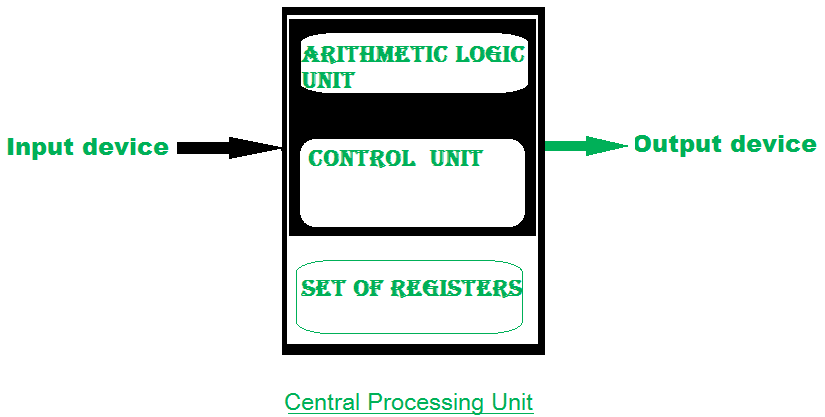
**Module 4**

**The Processor**

# Introduction of ALU and Data Path

Representing and storing numbers were the basic operation of the computers of earlier times. The real goal came when computation, manipulating numbers like adding, multiplying came into the picture. These operations are handled by the computer’s **arithmetic logic unit (ALU)**. The ALU is the mathematical brain of a computer. The first ALU was INTEL 74181 implemented as a 7400 series is a TTL integrated circuit that was released in 1970.

The **ALU** is a digital circuit that provides arithmetic and logic operations. It is the fundamental building block of the central processing unit of a computer. A modern CPU has a very powerful ALU and it is complex in design. In addition to ALU modern CPU contains a control unit and a set of registers. Most of the operations are performed by one or more ALU’s, which load data from the input register. Registers are a small amount of storage available to the CPU. These registers can be accessed very fast. The control unit tells ALU what operation to perform on the available data. After calculation/manipulation, the ALU stores the output in an output register.



The CPU can be divided into two sections: the data section and the control section. The DATA section is also known as the data path.

As mentioned, the CPU can be divided into a **data section** and a **control section**. The **data section**, which is also called the datapath, contains the registers and the ALU. The datapath is capable of performing certain operations on data items.

The **control section** is basically the control unit, which issues control signals to the datapath. Internal to the CPU, data move from one register to another and between ALU and registers. Internal data movements are performed via **local buses, which may carry data, instructions, and addresses**. Externally, data move from registers to memory and I/O devices, often by means of a system bus. Internal data movement among registers and between the ALU and registers may be carried out using different organizations including **one-bus, two-bus, or three-bus organizations.**

**BUS:** In early computers “BUS” were parallel electrical wires with multiple hardware connections. Therefore, a bus is a communication system that transfers data between components inside a computer, or between computers. It includes hardware components like wires, optical fibres, etc and software, including communication protocols. The Registers, ALU, and the interconnecting BUS are collectively referred to as data paths.

Types of the bus are: 

1. **Address bus:** The buses which are used to carry address.
2. **Data bus:** The buses which are used to carry data.
3. **Control bus:** If the bus is carrying control signals.
4. **Power bus:** If it is carrying clock pulse, power signals it is known as a power bus, and so on.
5. **Program Counter –**   
   A program counter (PC) is a CPU register in the computer processor which has the address of the next instruction to be executed from memory. As each instruction gets fetched, the program counter increases its stored value by 1. It is a digital counter needed for faster execution of tasks as well as for tracking the current execution point.

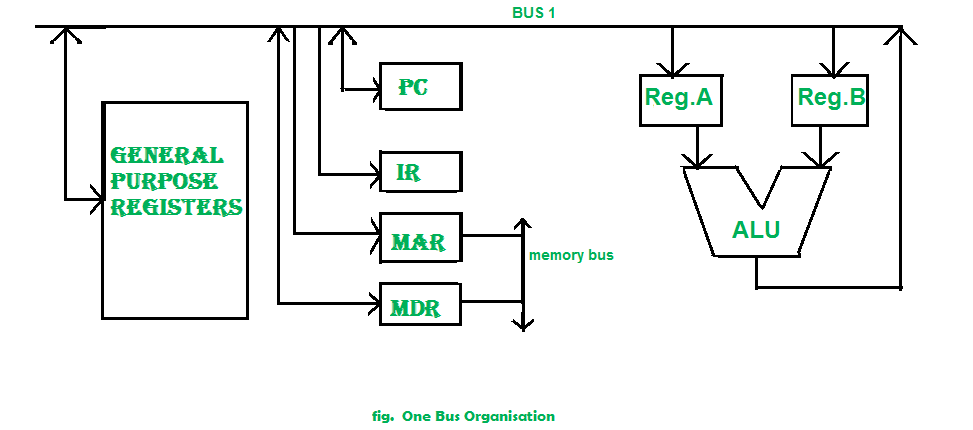
1. **Instruction Register –**   
   In computing, an instruction registers (IR) is the part of a CPU’s control unit that holds the instruction currently being executed or decoded. The instruction register specifically holds the instruction and provides it to the instruction decoder circuit.

1. **Memory Address Register –**   
   The Memory Address Register (MAR) is the CPU register that either stores the memory address from which data will be fetched from the CPU, or the address to which data will be sent and stored. It is a temporary storage component in the CPU (central processing unit) that temporarily stores the address (location) of the data sent by the memory unit until the instruction for the particular data is executed.

1. **Memory Data Register –**   
   The memory data register (MDR) is the register in a computer’s processor, or central processing unit, CPU, that stores the data being transferred to and from the immediate access storage. Memory data register (MDR) is also known as memory buffer register (MBR).

1. **General Purpose Register –**   
   General-purpose registers are used to store temporary data within the microprocessor. It is a multipurpose register. They can be used either by a programmer or by a user.

**One Bus organization –**



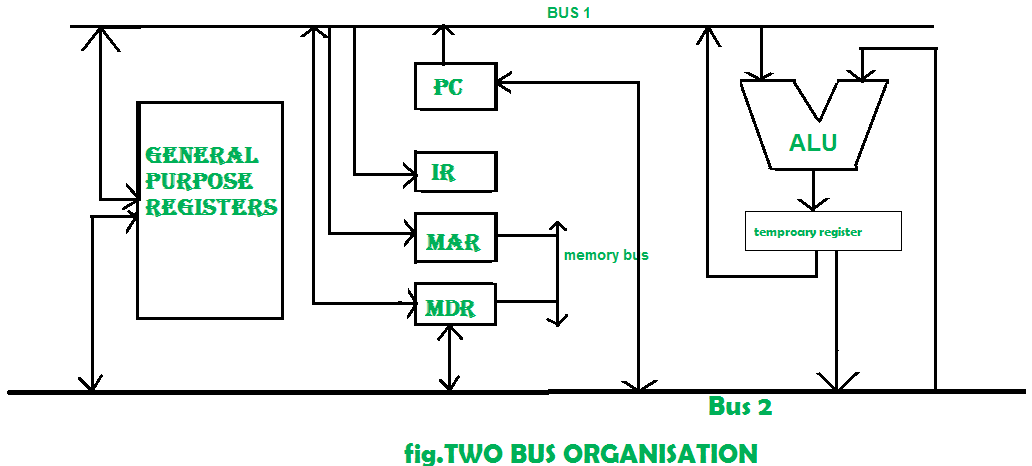
In one bus organization, a single bus is used for multiple purposes. A set of general-purpose registers, program counters, instruction registers, memory address registers (MAR), memory data registers (MDR) are connected with the single bus. Memory read/write can be done with MAR and MDR. The program counterpoints to the memory location from where the next instruction is to be fetched. Instruction register is that very register will hold the copy of the current instruction. In the case of one bus organization, at a time only one operand can be read from the bus.

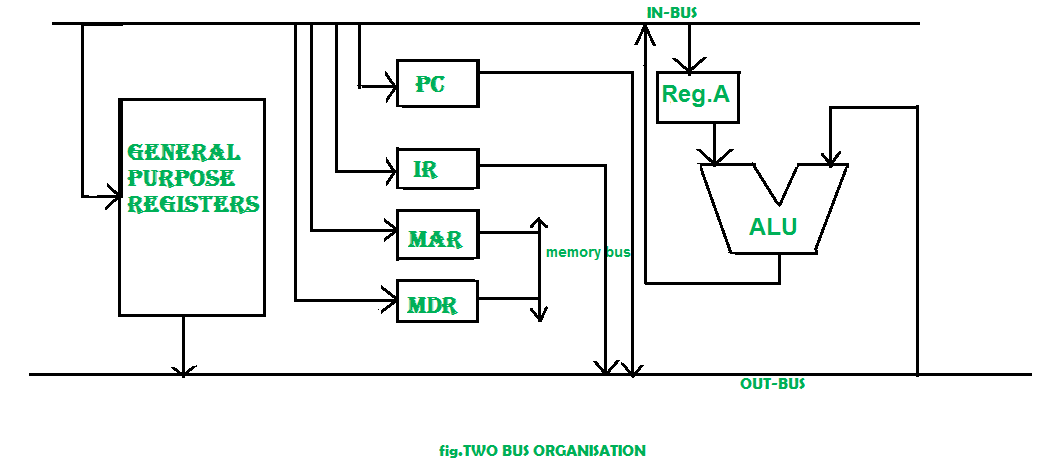
As a result, if the requirement is to read two operands for the operation then the read operation needs to be carried twice. So that’s why it is making the process a little longer. One of the advantages of one bus organization is that it is one of the simplest and also this is very cheap to implement. At the same time a disadvantage lies that it has only one bus and this “one bus” is accessed by all general-purpose registers, program counter, instruction register, MAR, MDR making each and every operation sequential. No one recommends this architecture nowadays

**Two Bus organizations:**

To overcome the disadvantage of one bus organization another architecture was developed known as two bus organization. In two bus organizations, there are two buses. The general-purpose register can read/write from both the buses. In this case, two operands can be fetched at the same time because of the two buses. One bus fetch operand for ALU and another bus fetch for register. The situation arises when both buses are busy fetching operands, the output can be stored in a temporary register and when the buses are free, the particular output can be dumped on the buses.

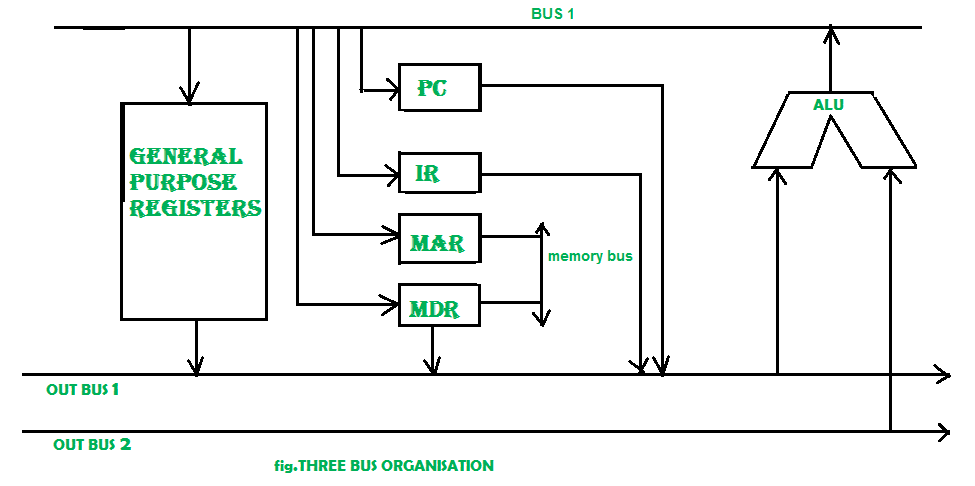
There are two versions of two bus organizations, i.e., in-bus and out-bus. From in-bus, the general-purpose register can read data and to the out bus, the general-purpose registers can write data. Here buses get dedicated.





**Three Bus organization –**

In three bus organizations we have three buses, OUT bus1, OUT bus2, and an IN bus. From the out buses, we can get the operand which can come from the general-purpose register and evaluated in ALU and the output is dropped on In Bus so it can be sent to respective registers. This implementation is a bit complex but faster in nature because in parallel two operands can flow into ALU and out of ALU. It was developed to overcome the “busy waiting” problem of two bus organizations. In this structure after execution, the output can be dropped on the bus without waiting because of the presence of an extra bus. The structure is given below in the figure.



The main **advantages** of multiple bus organizations over the single bus are as given below. 

1. Increase in size of the registers.
2. Reduction in the number of cycles for execution.
3. Increases the speed of execution or we can say faster execution.

**Introduction: LOGIC DESIGN CONVENTIONS:**

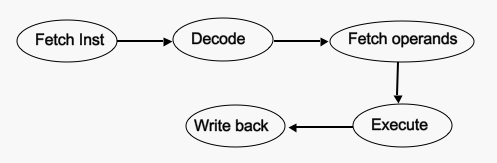
This section contains an explanation of the principles and techniques used in implementing a processor. It is followed by a section that builds up a datapath and constructs a simple version of a processor sufficient to implement an instruction set like MIPS (**Million instructions per second**).

**A Basic MIPS Implementation**

A subset of the MIPS instruction set contains:

* The memory-reference instructions load word (lw) and store word (sw)
* The arithmetic-logical instructions add, sub, AND, OR, and slt (set less than)
* The instructions branch equal (beq) and jump (j)

**What to be done for each instruction**



Generic implementation

* use the program counter (PC) to supply the instruction address and fetch the instruction from memory (and update the PC)
* decode the instruction (and read registers)
* execute the instruction

An abstract view of the implementation of the MIPS subset showing the major functional units and the major connections between them:

**Building a Datapath:**

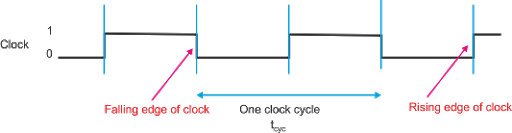
**A Datapath is a set of functional units that carry out**[**data processing**](https://www.computerhope.com/jargon/d/dataproc.htm)**operations.**

**The Datapath elements in the MIPS implementation** consist of two different types of logic elements: elements that operate on data values and elements that contain state.

* *Combinational* - The elements that operate on data values are all combinational, which means their outputs depend only on the current inputs Ex: ALU, Adder, MUX.
* *Sequential* – Their outputs depend only on the current inputs and contents of the internal state Ex: Data memory, register

**Clocking Methodology**

A clocking methodology defines when signals can be read and when they can be written. The clock cycle/period is divided into two portions high clock and low clock .



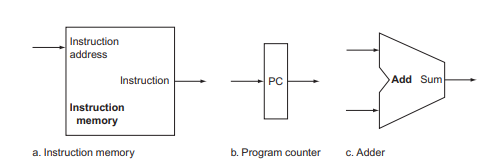
An edge-triggered clocking methodology means that any values stored in a sequential logic element are updated only on a clock edge, which is a quick transition from low to high or vice versa (see Figure4.3).

In the MIPS implementation, the **Data path elements include:**

 1. Memory unit - to store the instructions of a program and supply instructions given an address.

2. program counter (PC) - a register that holds the address of the instruction which is executed next

3. adder - to increment the PC to the address of the next instruction.

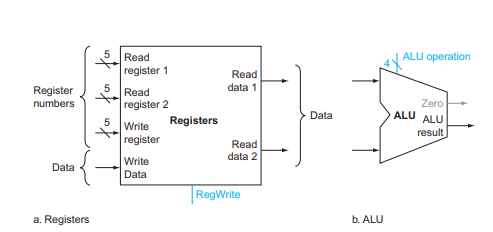


**The interconnection of these simple components to form a basic data path**. To execute any instruction, we must start by fetching the instruction from memory. To prepare for executing the next instruction, we must also increment the program counter so that it points at the next instruction, 4 bytes later.

**1,R-format instructions**: They all read two registers, perform an ALU operation on the contents of the registers, and write the result to a register. We call these instructions either R-type instructions or arithmetic-logical instructions (since they perform arithmetic or logical operations). This instruction class includes add, sub, AND, OR.

**Example: add t1, t2, t3**

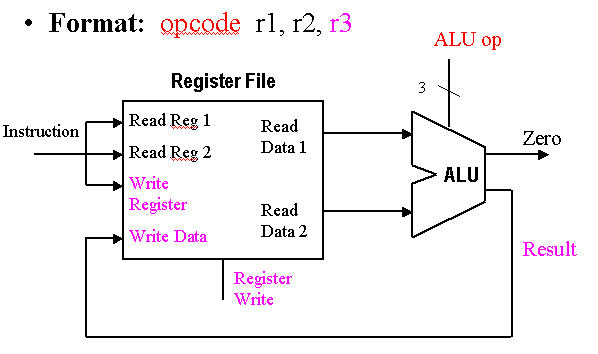
The processor’s 32 general-purpose registers are stored in a structure called a register file. ***A register file*** is a collection of registers in which any register can be read or written by specifying the number of the register in the file. The register file contains the register state of the computer. In addition, we will need an ALU to operate on the values read from the registers. We need a total of four inputs (three for register numbers and one for data) and two outputs (both for data).



**The two elements needed to implement R-format ALU operations are the register file and the ALU**

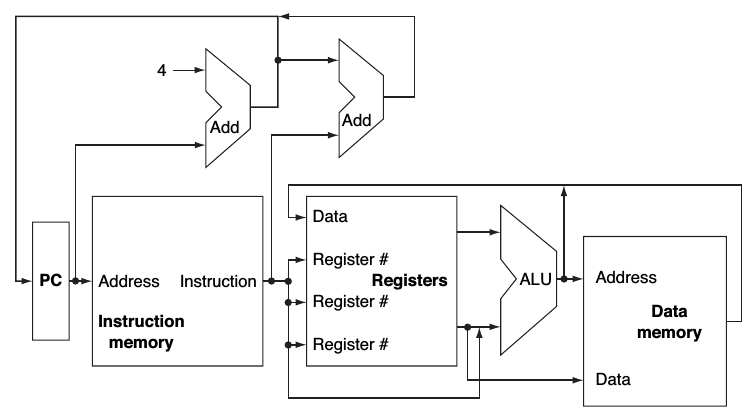
R-format instructions have three register operands, so we will need

to read two data words from the register file and write one data word into the register file for each instruction. To write a data word, we will need two inputs: one to specify the register number to be written and one to supply the data to be written into the register.



Implementation of the data path for R-format instructions is fairly straightforward - the register file and the ALU are all that is required. The ALU accepts its input from the Data Read ports of the register file, and the register file is written to by the ALU Result output of the ALU, in combination with the Reg Write signal.

**An abstract view of the implementation of the MIPS:**



**An Overview of the Implementation**

For every instruction, the first two steps are identical:

1. Send the program counter (PC) to the memory that contains the code and fetch the instruction from that memory.

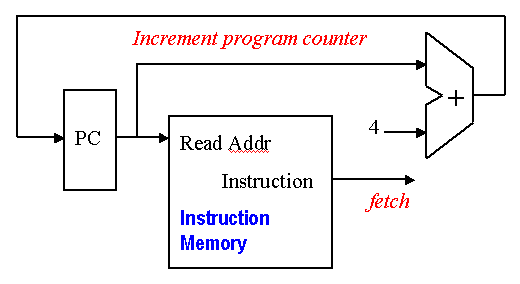
2. Read one or two registers, using fields of the instruction to select the registers to read. For the load word instruction, we need to read only one register, but most other instructions require reading two registers.

After these two steps, the actions required to complete the instruction depend on the instruction class (memory-reference, arithmetic-logical, and branches). For example, all instruction classes, except jump, use the arithmetic-logical unit (ALU) after reading the registers.

The memory-reference instructions use the ALU for an address calculation, the arithmetic-logical instructions for the operation execution, and branches for comparison. A memory-reference instruction will need to access the memory either to read data for a load or write data for a store. An arithmetic-logical or load instruction must write the data from the ALU or memory back into a register. Lastly, for a branch instruction, we may need to change the next instruction address based on the comparison; otherwise, the PC should be incremented by 4 to get the address of the next instruction**. A value of 4 is used since memory is byte addressable and each instruction is 4 bytes long.**

### 

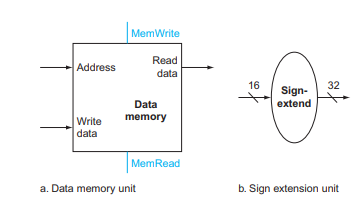
**Fetching Instructions**



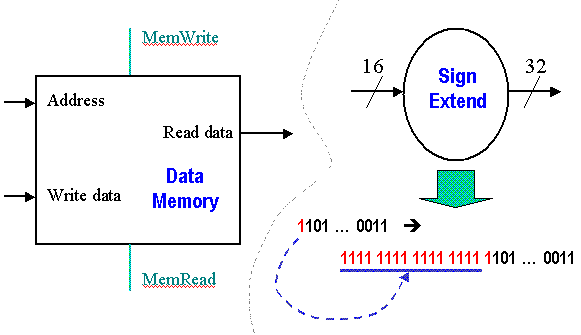
**A portion of the data path used for fetching instructions and incrementing the program counter**.

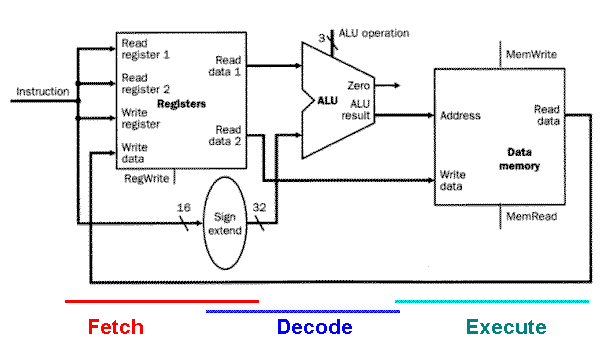
**2, I format intructions-Load/Store Datapath**

The load/store datapath uses instructions such as **lw t1, offset(t2) or sw t1, offset(t2)**. These instructions compute a memory address by adding the base register, which is t2, to the 16-bit signed offset field contained in the instruction. If the instruction is a store, the value to be stored must also be read from the register file where it resides in t1. If the instruction is a load, the value read from memory must be written into the register file in the specified register, which is t1. Thus, we will need both the register file and the ALU. In addition, we will need a unit to sign-extend the 16-bit offset field in the instruction to a 32-bit signed value, and a data memory unit to read from or write to. The data memory must be written on store instructions; hence, data memory has read and write control signals, an address input, and an input for the data to be written into memory.



**The two units needed to implement loads and stores, in addition to the register fi le and ALU are the data memory unit and the sign extension unit.**





The load/store data path is illustrated in Figure 4.8, and performs the following actions in the order given:

1. *Register Access* takes input from the register file, to implement the instruction, data, or address *fetch* step of the fetch-decode-execute cycle.
2. *Memory Address Calculation* decodes the base address and offset, combining them to produce the actual memory address. This step uses the sign extender and ALU.
3. *Read/Write from Memory* takes data or instructions from the data memory, and implements the first part of the *execute* step of the fetch/decode/execute cycle.
4. *Write into Register File* puts data or instructions into the data memory, implementing the second part of the *execute* step of the fetch/decode/execute cycle.

The load/store data path takes operand #1 (the base address) from the register file, and sign-extends the offset, which is obtained from the instruction input to the register file. The sign-extended offset and the base address are combined by the ALU to yield the memory address, which is input to the Address port of the data memory. The Mem Read signal is then activated, and the output data obtained from the Read Data port of the data memory is then written back to the Register File using its Write Data port, with Reg Write asserted.

#### 3, J format intructions-Branch/Jump Datapath

#### The branch data path (jump is an unconditional branch) uses instructions such as beq(branch on equal) t1, t2, offset, where *offset* is a 16-bit offset for computing the branch target address via PC-relative addressing. The beq instruction reads from registers t1 and t2, then compares the data obtained from these registers to see if they are equal. If equal, the branch is taken. Otherwise, the branch is not taken.

#### (By *taking the branch*, the ISA specification means that the ALU adds a sign-extended offset to the program counter (PC). The offset is shifted left 2 bits to allow for word alignment (since 22 = 4, and words are comprised of 4 bytes).)

#### When the condition is true (i.e., the operands are equal), the branch target address becomes the new PC, and we say that the *branch is taken*. If the operands are not equal, the incremented PC should replace the current PC (just as for any other normal instruction); in this case, we say that the *branch is not taken*.

#### Thus, the branch data path must do two operations: compute the branch target address and compare the register contents.

#### To compute the branch target address, the branch data path includes a sign extension unit, from and an adder. To perform the compare, we need to use the register file to supply the two register operands. ALU provides an output signal that indicates whether the result was 0, we can send the two register operands to the ALU with the control set to do a subtract. If the Zero signal out of the ALU unit is asserted, we know that the two values are equal. Although the Zero output always signals if the result is 0, we will be using it only to implement the equal test of branches.

#### 

#### The data path for a branch uses the ALU to evaluate the branch condition and a separate adder to compute the branch target as the sum of the incremented PC and the sign-extended, lower 16 bits of the instruction (the branch displacement), shifted left 2 bits.

#### <https://www.cise.ufl.edu/~mssz/CompOrg/CDA-proc.html>