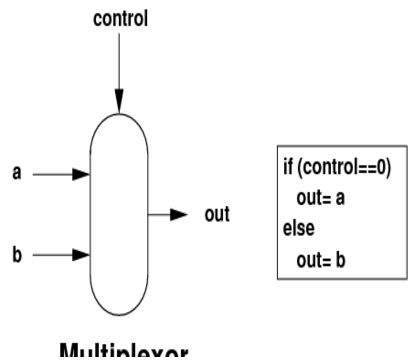


a	b	out
0	0	0
0	1	1
1	0	1
1	1	0



in	out
0	1
1	0

- We are going to use another component, the *multiplexor*. The job of the multiplexor is to choose one of several inputs, based on a control line, and send the chosen input to the output.



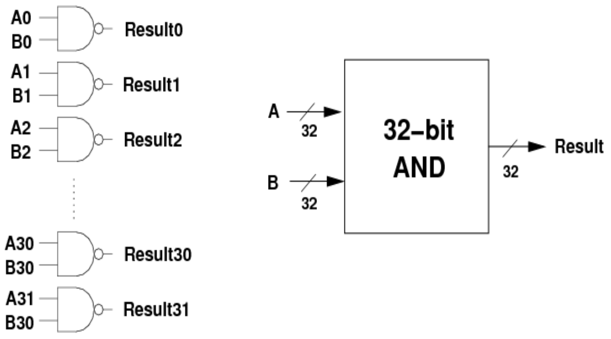
```
if (control==0)
  out= a
else
  out= b
```

Multiplexor

- This multiplexor above is a 1-bit wide 2-way multiplexor: 2 inputs, 1 bit wide each. If you add extra control lines, you can choose more inputs: 2 control lines is used in a 4-way multiplexo
  lines in an 8-way multiplexor etc.
- And by using multiple multiplexors in parallel, you can make N-bit wide multiplexors.
- I'm not going to show you the internals of the multiplexor. You should, however, know that it can be easily built with the 4 logic gates above.

1.2 Bitwise AND

- Bitwise AND is very useful, for example to calculate an IP network's identity:
  - 131.245.7.18 AND 255.255.255.0 => 131.245.7.0
- Building the logic to do 32-bit AND on two inputs is easy: as each bit is independent, we just need 32 AND gates in parallel.



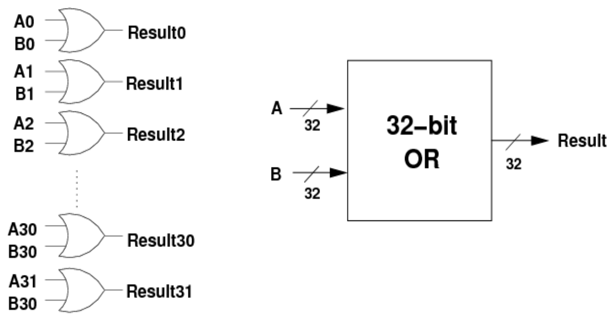
- Note the interface diagram on the right. Each time we build a larger component, we are going to hide it behind an interface.

What To Do

- Download the digital logic circuit simulator Logisim. Make sure you select the correct binary for your platform.
- Run Logisim and and go through the tutorial available under the Help menu. Cover the Beginner's section first, then the Subcircuits and the Wire bundle section. You will need to use a split
your ALU implementation. The interface is simple and intuitive.
- Here is a 4-bit ALU implemented in Logisim: ALU4.circ. Download this file and make a copy of it called ALU6.circ. (You will need the original ALU4.circ in later steps.) Open ALU6.circ
then double-click on the 4-bit AND component in the left drop-down menu. The 4-bit AND circuit should open up for you. Select the Hand icon in the top-left of the Logisim window, then
data inputs to change their values. Make sure that the AND component works as expected.
- Extend the AND component to work with 6-bit values instead of 4 bits.

1.3 Bitwise OR

- 32-bit bitwise OR is just as easy as bitwise AND, we just need 32 OR gates in parallel.

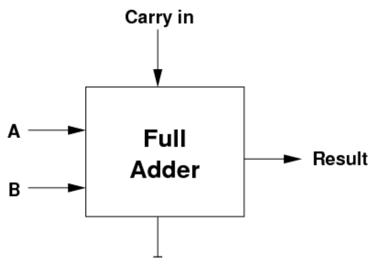


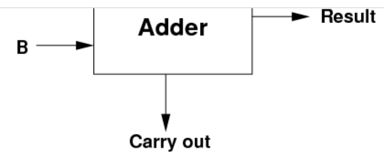
What To Do

- Make a copy of your current ALU4.circ version, in case you need to revert to it later.
- Open the 4-bit OR circuit by double-clicking on it in the left drop-down menu. Select the Hand icon in the top-left of the Logisim window, then click on the data inputs to change their value
that the OR component works as expected.
- Extend the OR component to work with 6-bit values instead of 4 bits.

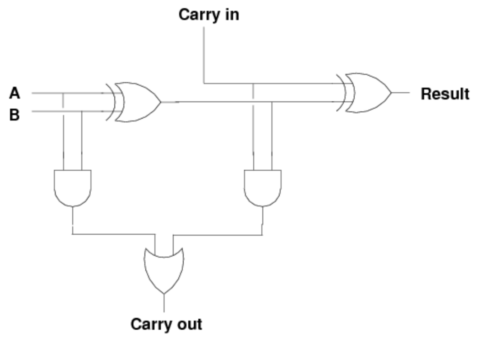
1.4 Addition

- Addition isn't going to be so easy.
- We saw previously that we have to add bits, and this may produce a carry. Columns further up need to accept a carry as input, along with two inputs, and produce the 1-bit output and anothe
the next column up.
- The component which will perform a 1-bit ADD, receiving a carry in and producing a 1-bit output and a carry out is called a **full adder**. Its interface looks like the following:

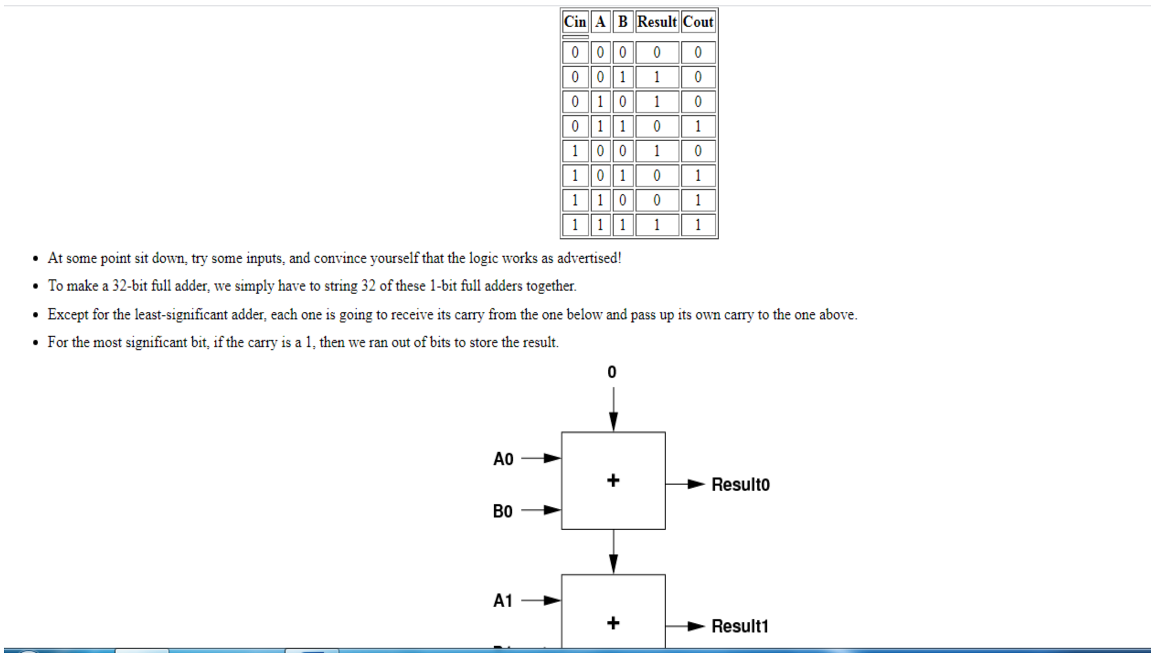




- Internally, here is what a full adder looks like, just 5 logic gates.

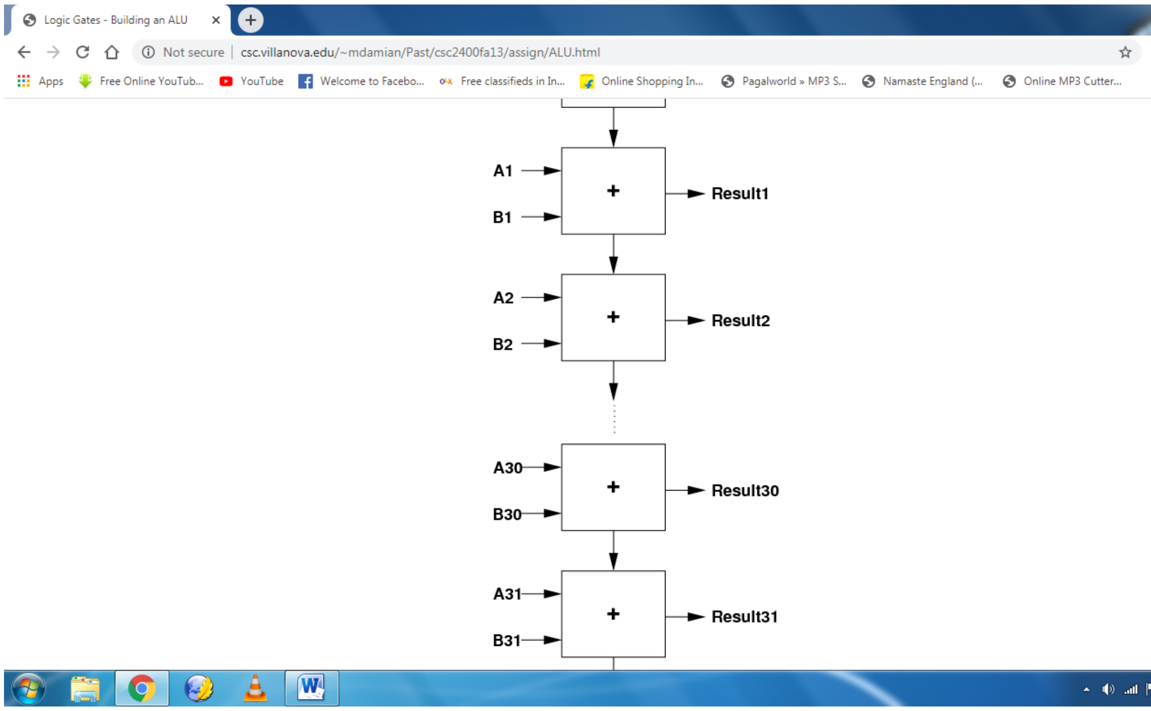


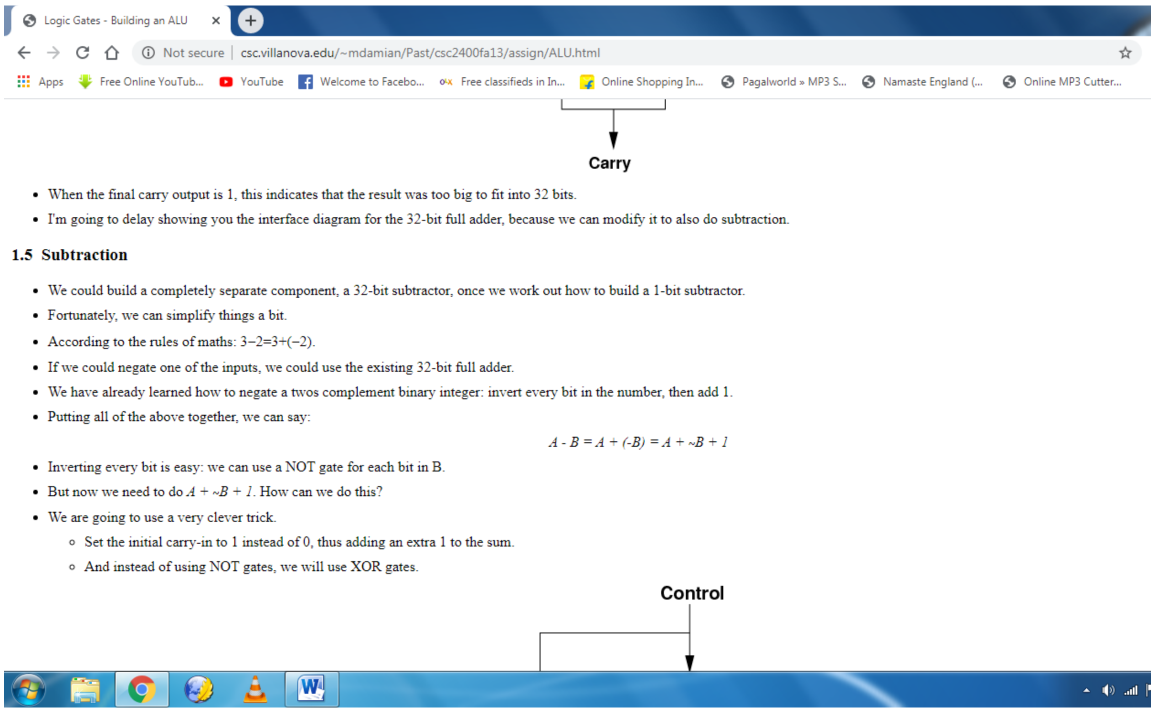
- The truth table for the full adder is:



Cin	A	B	Result	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

- At some point sit down, try some inputs, and convince yourself that the logic works as advertised!
- To make a 32-bit full adder, we simply have to string 32 of these 1-bit full adders together.
- Except for the least-significant adder, each one is going to receive its carry from the one below and pass up its own carry to the one above.
- For the most significant bit, if the carry is a 1, then we ran out of bits to store the result.





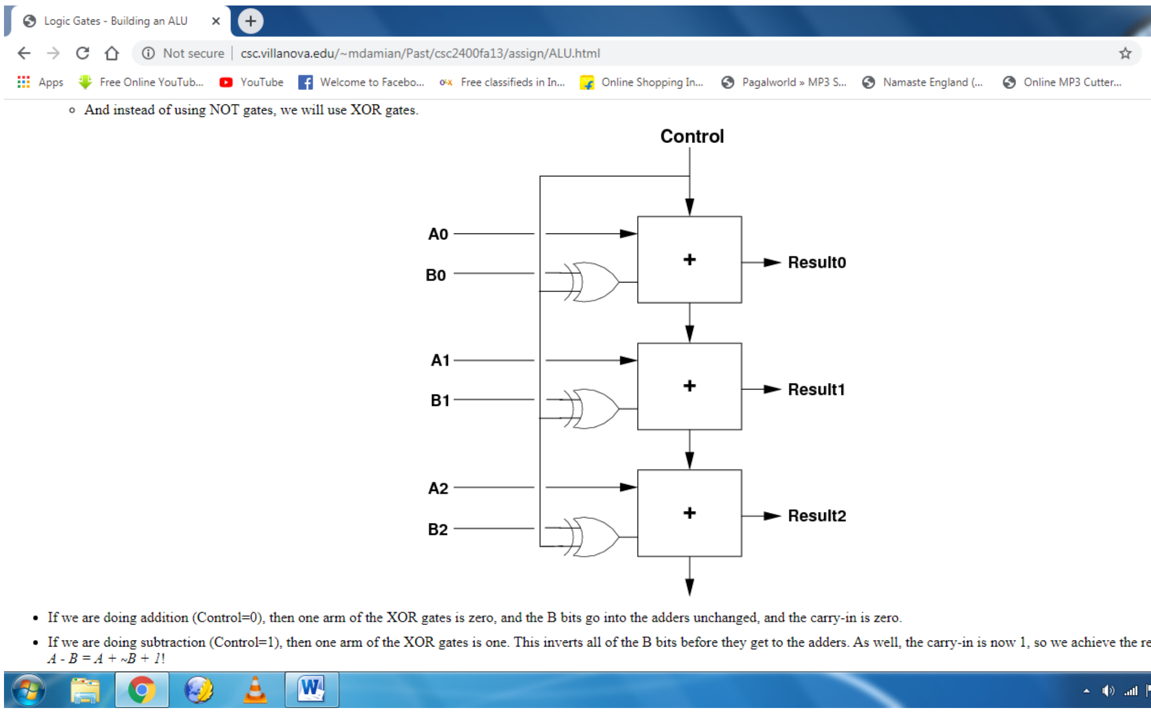
- When the final carry output is 1, this indicates that the result was too big to fit into 32 bits.
- I'm going to delay showing you the interface diagram for the 32-bit full adder, because we can modify it to also do subtraction.

1.5 Subtraction

- We could build a completely separate component, a 32-bit subtractor, once we work out how to build a 1-bit subtractor.
- Fortunately, we can simplify things a bit.
- According to the rules of maths: 3−2=3+(−2).
- If we could negate one of the inputs, we could use the existing 32-bit full adder.
- We have already learned how to negate a twos complement binary integer: invert every bit in the number, then add 1.
- Putting all of the above together, we can say:

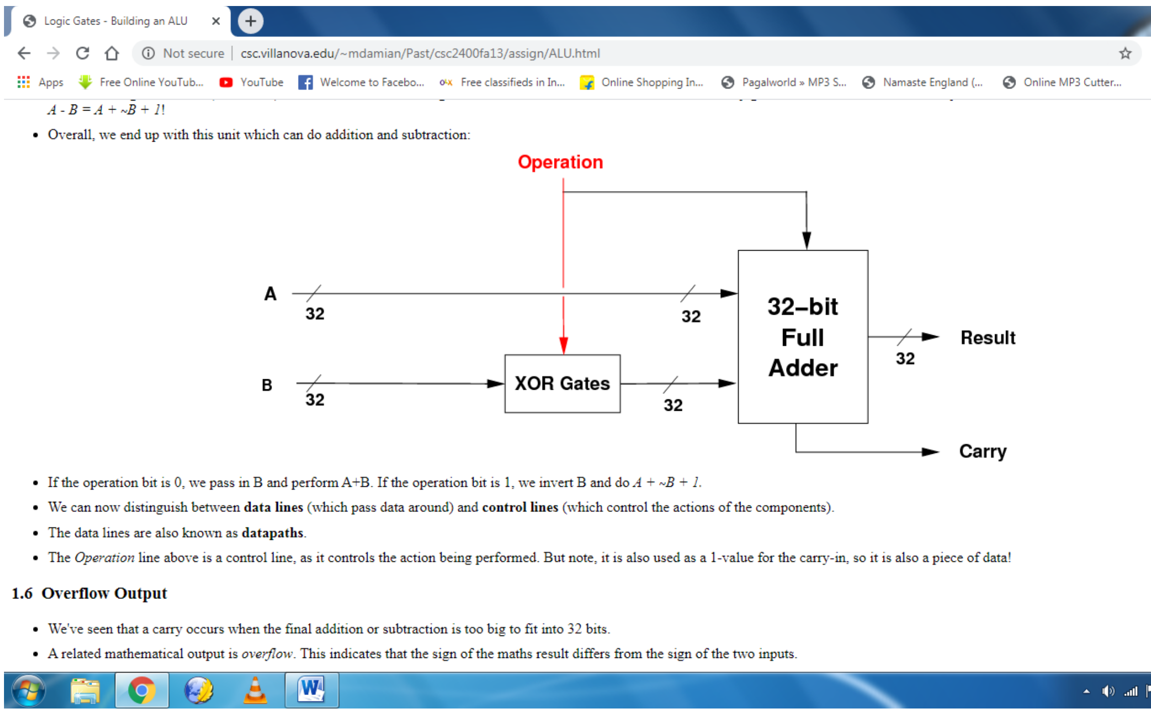
$$A - B = A + (-B) = A + \sim B + 1$$

- Inverting every bit is easy: we can use a NOT gate for each bit in B.
- But now we need to do $A + \sim B + 1$. How can we do this?
- We are going to use a very clever trick.
    - Set the initial carry-in to 1 instead of 0, thus adding an extra 1 to the sum.
    - And instead of using NOT gates, we will use XOR gates.



- And instead of using NOT gates, we will use XOR gates.

- If we are doing addition (Control=0), then one arm of the XOR gates is zero, and the B bits go into the adders unchanged, and the carry-in is zero.
- If we are doing subtraction (Control=1), then one arm of the XOR gates is one. This inverts all of the B bits before they get to the adders. As well, the carry-in is now 1, so we achieve the re $A - B = A + \sim B + 1$!



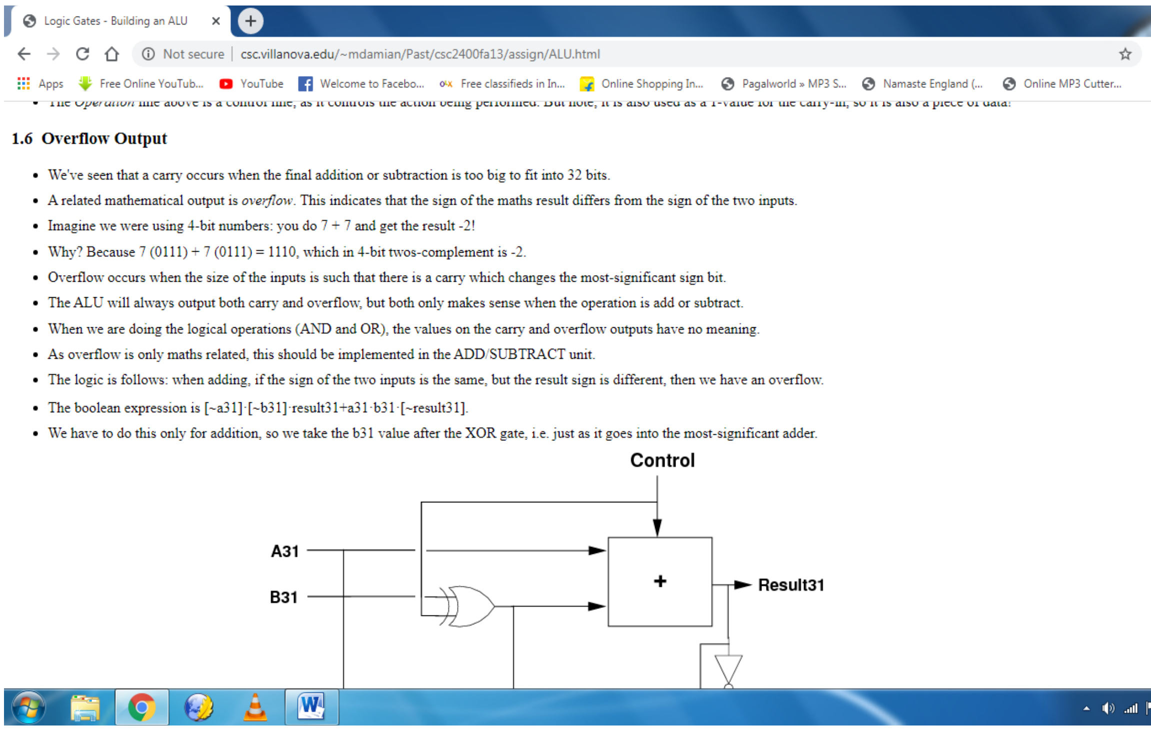
A - *B* = *A* + ~*B* + *1*!

- Overall, we end up with this unit which can do addition and subtraction:

- If the operation bit is 0, we pass in B and perform A+B. If the operation bit is 1, we invert B and do *A* + ~*B* + *1*.
- We can now distinguish between **data lines** (which pass data around) and **control lines** (which control the actions of the components).
- The data lines are also known as **datapaths**.
- The *Operation* line above is a control line, as it controls the action being performed. But note, it is also used as a 1-value for the carry-in, so it is also a piece of data!

1.6 Overflow Output

- We've seen that a carry occurs when the final addition or subtraction is too big to fit into 32 bits.
- A related mathematical output is *overflow*. This indicates that the sign of the maths result differs from the sign of the two inputs.



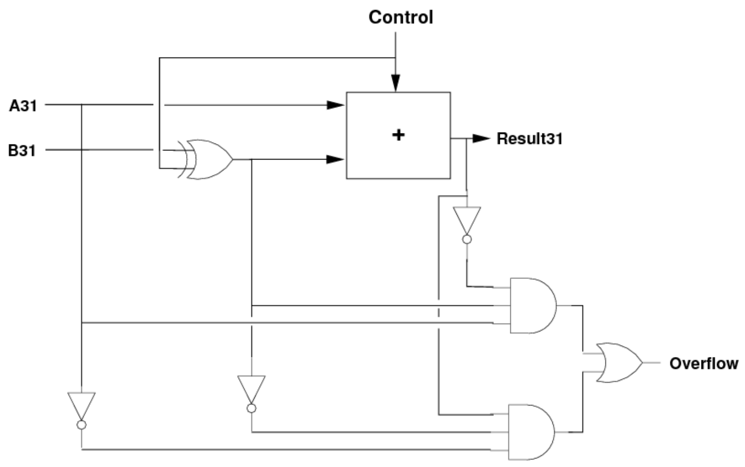
- The *Operation* line above is a control line, as it controls the action being performed. But note, it is also used as a 1-value for the carry-in, so it is also a piece of data!

1.6 Overflow Output

- We've seen that a carry occurs when the final addition or subtraction is too big to fit into 32 bits.
- A related mathematical output is *overflow*. This indicates that the sign of the maths result differs from the sign of the two inputs.
- Imagine we were using 4-bit numbers: you do 7 + 7 and get the result -2!
- Why? Because 7 (0111) + 7 (0111) = 1110, which in 4-bit twos-complement is -2.
- Overflow occurs when the size of the inputs is such that there is a carry which changes the most-significant sign bit.
- The ALU will always output both carry and overflow, but both only makes sense when the operation is add or subtract.
- When we are doing the logical operations (AND and OR), the values on the carry and overflow outputs have no meaning.
- As overflow is only maths related, this should be implemented in the ADD/SUBTRACT unit.
- The logic is follows: when adding, if the sign of the two inputs is the same, but the result sign is different, then we have an overflow.
- The boolean expression is [~a31]·[~b31]·result31+a31·b31·[~result31].
- We have to do this only for addition, so we take the b31 value after the XOR gate, i.e. just as it goes into the most-significant adder.

1.6 Overflow Output

- We've seen that a carry occurs when the final addition or subtraction is too big to fit into 32 bits.
- A related mathematical output is *overflow*. This indicates that the sign of the maths result differs from the sign of the two inputs.
- Imagine we were using 4-bit numbers: you do 7 + 7 and get the result -2!
- Why? Because 7 (0111) + 7 (0111) = 1110, which in 4-bit twos-complement is -2.
- Overflow occurs when the size of the inputs is such that there is a carry which changes the most-significant sign bit.
- The ALU will always output both carry and overflow, but both only makes sense when the operation is add or subtract.
- When we are doing the logical operations (AND and OR), the values on the carry and overflow outputs have no meaning.
- As overflow is only maths related, this should be implemented in the ADD/SUBTRACT unit.
- The logic is follows: when adding, if the sign of the two inputs is the same, but the result sign is different, then we have an overfl
- The boolean expression is [~a31]·[~b31]·result31+a31·b31·[~result31].
- We have to do this only for addition, so we take the b31 value after the XOR gate, i.e. just as it goes into the most-significant add

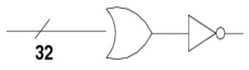


1.7 Negative Output

- When is the result negative? When its most-significant bit is 1.
- We can wire this bit directly out of the ALU, so that it indicates if the result is negative.

1.8 Zero Output

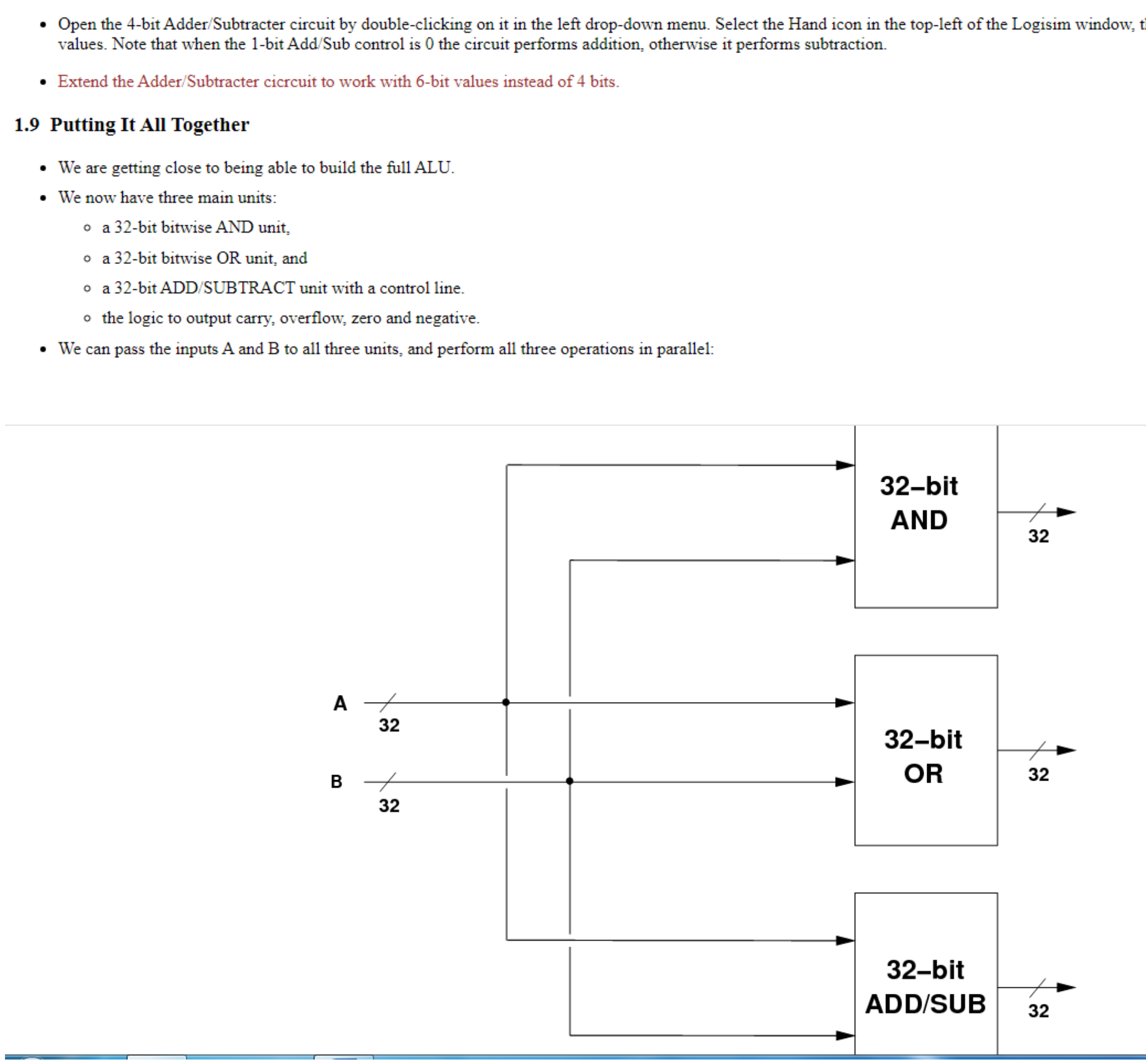
- One thing left to do is to provide the zero output: is the result zero?
- This is only true if **all** of the bits of the result are zero. We can do this for the logical and the maths units.
- To do this, we can use a 32-bit OR gate followed by a 1-bit NOT gate:



- The OR gate outputs a 0 only if all input bits are 0. If any input bit is a 1, the OR's output is a 1.
- The NOT gate simply inverts this, giving the result:
  - if any result bit is on, the output is 0, i.e. not zero.
  - if all result bits are off, the output is 1, i.e. zero.

What To Do

- Make a copy of your current ALU4.circ version, in case you need to revert to it later.
- Open the 4-bit Adder/Subtracter circuit by double-clicking on it in the left drop-down menu. Select the Hand icon in the top-left of the Logisim window, then click on the data inputs to char values. Note that when the 1-bit Add/Sub control is 0 the circuit performs addition, otherwise it performs subtraction.



- Open the 4-bit Adder/Subtracter circuit by double-clicking on it in the left drop-down menu. Select the Hand icon in the top-left of the Logisim window, t values. Note that when the 1-bit Add/Sub control is 0 the circuit performs addition, otherwise it performs subtraction.
- Extend the Adder/Subtracter cicrcuit to work with 6-bit values instead of 4 bits.

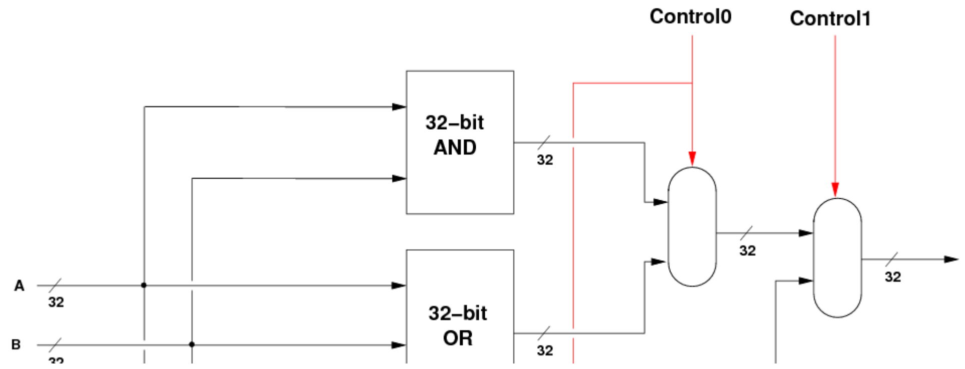
1.9 Putting It All Together

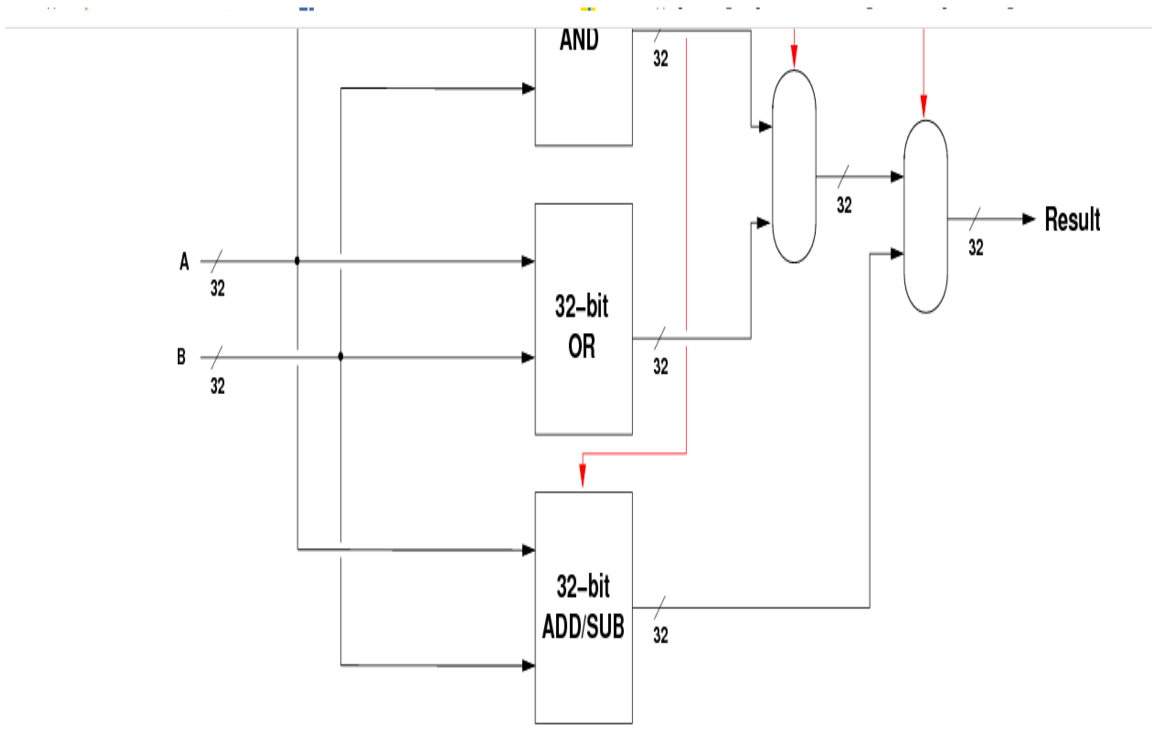
- We are getting close to being able to build the full ALU.
- We now have three main units:
  - a 32-bit bitwise AND unit,
  - a 32-bit bitwise OR unit, and
  - a 32-bit ADD/SUBTRACT unit with a control line.
  - the logic to output carry, overflow, zero and negative.
- We can pass the inputs A and B to all three units, and perform all three operations in parallel:

- But, we need to choose which of the three results we want, based on the two control lines coming into the ALU. We would like:

c_1	c_0	Result
0	0	A AND B
0	1	A OR B
1	0	A + B
1	1	A - B

- How do we control which of the four operations actually becomes the result? With multiplexors again.

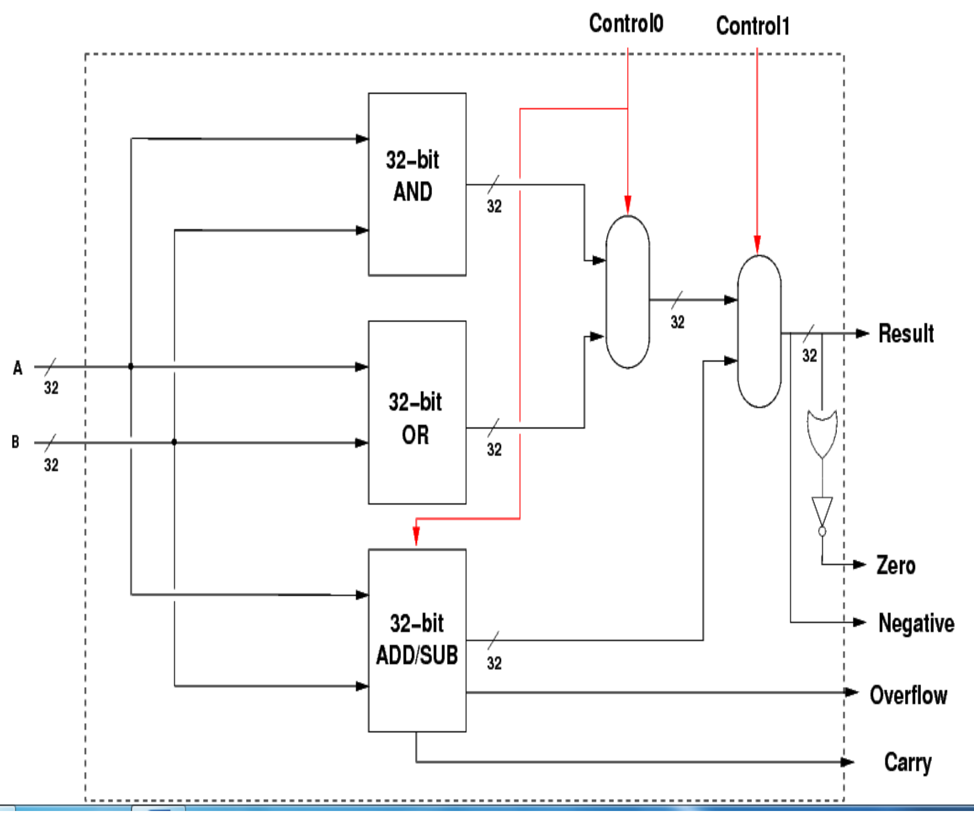


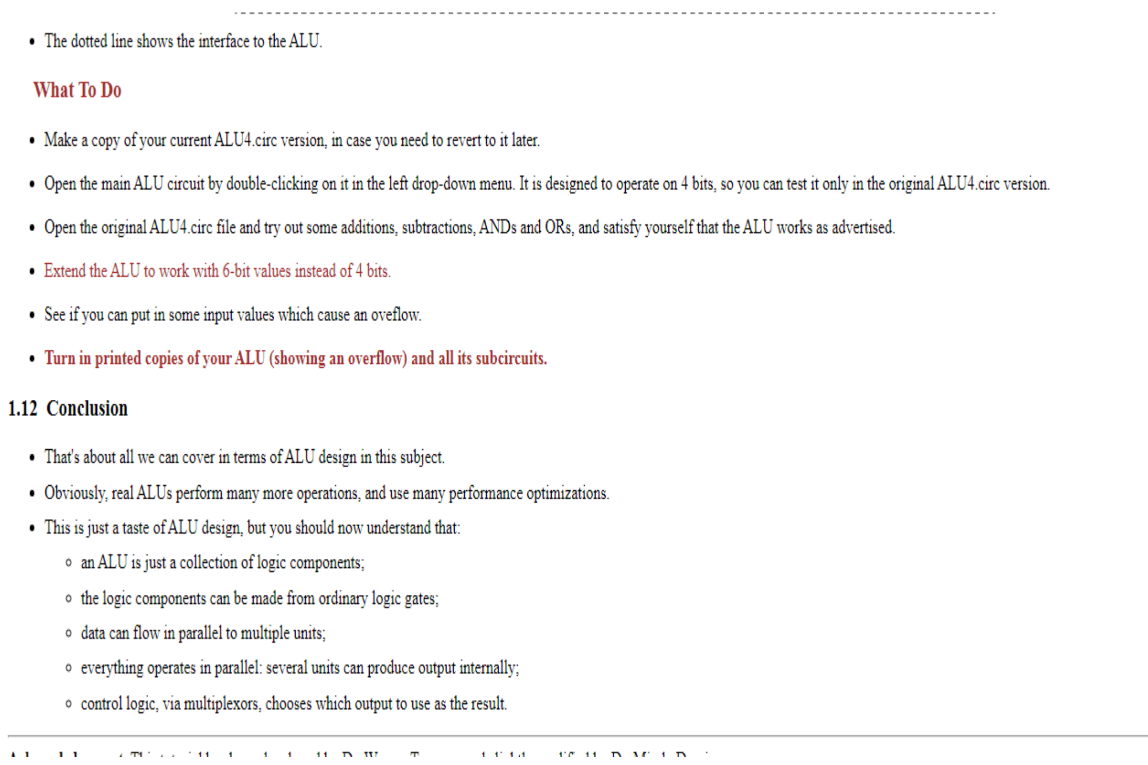


- When c_0=0, the ANDORresult is A AND B, and the ADDSUBresult is A + B.
- When c_0=1, the ANDORresult is A OR B, and the ADDSUBresult is A - B.
- Now we just have to choose between these two results, and that is done with the second multiplexor, which chooses either the bitwise logic result or the maths result.

1.10 Finally

- To finish off, we can draw the three components, the multiplexors and the zero logic, to reveal the final 32-bit ALU.


Control0
Control1
32-bit AND
32-bit OR
32-bit ADD/SUB
A
B
32
Result
Zero
Negative
Overflow
Carry



- The dotted line shows the interface to the ALU.

What To Do

- Make a copy of your current ALU4.circ version, in case you need to revert to it later.
- Open the main ALU circuit by double-clicking on it in the left drop-down menu. It is designed to operate on 4 bits, so you can test it only in the original ALU4.circ version.
- Open the original ALU4.circ file and try out some additions, subtractions, ANDs and ORs, and satisfy yourself that the ALU works as advertised.
- Extend the ALU to work with 6-bit values instead of 4 bits.
- See if you can put in some input values which cause an oveflow.
- **Turn in printed copies of your ALU (showing an overflow) and all its subcircuits.**

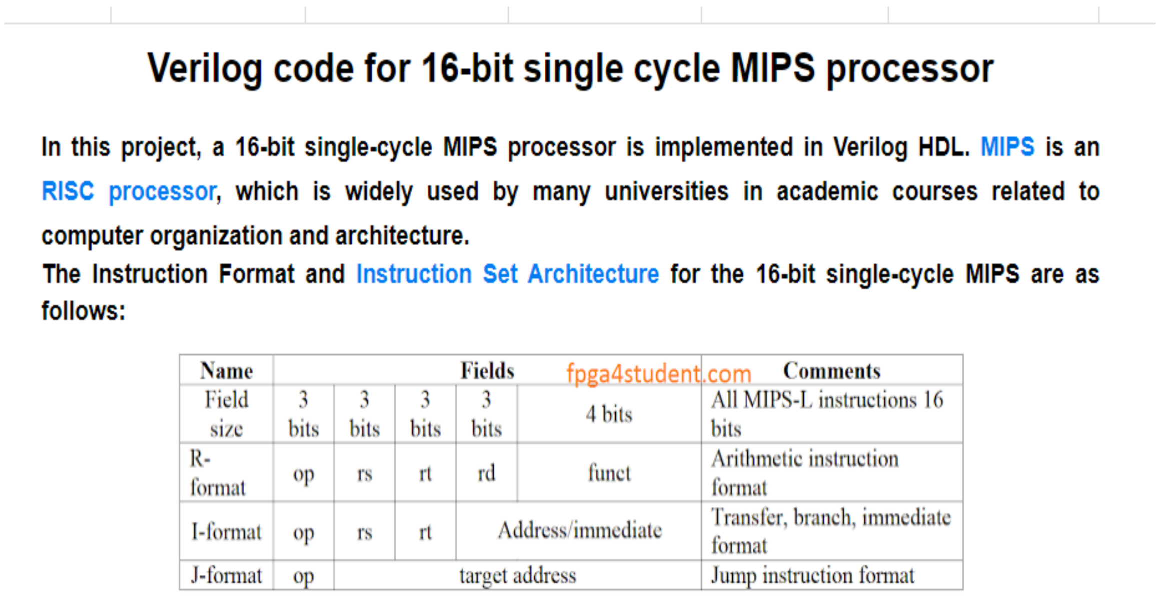
1.12 Conclusion

- That's about all we can cover in terms of ALU design in this subject.
- Obviously, real ALUs perform many more operations, and use many performance optimizations.
- This is just a taste of ALU design, but you should now understand that:
  - an ALU is just a collection of logic components;
  - the logic components can be made from ordinary logic gates;
  - data can flow in parallel to multiple units;
  - everything operates in parallel: several units can produce output internally;
  - control logic, via multiplexors, chooses which output to use as the result.

[illegible]

Practical 8

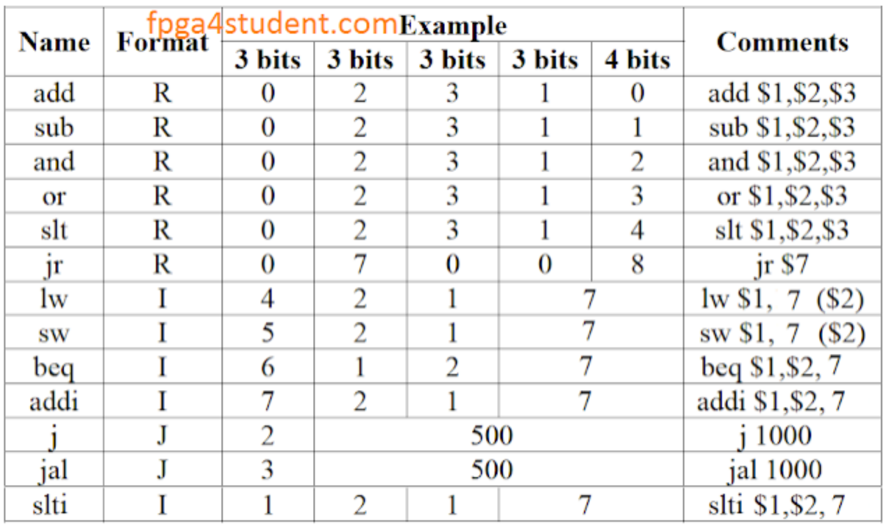
- **Aim:** Implement 16-bit single cycle MIPS processor in verilog HDL



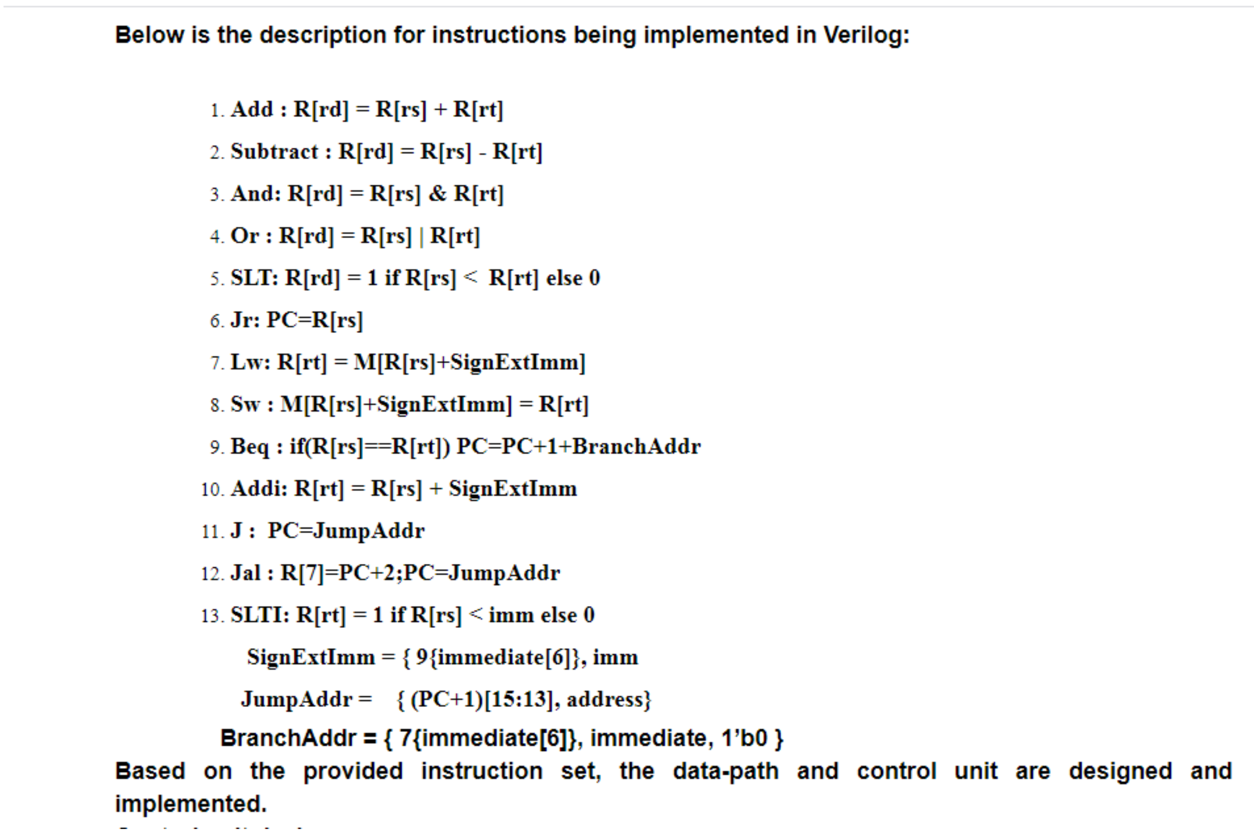
Verilog code for 16-bit single cycle MIPS processor

In this project, a 16-bit single-cycle MIPS processor is implemented in Verilog HDL. MIPS is an RISC processor, which is widely used by many universities in academic courses related to computer organization and architecture.
The Instruction Format and Instruction Set Architecture for the 16-bit single-cycle MIPS are as follows:

Name	Fields					Comments
Field size	3 bits	3 bits	3 bits	3 bits	4 bits	All MIPS-L instructions 16 bits
R-format	op	rs	rt	rd	funct	Arithmetic instruction format
I-format	op	rs	rt	Address/immediate		Transfer, branch, immediate format
J-format	op	target address				Jump instruction format



Name	Format	Example					Comments
		3 bits	3 bits	3 bits	3 bits	4 bits	
add	R	0	2	3	1	0	add $1,$2,$3
sub	R	0	2	3	1	1	sub $1,$2,$3
and	R	0	2	3	1	2	and $1,$2,$3
or	R	0	2	3	1	3	or $1,$2,$3
slt	R	0	2	3	1	4	slt $1,$2,$3
jr	R	0	7	0	0	8	jr $7
lw	I	4	2	1	7		lw $1, 7 ($2)
sw	I	5	2	1	7		sw $1, 7 ($2)
beq	I	6	1	2	7		beq $1,$2, 7
addi	I	7	2	1	7		addi $1,$2, 7
j	J	2	500				j 1000
jal	J	3	500				jal 1000
slti	I	1	2	1	7		slti $1,$2, 7



Below is the description for instructions being implemented in Verilog:

1. **Add : R[rd] = R[rs] + R[rt]**
2. **Subtract : R[rd] = R[rs] - R[rt]**
3. **And: R[rd] = R[rs] & R[rt]**
4. **Or : R[rd] = R[rs] | R[rt]**
5. **SLT: R[rd] = 1 if R[rs] < R[rt] else 0**
6. **Jr: PC=R[rs]**
7. **Lw: R[rt] = M[R[rs]+SignExtImm]**
8. **Sw : M[R[rs]+SignExtImm] = R[rt]**
9. **Beq : if(R[rs]==R[rt]) PC=PC+1+BranchAddr**
10. **Addi: R[rt] = R[rs] + SignExtImm**
11. **J : PC=JumpAddr**
12. **Jal : R[7]=PC+2;PC=JumpAddr**
13. **SLTI: R[rt] = 1 if R[rs] < imm else 0**

SignExtImm = { 9{immediate[6]}, imm

JumpAddr = { (PC+1)[15:13], address}

BranchAddr = { 7{immediate[6]}, immediate, 1'b0 }

Based on the provided instruction set, the data-path and control unit are designed and implemented.

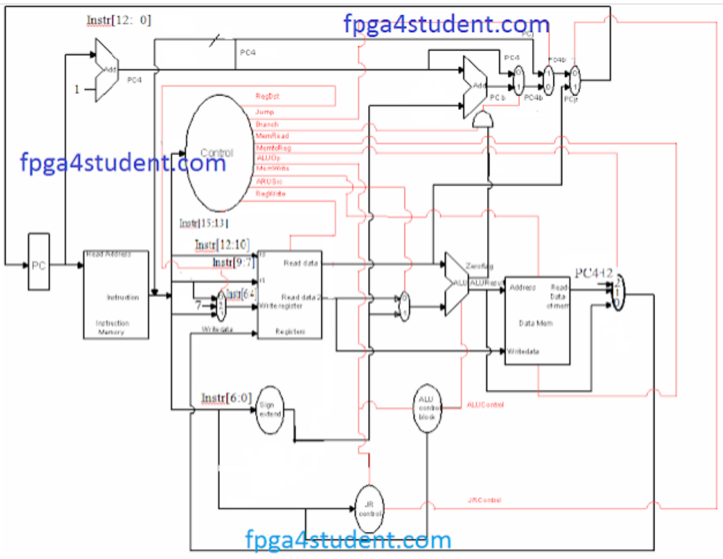
BranchAddr = { 7{immediate[6]}, immediate, 1'b0 }

Based on the provided instruction set, the data-path and control unit are designed and implemented.

Control unit design:

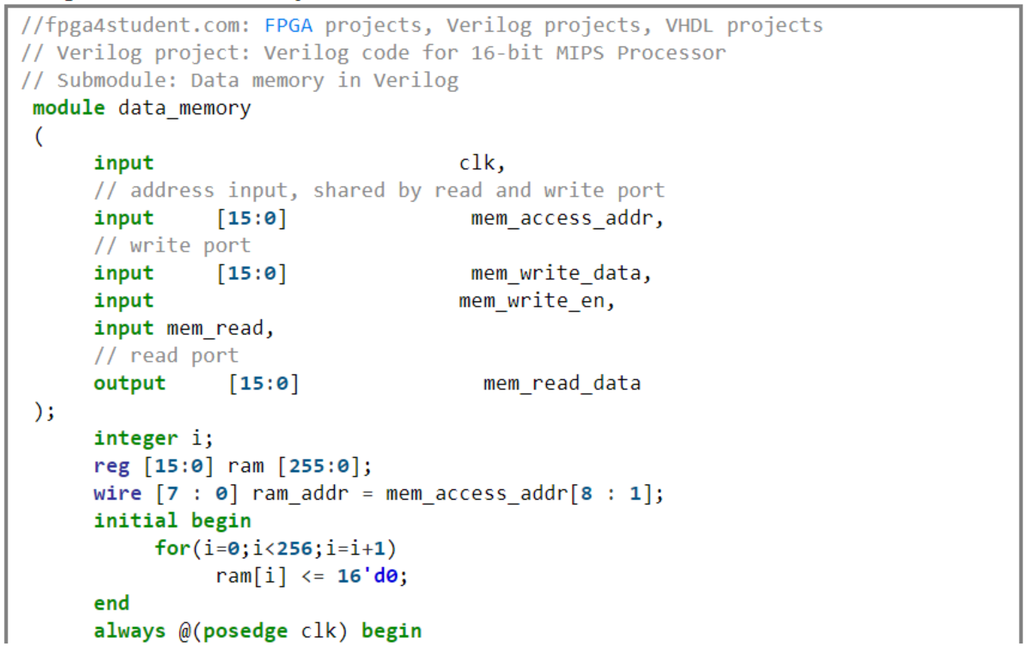
Control signals									
Instruction	**Reg Dst**	**ALU Src**	**Memto Reg**	**Reg Write**	**MemRead**	**Mem Write**	**Branch**	**ALUOp**	**Jump**
R-type	1	0	0	1	0	0	0	00	0
LW	0	1	1	1	1	0	0	11	0
SW	0	1	0	0	0	1	0	11	0
addi	0	1	0	1	0	0	0	11	0
beq	0	0	0	0	0	0	1	01	0
j	0	0	0	0	0	0	0	00	1
jal	2	0	2	1	0	0	0	00	1
slti	0	1	0	1	0	0	0	10	0

ALU Control				
ALU op	**Function**	**ALUcnt**	**ALU Operation**	**Instruction**
11	xxxx	000	ADD	Addi,lw,sw
01	xxxx	001	SUB	BEQ
00	00	000	ADD	R-type: ADD
00	01	001	SUB	R-type: sub
00	02	010	AND	R-type: AND
00	03	011	OR	R-type: OR
00	04	100	slt	R-type: slt
10	xxxxxx	100	slt	i-type: slti

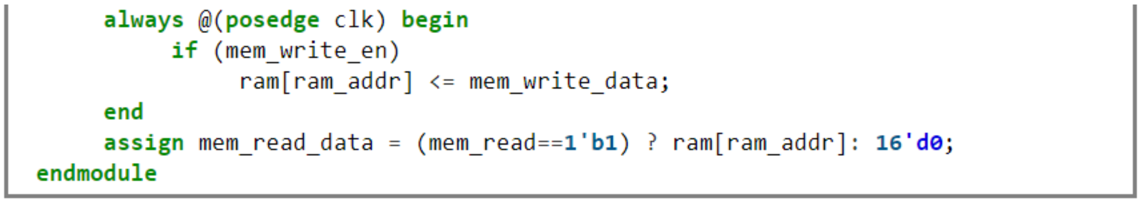


After completing the design for the MIPS processor, it is easy to write Verilog code for the MIPS processor. The Verilog code for the whole design of the MIPS processor as follows:
Verilog code for ALU unit

Verilog code for register file
Verilog code for instruction memory
Verilog code for data memory:

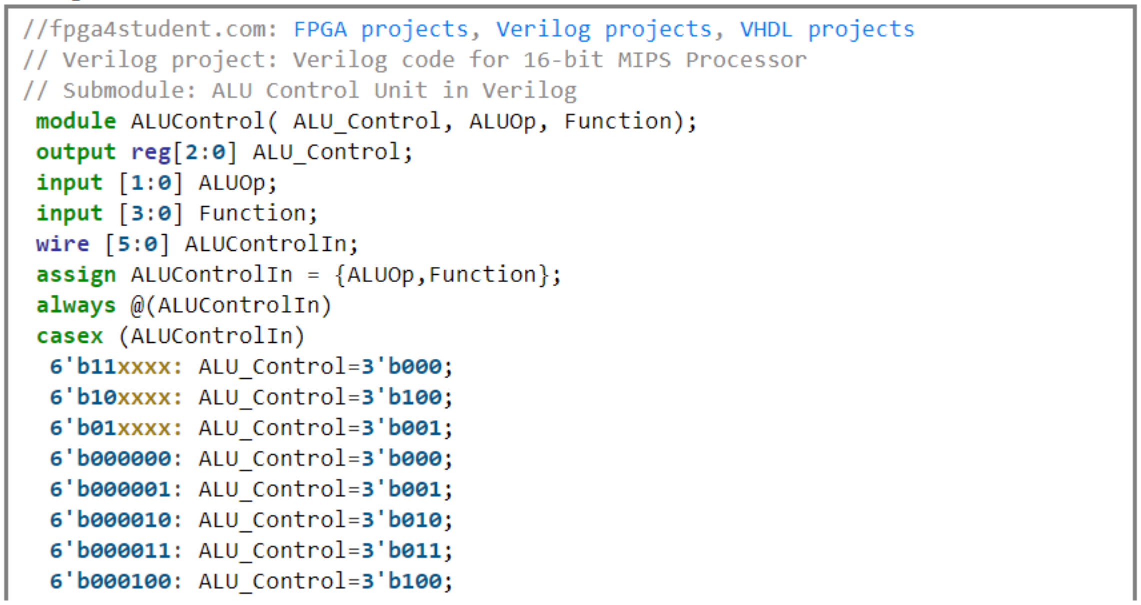


```
//fpga4student.com: FPGA projects, Verilog projects, VHDL projects
// Verilog project: Verilog code for 16-bit MIPS Processor
// Submodule: Data memory in Verilog
 module data_memory
 (
      input                         clk,
      // address input, shared by read and write port
      input     [15:0]               mem_access_addr,
      // write port
      input     [15:0]               mem_write_data,
      input                         mem_write_en,
      input mem_read,
      // read port
      output     [15:0]               mem_read_data
 );
      integer i;
      reg [15:0] ram [255:0];
      wire [7 : 0] ram_addr = mem_access_addr[8 : 1];
      initial begin
           for(i=0;i<256;i=i+1)
                ram[i] <= 16'd0;
      end
      always @(posedge clk) begin
```

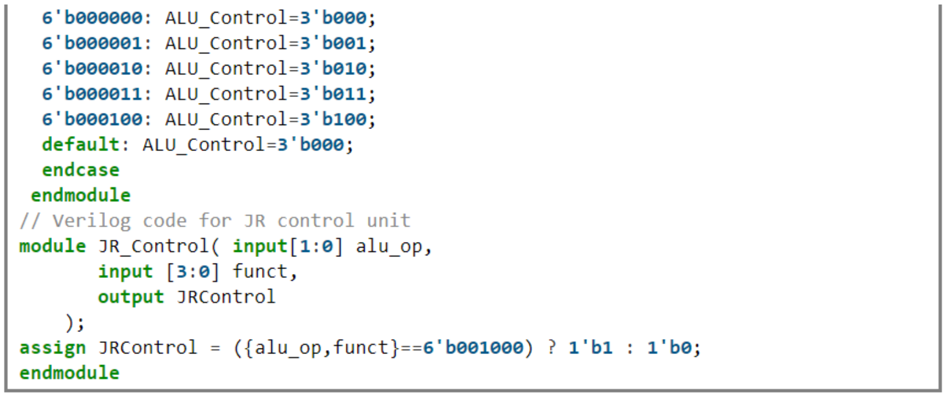


```
      always @(posedge clk) begin
           if (mem_write_en)
                ram[ram_addr] <= mem_write_data;
      end
      assign mem_read_data = (mem_read==1'b1) ? ram[ram_addr]: 16'd0;
 endmodule
```

Verilog code for ALU Control unit:

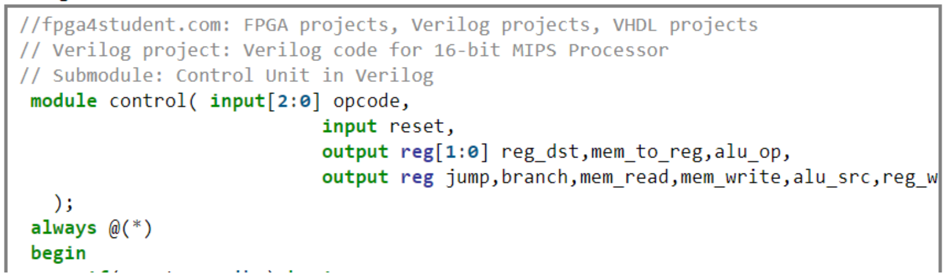


```
//fpga4student.com: FPGA projects, Verilog projects, VHDL projects
// Verilog project: Verilog code for 16-bit MIPS Processor
// Submodule: ALU Control Unit in Verilog
 module ALUControl( ALU_Control, ALUOp, Function);
 output reg[2:0] ALU_Control;
 input [1:0] ALUOp;
 input [3:0] Function;
 wire [5:0] ALUControlIn;
 assign ALUControlIn = {ALUOp,Function};
 always @(ALUControlIn)
 casex (ALUControlIn)
  6'b11xxxx: ALU_Control=3'b000;
  6'b10xxxx: ALU_Control=3'b100;
  6'b01xxxx: ALU_Control=3'b001;
  6'b000000: ALU_Control=3'b000;
  6'b000001: ALU_Control=3'b001;
  6'b000010: ALU_Control=3'b010;
  6'b000011: ALU_Control=3'b011;
  6'b000100: ALU_Control=3'b100;
```

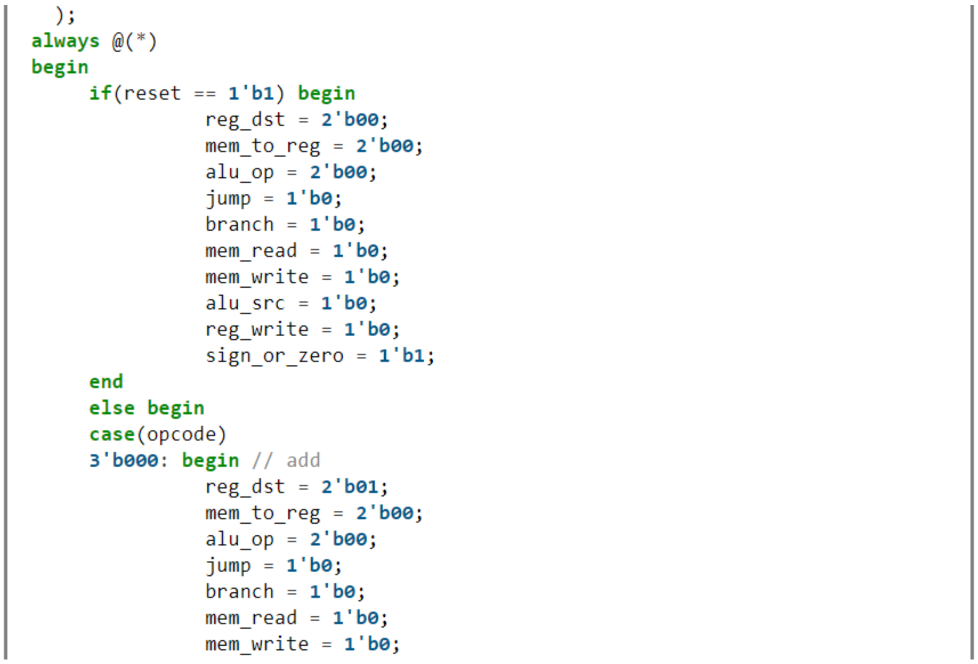


```
  6'b000000: ALU_Control=3'b000;
  6'b000001: ALU_Control=3'b001;
  6'b000010: ALU_Control=3'b010;
  6'b000011: ALU_Control=3'b011;
  6'b000100: ALU_Control=3'b100;
  default: ALU_Control=3'b000;
  endcase
 endmodule
// Verilog code for JR control unit
module JR_Control( input[1:0] alu_op,
       input [3:0] funct,
       output JRControl
    );
assign JRControl = ({alu_op,funct}==6'b001000) ? 1'b1 : 1'b0;
endmodule
```

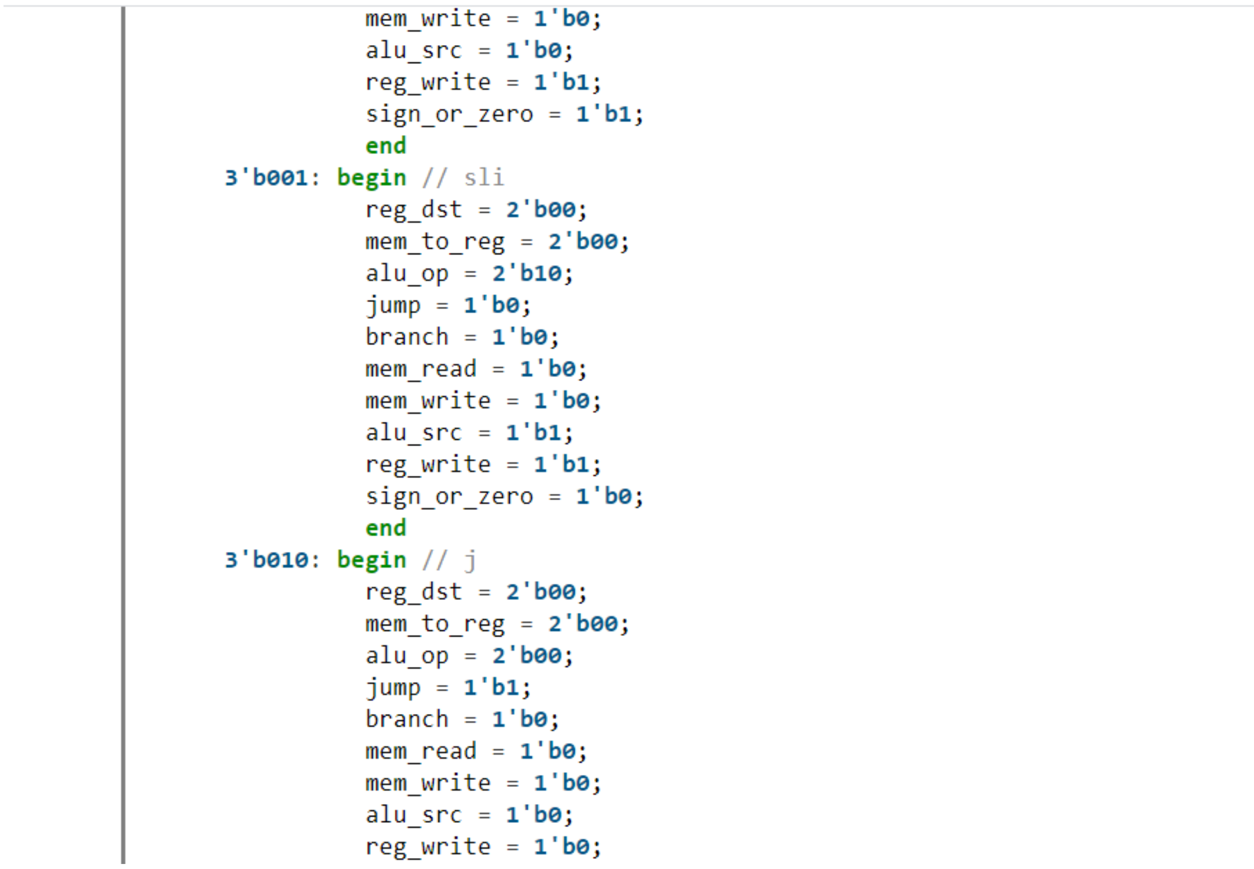
Verilog code for control unit:



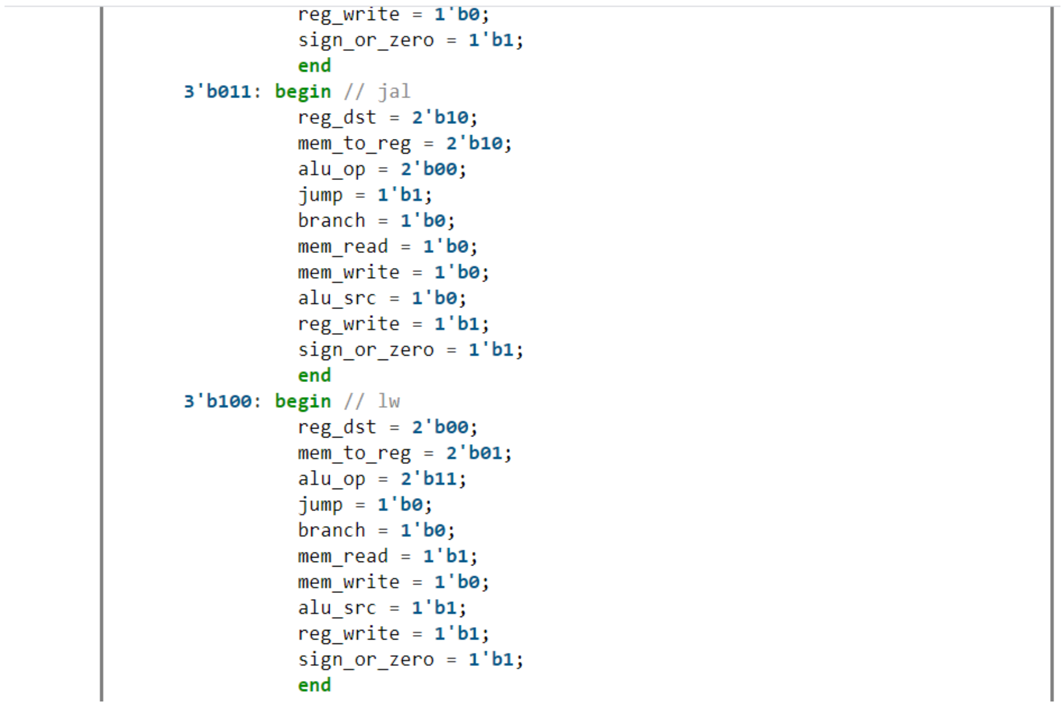
```
//fpga4student.com: FPGA projects, Verilog projects, VHDL projects
// Verilog project: Verilog code for 16-bit MIPS Processor
// Submodule: Control Unit in Verilog
 module control( input[2:0] opcode,
                           input reset,
                           output reg[1:0] reg_dst,mem_to_reg,alu_op,
                           output reg jump,branch,mem_read,mem_write,alu_src,reg_w
   );
 always @(*)
 begin
```



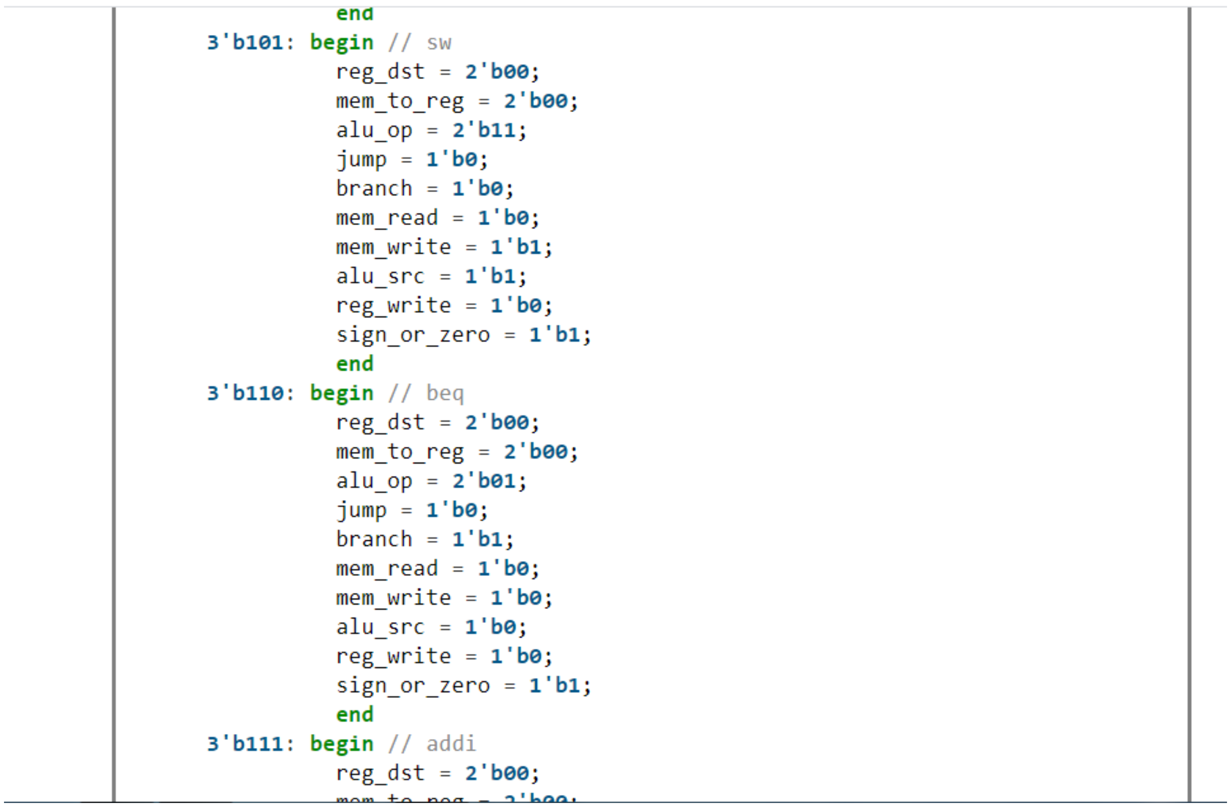
```
   );
 always @(*)
 begin
      if(reset == 1'b1) begin
                reg_dst = 2'b00;
                mem_to_reg = 2'b00;
                alu_op = 2'b00;
                jump = 1'b0;
                branch = 1'b0;
                mem_read = 1'b0;
                mem_write = 1'b0;
                alu_src = 1'b0;
                reg_write = 1'b0;
                sign_or_zero = 1'b1;
      end
      else begin
      case(opcode)
      3'b000: begin // add
                reg_dst = 2'b01;
                mem_to_reg = 2'b00;
                alu_op = 2'b00;
                jump = 1'b0;
                branch = 1'b0;
                mem_read = 1'b0;
                mem_write = 1'b0;
```



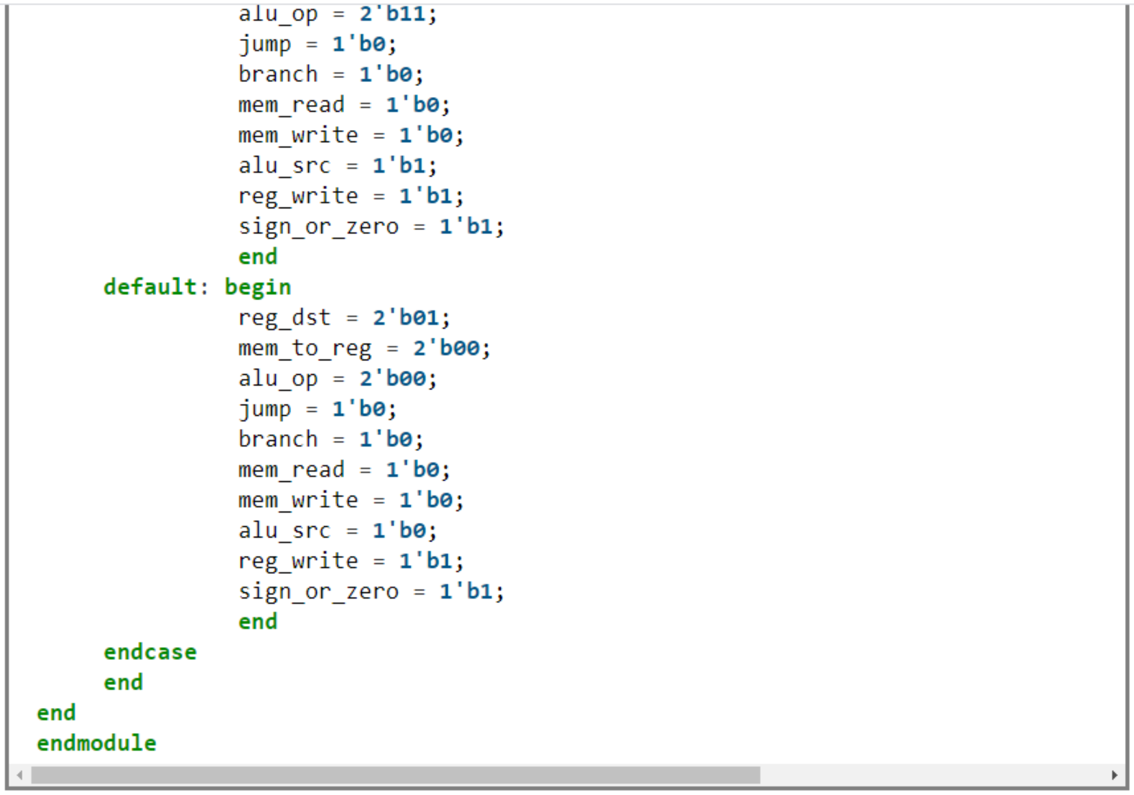
```
                    mem_write = 1'b0;
                    alu_src = 1'b0;
                    reg_write = 1'b1;
                    sign_or_zero = 1'b1;
                    end
          3'b001: begin // sli
                    reg_dst = 2'b00;
                    mem_to_reg = 2'b00;
                    alu_op = 2'b10;
                    jump = 1'b0;
                    branch = 1'b0;
                    mem_read = 1'b0;
                    mem_write = 1'b0;
                    alu_src = 1'b1;
                    reg_write = 1'b1;
                    sign_or_zero = 1'b0;
                    end
          3'b010: begin // j
                    reg_dst = 2'b00;
                    mem_to_reg = 2'b00;
                    alu_op = 2'b00;
                    jump = 1'b1;
                    branch = 1'b0;
                    mem_read = 1'b0;
                    mem_write = 1'b0;
                    alu_src = 1'b0;
                    reg_write = 1'b0;
```



```
          reg_write = 1'b0;
          sign_or_zero = 1'b1;
          end
3'b011: begin // jal
          reg_dst = 2'b10;
          mem_to_reg = 2'b10;
          alu_op = 2'b00;
          jump = 1'b1;
          branch = 1'b0;
          mem_read = 1'b0;
          mem_write = 1'b0;
          alu_src = 1'b0;
          reg_write = 1'b1;
          sign_or_zero = 1'b1;
          end
3'b100: begin // lw
          reg_dst = 2'b00;
          mem_to_reg = 2'b01;
          alu_op = 2'b11;
          jump = 1'b0;
          branch = 1'b0;
          mem_read = 1'b1;
          mem_write = 1'b0;
          alu_src = 1'b1;
          reg_write = 1'b1;
          sign_or_zero = 1'b1;
          end
```

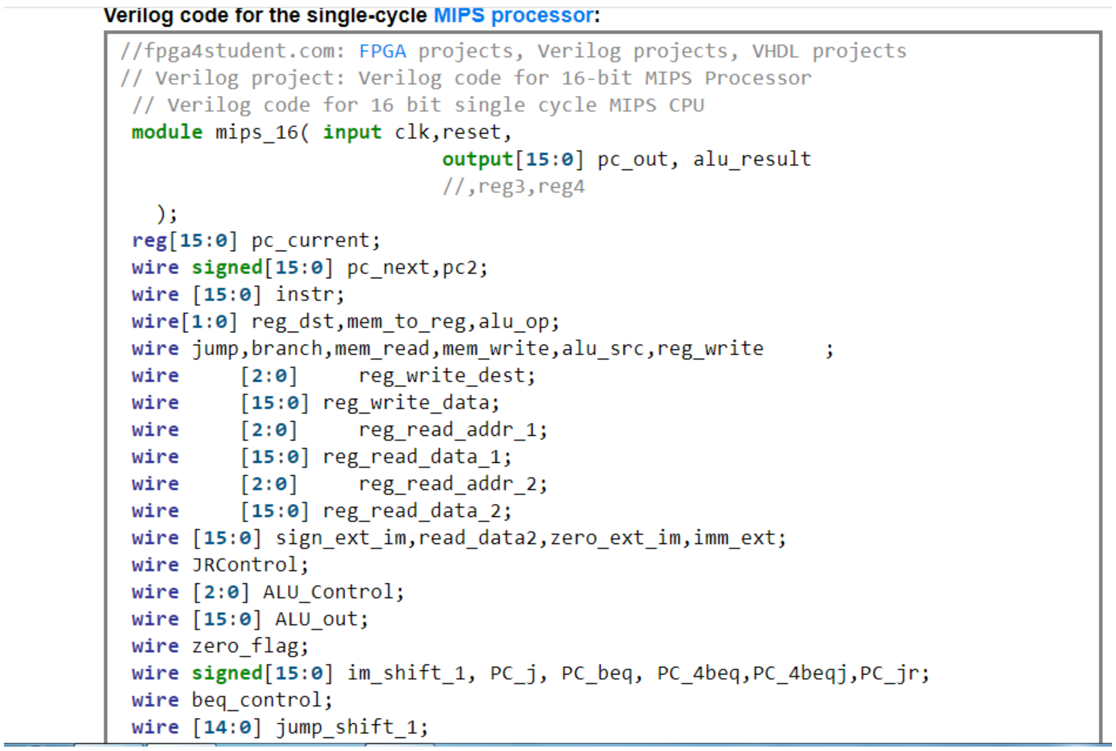


```
          end
3'b101: begin // sw
          reg_dst = 2'b00;
          mem_to_reg = 2'b00;
          alu_op = 2'b11;
          jump = 1'b0;
          branch = 1'b0;
          mem_read = 1'b0;
          mem_write = 1'b1;
          alu_src = 1'b1;
          reg_write = 1'b0;
          sign_or_zero = 1'b1;
          end
3'b110: begin // beq
          reg_dst = 2'b00;
          mem_to_reg = 2'b00;
          alu_op = 2'b01;
          jump = 1'b0;
          branch = 1'b1;
          mem_read = 1'b0;
          mem_write = 1'b0;
          alu_src = 1'b0;
          reg_write = 1'b0;
          sign_or_zero = 1'b1;
          end
3'b111: begin // addi
          reg_dst = 2'b00;
          mem_to_reg = 2'b00;
```



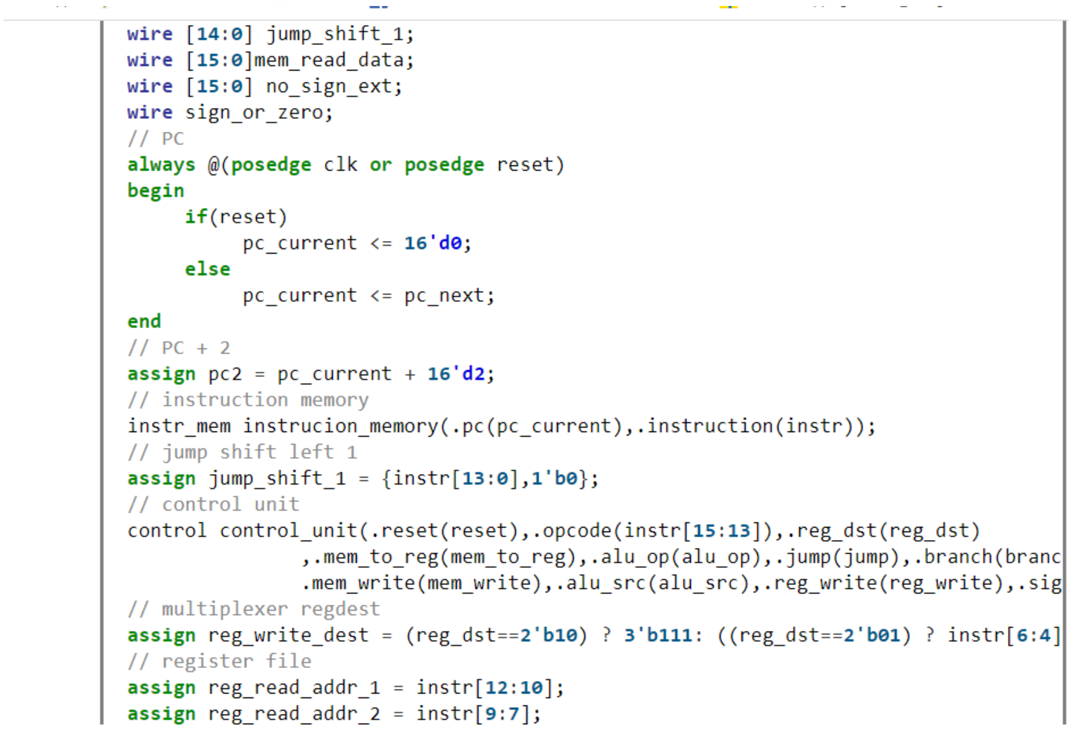
```
                    alu_op = 2'b11;
                    jump = 1'b0;
                    branch = 1'b0;
                    mem_read = 1'b0;
                    mem_write = 1'b0;
                    alu_src = 1'b1;
                    reg_write = 1'b1;
                    sign_or_zero = 1'b1;
                    end
          default: begin
                    reg_dst = 2'b01;
                    mem_to_reg = 2'b00;
                    alu_op = 2'b00;
                    jump = 1'b0;
                    branch = 1'b0;
                    mem_read = 1'b0;
                    mem_write = 1'b0;
                    alu_src = 1'b0;
                    reg_write = 1'b1;
                    sign_or_zero = 1'b1;
                    end
          endcase
          end
     end
     endmodule
```

Verilog code for the single-cycle MIPS processor:

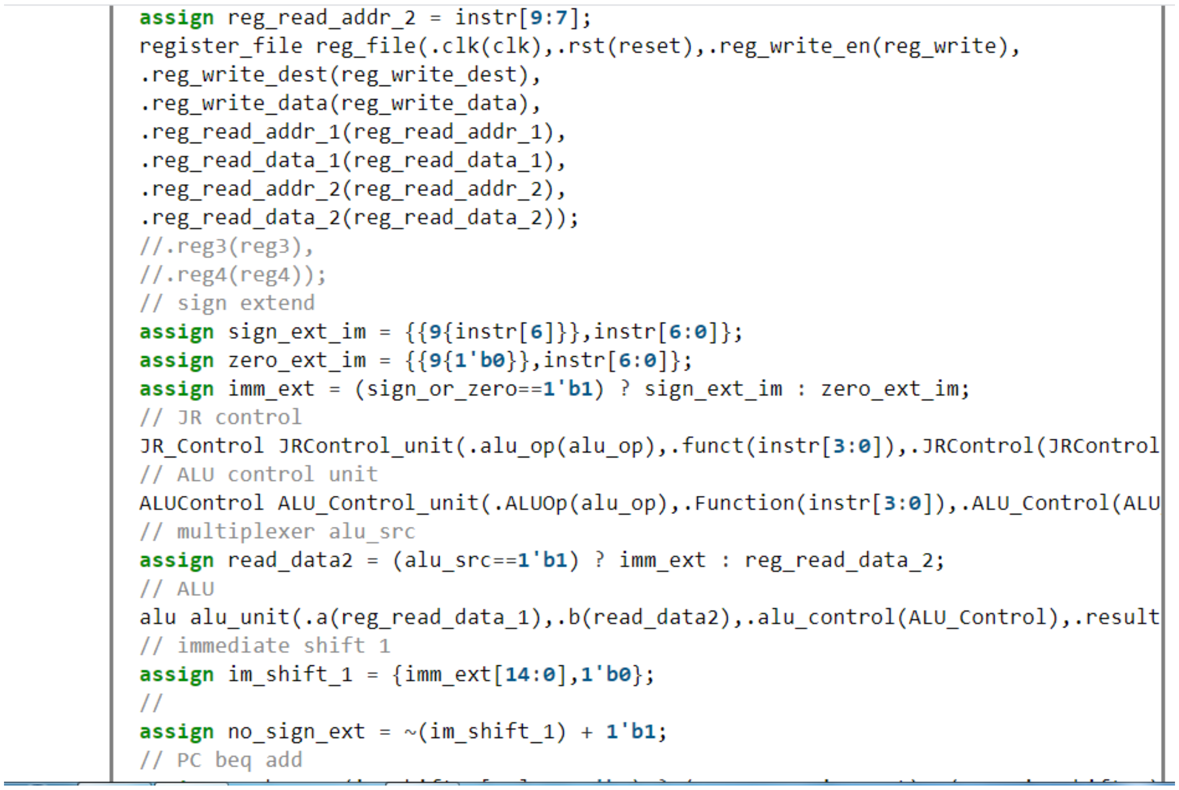


Verilog code for the single-cycle MIPS processor:

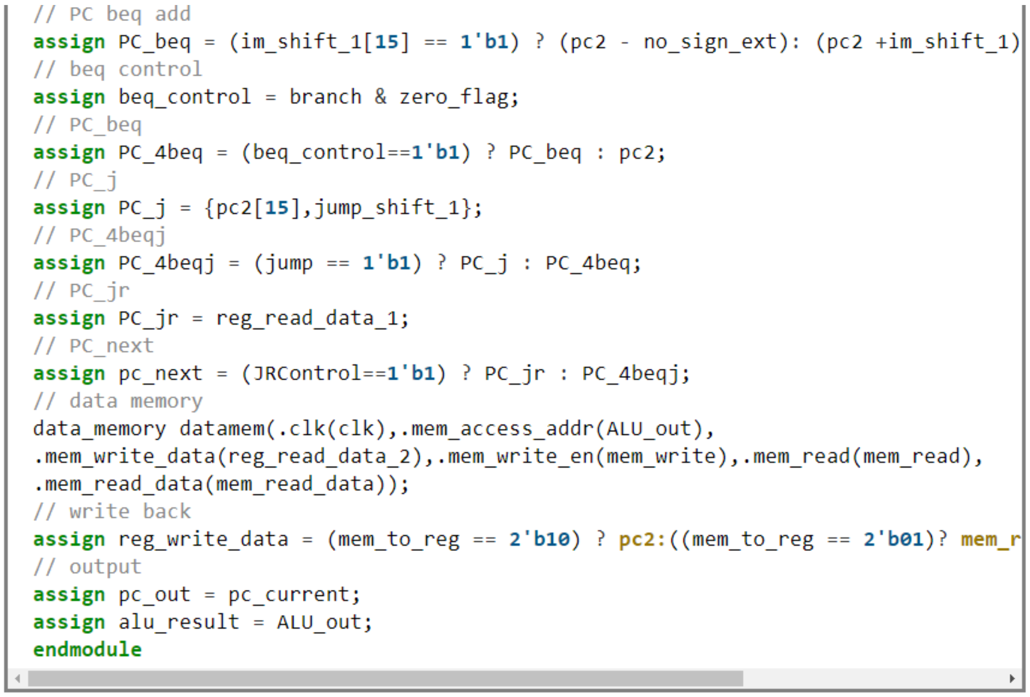
```
//fpga4student.com: FPGA projects, Verilog projects, VHDL projects
// Verilog project: Verilog code for 16-bit MIPS Processor
 // Verilog code for 16 bit single cycle MIPS CPU
 module mips_16( input clk,reset,
                          output[15:0] pc_out, alu_result
                          //,reg3,reg4
   );
 reg[15:0] pc_current;
 wire signed[15:0] pc_next,pc2;
 wire [15:0] instr;
 wire[1:0] reg_dst,mem_to_reg,alu_op;
 wire jump,branch,mem_read,mem_write,alu_src,reg_write     ;
 wire     [2:0]     reg_write_dest;
 wire     [15:0] reg_write_data;
 wire     [2:0]     reg_read_addr_1;
 wire     [15:0] reg_read_data_1;
 wire     [2:0]     reg_read_addr_2;
 wire     [15:0] reg_read_data_2;
 wire [15:0] sign_ext_im,read_data2,zero_ext_im,imm_ext;
 wire JRControl;
 wire [2:0] ALU_Control;
 wire [15:0] ALU_out;
 wire zero_flag;
 wire signed[15:0] im_shift_1, PC_j, PC_beq, PC_4beq,PC_4beqj,PC_jr;
 wire beq_control;
 wire [14:0] jump_shift_1;
```



```
wire [14:0] jump_shift_1;
wire [15:0]mem_read_data;
wire [15:0] no_sign_ext;
wire sign_or_zero;
// PC
always @(posedge clk or posedge reset)
begin
     if(reset)
          pc_current <= 16'd0;
     else
          pc_current <= pc_next;
end
// PC + 2
assign pc2 = pc_current + 16'd2;
// instruction memory
instr_mem instrucion_memory(.pc(pc_current),.instruction(instr));
// jump shift left 1
assign jump_shift_1 = {instr[13:0],1'b0};
// control unit
control control_unit(.reset(reset),.opcode(instr[15:13]),.reg_dst(reg_dst)
               ,.mem_to_reg(mem_to_reg),.alu_op(alu_op),.jump(jump),.branch(branc
               .mem_write(mem_write),.alu_src(alu_src),.reg_write(reg_write),.sig
// multiplexer regdest
assign reg_write_dest = (reg_dst==2'b10) ? 3'b111: ((reg_dst==2'b01) ? instr[6:4]
// register file
assign reg_read_addr_1 = instr[12:10];
assign reg_read_addr_2 = instr[9:7];
```

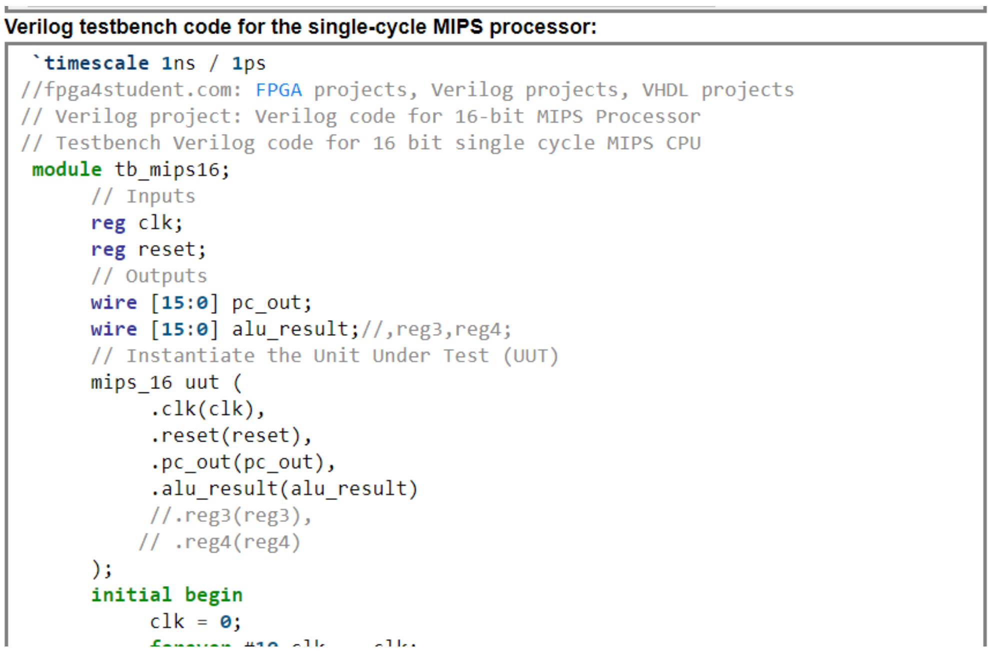


```
assign reg_read_addr_2 = instr[9:7];
register_file reg_file(.clk(clk),.rst(reset),.reg_write_en(reg_write),
.reg_write_dest(reg_write_dest),
.reg_write_data(reg_write_data),
.reg_read_addr_1(reg_read_addr_1),
.reg_read_data_1(reg_read_data_1),
.reg_read_addr_2(reg_read_addr_2),
.reg_read_data_2(reg_read_data_2));
//.reg3(reg3),
//.reg4(reg4));
// sign extend
assign sign_ext_im = {{9{instr[6]}},instr[6:0]};
assign zero_ext_im = {{9{1'b0}},instr[6:0]};
assign imm_ext = (sign_or_zero==1'b1) ? sign_ext_im : zero_ext_im;
// JR control
JR_Control JRControl_unit(.alu_op(alu_op),.funct(instr[3:0]),.JRControl(JRControl
// ALU control unit
ALUControl ALU_Control_unit(.ALUOp(alu_op),.Function(instr[3:0]),.ALU_Control(ALU
// multiplexer alu_src
assign read_data2 = (alu_src==1'b1) ? imm_ext : reg_read_data_2;
// ALU
alu alu_unit(.a(reg_read_data_1),.b(read_data2),.alu_control(ALU_Control),.result
// immediate shift 1
assign im_shift_1 = {imm_ext[14:0],1'b0};
//
assign no_sign_ext = ~(im_shift_1) + 1'b1;
// PC beq add
```



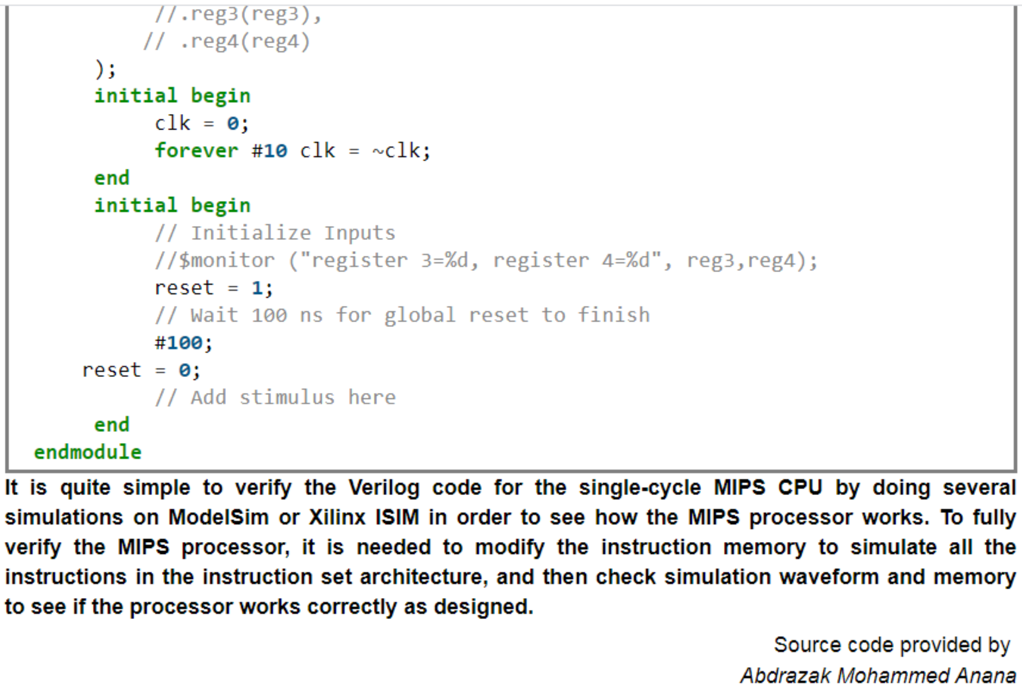
```
// PC beq add
assign PC_beq = (im_shift_1[15] == 1'b1) ? (pc2 - no_sign_ext): (pc2 +im_shift_1)
// beq control
assign beq_control = branch & zero_flag;
// PC_beq
assign PC_4beq = (beq_control==1'b1) ? PC_beq : pc2;
// PC_j
assign PC_j = {pc2[15],jump_shift_1};
// PC_4beqj
assign PC_4beqj = (jump == 1'b1) ? PC_j : PC_4beq;
// PC_jr
assign PC_jr = reg_read_data_1;
// PC_next
assign pc_next = (JRControl==1'b1) ? PC_jr : PC_4beqj;
// data memory
data_memory datamem(.clk(clk),.mem_access_addr(ALU_out),
.mem_write_data(reg_read_data_2),.mem_write_en(mem_write),.mem_read(mem_read),
.mem_read_data(mem_read_data));
// write back
assign reg_write_data = (mem_to_reg == 2'b10) ? pc2:((mem_to_reg == 2'b01)? mem_r
// output
assign pc_out = pc_current;
assign alu_result = ALU_out;
endmodule
```

Verilog testbench code for the single-cycle MIPS processor:



Verilog testbench code for the single-cycle MIPS processor:

```
 `timescale 1ns / 1ps
//fpga4student.com: FPGA projects, Verilog projects, VHDL projects
// Verilog project: Verilog code for 16-bit MIPS Processor
// Testbench Verilog code for 16 bit single cycle MIPS CPU
 module tb_mips16;
      // Inputs
      reg clk;
      reg reset;
      // Outputs
      wire [15:0] pc_out;
      wire [15:0] alu_result;//,reg3,reg4;
      // Instantiate the Unit Under Test (UUT)
      mips_16 uut (
           .clk(clk),
           .reset(reset),
           .pc_out(pc_out),
           .alu_result(alu_result)
           //.reg3(reg3),
          // .reg4(reg4)
      );
      initial begin
           clk = 0;
           forever #10 clk = ~clk;
```



```
            //.reg3(reg3),
          // .reg4(reg4)
        );
        initial begin
             clk = 0;
             forever #10 clk = ~clk;
        end
        initial begin
             // Initialize Inputs
             //$monitor ("register 3=%d, register 4=%d", reg3,reg4);
             reset = 1;
             // Wait 100 ns for global reset to finish
             #100;
       reset = 0;
             // Add stimulus here
        end
   endmodule
```

It is quite simple to verify the Verilog code for the single-cycle MIPS CPU by doing several simulations on ModelSim or Xilinx ISIM in order to see how the MIPS processor works. To fully verify the MIPS processor, it is needed to modify the instruction memory to simulate all the instructions in the instruction set architecture, and then check simulation waveform and memory to see if the processor works correctly as designed.

Source code provided by
Abdrazak Mohammed Anana